PYTHON IN HIGH SCHOOL

ARNAUD BODIN

ALGORITHMS AND MATHEMATICS

Exo7

Python in high school

Let's go!

Everyone uses a computer, but it's another thing to drive it! Here you will learn the basics of programming. The objective of this book is twofold: to deepen mathematics through computer science and to master programming with the help of mathematics.

Python

Choosing a programming language to start with is tricky. You need a language with easy handling, well documented, with a large community of users. Python has all these qualities and more. It is modern, powerful and widely used, including by professional programmers.

Despite all these qualities, starting programming (with Python or another language) is difficult. The best thing is to already have experience with the code, using *Scratch* for example. There are still big steps to climb and this book is here to accompany you.

Objective

Mastering Python will allow you to easily learn other languages. Especially since the language is not the most important, the most important things are the algorithms. Algorithms are like cooking recipes, you have to follow the instructions step by step and what counts is the final result and not the language with which the recipe was written. This book is therefore neither a complete Python manual nor a computer course, nor is it about using Python as a super-calculator.

The aim is to discover algorithms, to learn step-by-step programming through mathematical/computer activities. This will allow you to put mathematics into practice with the willingness to limit yourself to the knowledge acquired during the first years.

Mathematics for computer science
Computer science for mathematics

Since computers only handle numbers, mathematics is essential to communicate with them. Another example is the graphical on-screen display that requires a good understanding of the coordinates (x, y), trigonometry....

Computers are a perfect match for mathematics! The computer becomes essential to manipulate very large numbers or to test conjecture on many cases. In this book you will discover fractals, L-systems, brownian trees and the beauty of complex mathematical phenomena.

You can retrieve all the activity `Python` codes and all the source files on the Exo7 *GitHub* page:
GitHub: Python in high school

Contents

Summary of the activities

Hello world!

Get into programming! In this very first activity, you will learn to manipulate numbers, variables and code your first loop with `Python`.

Turtle (Scratch with Python)

The `turtle` module allows you to easily make drawings in `Python`. It's about giving orders to a turtle with simple instructions like "go ahead", "turn"... It's the same principle as with *Scratch*, but with one difference: you no longer move blocks, instead you write the instructions.

If ... then ...

The computer can react according to a situation. If a condition is met, it acts in a certain way, otherwise it does something else.

Functions

Writing a function is the easiest way to group code for a particular task, in order to execute it once or several times later.

Arithmetic – While loop – I

The activities in this sheet focus on arithmetic: long division, prime numbers ... This is an opportunity to use the "while" loop intensively.

Strings – Analysis of a text

You're going to do some fun activities by manipulating strings and characters.

Lists I

A list is a way to group elements into a single object. After defining a list, you can retrieve each item of the list one by one, but also add new ones...

Statistics – Data visualization

It's good to know how to calculate the minimum, maximum, average and quartiles of a series. It's even better to visualize them all on the same graph!

Files

You will learn to read and write data with files.

Arithmetic – While loop – II

Our study of numbers is further developed with the "while" loop.

Binary I

The computers transform all data into numbers and manipulate only those numbers. These numbers are stored in the form of lists of 1's and 0's. It's the binary numeral system of numbers. To better understand this binary numeral system, you will first need to understand the decimal numeral system better.

Lists II

The lists are so useful that you have to know how to handle them in a simple and efficient way. That's the purpose of this chapter!

Binary II

We continue our exploration of the world of 1's and 0's.

Probabilities – Parrondo's paradox

You will program two simple games. When you play these games, you are more likely to lose than to win. However, when you play both games at the same time, you have a better chance of winning than losing! It's a paradoxical situation.

Find and replace

Finding and replacing are two very frequent tasks. Knowing how to use them and how they work will help you to be more effective.

Polish calculator – Stacks

You're going to program your own calculator! For that you will discover a new notation for formulas and also discover what a "stack" is in computer science.

Text viewer – Markdown

You will program a simple word processor that displays paragraphs cleanly and highlights words in bold and italics.

L-systems

L-systems offer a very simple way to code complex phenomena. From an initial word and a number of replacement operations, we arrive at complicated words. When you "draw" these words, you get beautiful fractal figures. The "L" comes from the botanist A. Lindenmayer who invented L-systems to model plants.

Dynamic images

We will distort images. By repeating these distortions, the images become blurred. But by a miracle after a certain number of repetitions the original image reappears!

Game of life

The *game of life* is a simple model of the evolution of a population of cells that split and die over time. The "game" consists of finding initial configurations that give interesting evolution: some groups of cells disappear, others stabilize, some move. . .

Ramsey graphs and combinatorics

You will see that a very simple problem, which concerns the relationships between only six people, will require a lot of calculations to be solved.

Bitcoin

The *bitcoin* is a dematerialized and decentralized currency. It is based on two computer principles: public key cryptography and proof of work. To understand this second principle, you will create a simple model of bitcoin.

Random blocks

You will program two methods to build figures that look like algae or corals. Each figure is made up of small randomly thrown blocks that stick together.

PART I
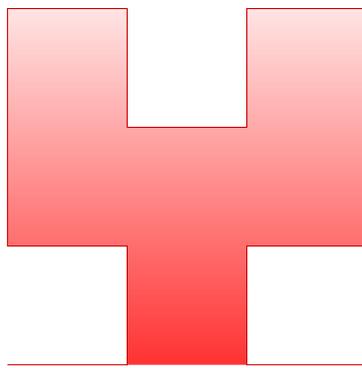
GETTING STARTED

Hello world!

Get into programming! In this very first activity, you will learn to manipulate numbers, variables and code your first loop with `Python`*.*

Lesson 1 (Numbers with `Python`).
Check that `Python` works correctly, by typing the following commands in the `Python` console:

```
>>> 2+2
>>> "Hello world!"
```

Here are some instructions to try.

- **Addition.** `5+7`.

- **Multiplication.** `6*7`; with brackets `3*(12+5)`; with decimal numbers `3*1.5`.

- **Power.** `3**2` for $3^2 = 9$; negative power `10**-3` for $10^{-3} = 0.001$.

- **Real division.** `14/4` is equal to `3.5`; `1/3` is equal to `0.3333333333333333`.

- **Integer division and modulo.**

  - `14//4` returns 3: it is the quotient of the Euclidean division of 14 by 4, note the double slash;

  - `14%4` returns 2: it is the remainder of the Euclidean division of 14 by 4, we also say "14 modulo 4".

Note. Inside the computer, decimals numbers are encoded as "floating point numbers".

Activity 1 (First steps).

Goal: code your first calculations with `Python`*.*

1. How many seconds are there in a century? (Do not take leap years into account.)
2. How far do you have to complete the dotted formula to obtain a number greater than one billion?
$$(1+2) \times (3+4) \times (5+6) \times (7+8) \times \cdots$$
3. What are the last three digits of
$$123456789 \times 123456789 \times 123456789 \times 123456789 \times 123456789 \times 123456789 \times 123456789 \quad ?$$
4. 7 is the first integer such that its inverse has a repeating decimal representation of period 6:
$$\frac{1}{7} = 0.\underbrace{142857}\,\underbrace{142857}\,\underbrace{142857}\ldots$$

   Find the first integer whose inverse has a repeating decimal representation of period 7:
$$\frac{1}{???} = 0.00\underbrace{abcdefg}\,\underbrace{abcdefg}\ldots$$

   Hint. The integer is bigger than 230!

5. Find the unique integer:
   - which gives a quotient of 107 when you divide it by 11 (with integer division),
   - and which gives a quotient of 90 when you divide it by 13 (with integer division),
   - and which gives a remainder equal to 6 modulo 7.

Lesson 2 (Working with an editor)**.**
From now on, it is better if you work in a text editor dedicated to `Python` rather than with the console. You must then explicitly ask to display the result:

```
print(2+2)
print("Hello world!")
```

In the following you continue to write your code in the editor but we will no longer indicate that you must use `print()` to display the results.

Lesson 3 (Variables)**.**
Variable. A *variable* is a name associated with a memory location. It is like a box that is identified by a label. The command "a = 3" means that I have a variable "a" associated with the value 3.
Here is a first example:

```
a = 3   # One variable
b = 5   # Another variable

print("The sum is",a+b)   # Display the sum
print("The product",a*b)  # Display the product

c = b**a      # New variable...
print(c)      # ... that is displayed
```

Comments. Any text following the hashtag character "#" is not executed by `Python` but is used to explain the program. It is a good habit to comment extensively on your code.

Names. It is very important to give a clear and precise name to the variables. For example, with the right names you should know what the following code calculates:

```
base = 8
height = 3
area = base * height / 2
print(area)
# print(Area)  # !! Error !!
```

Attention! `Python` is case sensitive. So `myvariable`, `Myvariable` and `MYVARIABLE` are different variables.

Re-assignment. Imagine you want to keep your daily accounts. You start with $S_0 = 1000$, the next day you earn 100, so now $S_1 = S_0 + 100$; the next day you add 200, so $S_2 = S_1 + 200$; then you lose 50, so on the third day $S_3 = S_2 - 50$. With `Python` you can use just one variable S for all these operations.

```
S = 1000
S = S + 100
S = S + 200
```
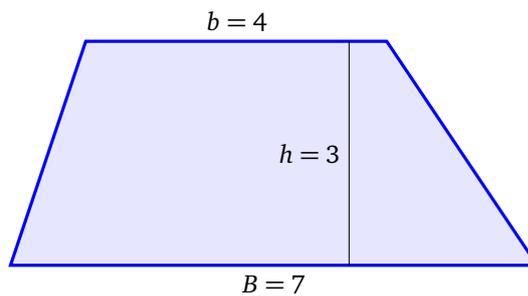
```
S = S - 50
print(S)
```

You have to understand the instruction "S = S + 100" like this: "I take the contents of the box S, I add 100, I put everything back in the same box".

Activity 2 (Variables).

Goal: use variables!

1. (a) Define variables, then calculate the area of a trapezoid. Your program should display "The value of the area is ..." using `print("The value of the area is",area)`.

$$b = 4$$
$$h = 3$$
$$B = 7$$

   (b) Define variables to calculate the volume of a box (a rectangular parallelepiped) whose dimensions are 10, 8, 3.
   (c) Define a variable PI equals to 3.14. Define a radius $R = 10$. Write the formula for the area of a disc of radius R.

2. Put the lines back in order so that, at the end, x has the value 46.

```
(1)   y = y - 1
(2)   y = 2*x
(3)   x = x + 3*y
(4)   x = 7
```

3. You place the sum of 1000 dollars in a savings account. Each year the interest on the money invested brings in 10% (the capital is multiplied by 1.10). Write the code to calculate the capital for the first three years.

4. I define two variables by a = 9 and b = 11. I would like to exchange the content of a and b. Which instructions should I use so that at the end a equals 11 and b equals 9?

```
a = b          c = b          c = a          c = a
b = a          a = b          a = b          a = c
               b = c          b = c          c = b
                                             b = c
```

Lesson 4 (Use functions).

- **Use** Python **functions.**
  You already know the `print()` function that displays a string of characters (or numbers). It can be used by `print("Hi there.")` or through a value:

```
string = "Hi there."
print(string)
```

There are many other functions. For example, the function `abs()` calculates the absolute value of a number: for example `abs(-3)` returns 3, `abs(5)` returns 5.

- **The module `math`.**
  Not all functions are directly accessible. They are often grouped into *modules*. For example, the `math` module contains mathematical functions. For instance, you will find the square root function `sqrt()`. Here's how to use it:

  ```
  from math import *

  x = sqrt(2)
  print(x)
  print(x**2)
  ```

  The first line imports all the functions of the module named `math`, the next lines calculate $x = \sqrt{2}$ (as an approximate value) and then display x and x^2.

- **Sine and cosine.**
  The `math` module contains the trigonometric functions sine and cosine and even the constant `pi` which is an approximate value of π. Be careful, the angles are expressed in radians.
  Here is the calculation of $\sin(\frac{\pi}{2})$.

  ```
  angle = pi/2
  print(angle)
  print(sin(angle))
  ```

- **Decimal to integer.**
  In the `math` module there are also functions to round a decimal number:
    - `round()` rounds to the nearest integer: `round(5.6)` returns 6, `round(1.5)` returns 2.
    - `floor()` returns the integer less than or equal to: `floor(5.6)` returns 5.
    - `ceil()` returns the integer greater than or equal to: `ceil(5.6)` returns 6.

Activity 3 (Use functions).

Goal: use functions from `Python` *and the* `math` *module.*

1. The `Python` function for gcd is `gcd(a,b)` (for greatest common divisor). Calculate the gcd of $a = 10\,403$ and $b = 10\,506$. Deduce the lcm from a and b. The function lcm does not exist, you must use the formula:
$$\text{lcm}(a, b) = \frac{a \times b}{\gcd(a, b)}.$$

2. By trial and error, find a real number x that checks all the following conditions (several solutions are possible):
   - `abs(x**2 - 15)` is less than `0.5`

- `round(2*x)` returns 8
- `floor(3*x)` returns 11
- `ceil(4*x)` returns 16
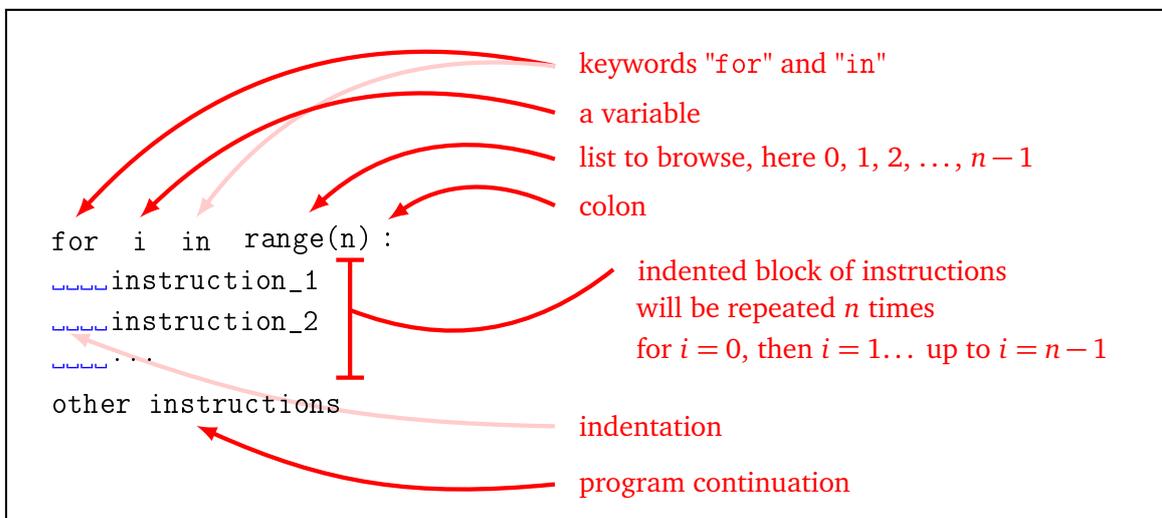
Hint. `abs()` refers to the absolute value function.

3. You know the trigonometric formula

$$\cos^2\theta + \sin^2\theta = 1.$$

Check that for $\theta = \frac{\pi}{7}$ (or other values) this formula is numerically true (this is not a proof of the formula, because `Python` only makes approximate computations of the sine and cosine).

Lesson 5 ("for" loop).

The "for" loop is the easiest way to repeat instructions.

Note that what delimits the block of instructions to be repeated is *indentation*, i.e. the spaces at the beginning of each line that shift the lines to the right. All lines in a block must have exactly the same indentation. In this book, we choose an indentation of 4 spaces.

Don't forget the colon ":" at the end of the line of the `for` declaration!

- **Example of a "for" loop.**
  Here is a loop that displays the squares of the first integers.

  ```
  for i in range(10):
      print(i*i)
  ```

  The second line is shifted and constitutes the block to be repeated. The variable `i` takes the value 0 and the instruction displays 0^2; then `i` takes the value 1, and the instruction displays 1^2; then 2^2, 3^2…
  In the end this program displays:

  $$0, 1, 4, 9, 16, 25, 36, 49, 64, 81.$$

  Warning: the last value taken by `i` is 9 (and not 10).

- **Browse any list.**
  The "for" loop allows you to browse any list. Here is a loop that displays the cube of the first prime numbers.

```
            for p in [2,3,5,7,11,13]:
                print(p**3)
```

- **Sum all.**
  Here is a program that calculates

$$0 + 1 + 2 + 3 + \cdots + 18 + 19.$$

```
mysum = 0
for i in range(20):
    mysum = mysum + i
print(mysum)
```

Understand this code well: a variable `mysum` is initialized at 0. We will add 0, then 1, then 2... This loop can be better understood by filling in a table:

Initialisation: `mysum`= 0

i	mysum
0	0
1	1
2	3
3	6
4	10
...	...
18	171
19	190

Display: 190

- `range()`.
  - With `range(n)` we run the entire range from 0 to $n-1$. For example `range(10)` corresponds to the list [0, 1, 2, 3, 4, 5, 6, 7, 8, 9].
    Attention! the list stops at $n-1$ and not at n. What to remember is that the list contains n items (because it starts at 0).
  - If you want to display the list of items browsed, you must use the command:
    $$\texttt{list(range(10))}$$
  - With `range(a,b)` we go through the elements from a to $b-1$. For example `range(10,20)` corresponds to the list [10, 11, 12, 13, 14, 15, 16, 17, 18, 19].
  - With `range(a,b,step)` you can browse the items $a, a+\text{step}, a+2\text{step}\ldots$ For example `range(10,20,2)` corresponds to the list [10, 12, 14, 16, 18].

- **Nested loops.**
  It is possible to nest loops, i.e. use a loop inside the block of another loop.

```
for x in [10,20,30,40,50]:
    for y in [3,7]:
        print(x+y)
```

In this small program x is first equal to 10, y takes the value 3 then the value 7 (so the program displays 13, then 17). Then $x = 20$, and y again equals 3, then 7 again (so the program displays 23, then 27). Finally the program displays:

$$13, 17, 23, 27, 33, 37, 43, 47, 53, 57.$$

Activity 4 ("for" loop).

Goal: build simple loops.

1. (a) Display the cubes of integers from 0 to 100.
   (b) Display the fourth powers of integers from 10 to 20.
   (c) Display the square roots of integers 0, 5, 10, 15,… up to 100.
2. Display the powers of 2, from 2^1 to 2^{10}, and memorize the results!
3. Experimentally search for a value close to the minimum of the function

$$f(x) = x^3 - x^2 - \frac{1}{4}x + 1$$

   on the interval $[0, 1]$.

   Hints.
   - Build a loop in which a variable i scans integers from 0 to 100.
   - Defined $x = \frac{i}{100}$. So $x = 0.00$, then $x = 0.01$, $x = 0.02…$
   - Calculate $y = x^3 - x^2 - \frac{1}{4}x + 1$.
   - Display the values using `print("x =",x, "y =",y)`.
   - Search by hand for which value of x you get the smallest possible y.
   - Feel free to modify your program to increase accuracy.
4. Seek an approximate value that must have the radius R of a ball so that its volume is 100.

   Hints.
   - Use a scanning method as in the previous question.
   - The formula for the volume of a ball is $V = \frac{4}{3}\pi R^3$.
   - Display values using `print("R =",R,"V =",V)`.
   - For π you can take the approximate value 3.14 or the approximate value `pi` of the `math` module.

Activity 5 ("for" loop (continued)).

Goal: build more complicated loops.

1. Define a variable n (for example $n = 20$). Calculate the sum

$$1^2 + 2^2 + 3^2 + \cdots + i^2 + \cdots + n^2.$$

2. Calculate the product:

$$1 \times 3 \times 5 \times \cdots \times 19.$$

   Hints. Begin by defining a `myproduct` variable initialized to the value 1. Use `range(a,b,2)` to get every other integer.
3. Display multiplication tables between 1 and 10. Here is an example of a line to display:

   ```
   7 x 9 = 63
   ```

   Use a display command of the style: `print(a,"x",b,"=",a*b)`.

Turtle (Scratch with Python)

The `turtle` *module allows you to easily make drawings in* `Python`*. It's about giving orders to a turtle with simple instructions like "go ahead", "turn"... It's the same principle as with* Scratch*, but with one difference: you no longer move blocks, instead you write the instructions.*
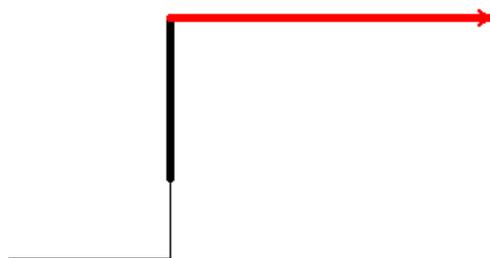
Lesson 1 (The Python turtle)**.**
Turtle is the ancestor of *Scratch*! In a few lines you can make beautiful drawings.

```python
from turtle import *

forward(100)    # Move forward
left(90)        # Turn 90 degrees left
forward(50)
width(5)        # Width of the pencil
forward(100)
color('red')
right(90)
forward(200)

exitonclick()
```

Here is a list of the main commands, accessible after writing:

```
from turtle import *
```

- `forward(length)` advances a number of steps

- `backward(length)` goes backwards

- `right(angle)` turns to the right (without advancing) at a given angle in degrees

- `left(angle)` turns left

- `setheading(direction)` points turtle in a direction (0 = right, 90 = top, −90 = bottom, 180 = left)
- `goto(x,y)` moves to the point (x, y)
- `setx(newx)` changes the value of the abscissa
- `sety(newy)` changes the value of the ordinate
- `down()` sets the pen down
- `up()` sets the pen up
- `width(size)` changes the thickness of the line
- `color(col)` changes the color: "red", "green", "blue", "orange", "purple"...
- `position()` returns the (x, y) position of the turtle
- `heading()` returns the direction `angle` to which the turtle is pointing
- `towards(x,y)` returns the angle between the horizontal and the segment starting at the turtle and ending at the point (x, y)
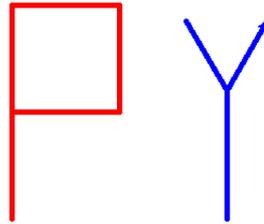- `exitonclick()` ends the program as soon as you click

The default screen coordinates range from −475 to +475 for x and from −400 to +400 for y; $(0, 0)$ is in the center of the screen.
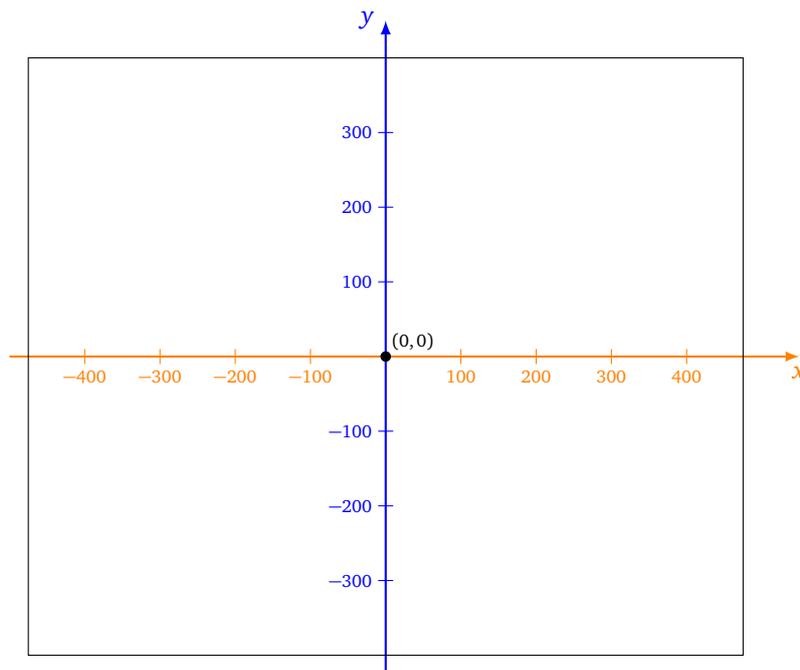
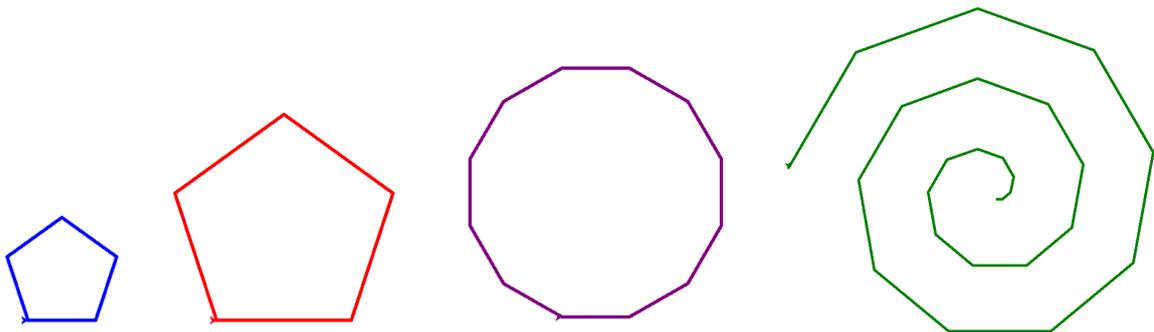
Activity 1 (First steps).

Goal: create your first drawings.

Trace the first letters of `Python`, for example as below.

Activity 2 (Figures).

Goal: draw geometric shapes.

1. **Pentagon.** Draw a first pentagon (in blue). You have to repeat 5 times: advance 100 steps, turn 72 degrees.
   Hint. To build a loop, use

   ```
   for i in range(5):
   ```
   (even if you do not use the variable i).
2. **Pentagon (bis).** Define a variable `length` which is equal to 200 and a variable `angle` which is equal to 72 degrees. Draw a second pentagon (in red), this time advancing by `length` and turning by `angle`.
3. **Dodecagon.** Draw a polygon having 12 sides (in purple).
   Hint. To draw a polygon with n sides, it is necessary to turn an angle of $360/n$ degrees.
4. **Spiral.** Draw a spiral (in green).
   Hint. Build a loop, in which you always turn at the same angle, but you move forward by a length that increases with each step.

Activity 3 (Function graph).

Goal: draw the graph of a function.
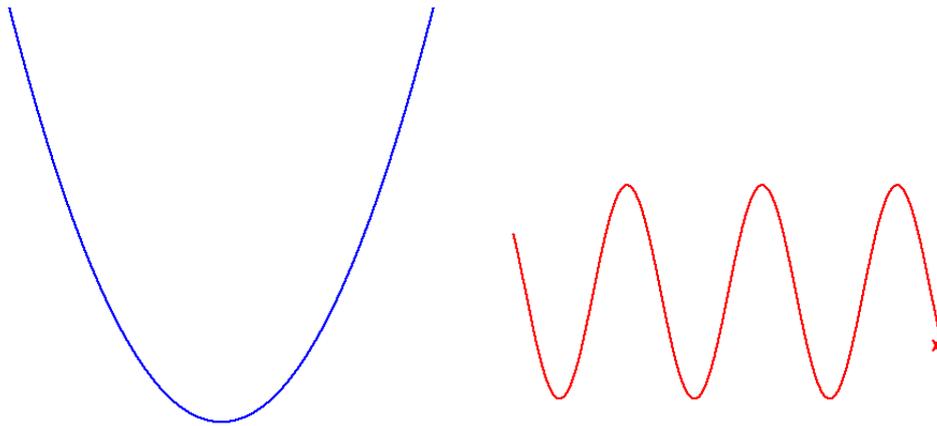
Plot the graph of the square function and the sine function.

In order to get a curve in the turtle window, repeat for x varying from -200 to $+200$:
- set $y = \frac{1}{100}x^2$,
- go to (x, y).

For the sinusoid, you can use the formula

$$y = 100 \sin\left(\frac{1}{20}x\right).$$

By default `Python` does not know the sine function, to use `sin()` you must first import the `math` module:

<div align="center">

`from math import *`

</div>

To make the turtle move faster, you can use the command `speed("fastest")`.

Activity 4 (Sierpinski triangle)**.**

Goal: trace the beginning of Sierpinski's fractal by nesting loops.
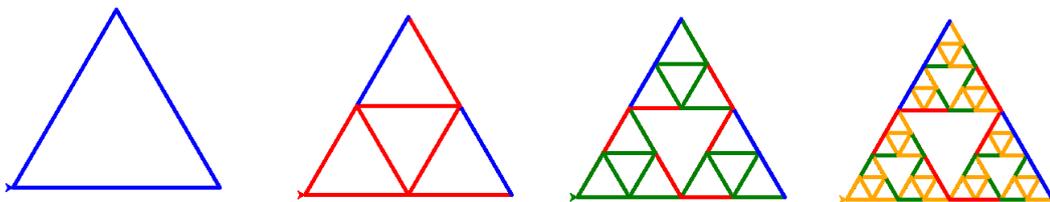
Here is how to picture the second drawing. Analyze the nesting of the loops and draw the next pictures.

```
for i in range(3):
    color("blue")
    forward(256)
    left(120)

    for i in range(3):
        color("red")
        forward(128)
        left(120)
```

Activity 5 (The heart of multiplication tables).

Goal: draw the multiplication tables.

We set an integer n. We are studying the 2 table, that is to say we calculate 2×0, 2×1, 2×2, up to $2 \times (n-1)$. In addition, the calculations will be modulo n. We therefore calculate

$$2 \times k \pmod{n} \qquad \text{for } k = 0, 1, \ldots, n-1$$

How do we draw this table?

We place n points on a circle, numbered from 0 to $n-1$. For each $k \in \{0, \ldots, n-1\}$, we connect the point number k with the point number $2 \times k \pmod{n}$ by a segment.

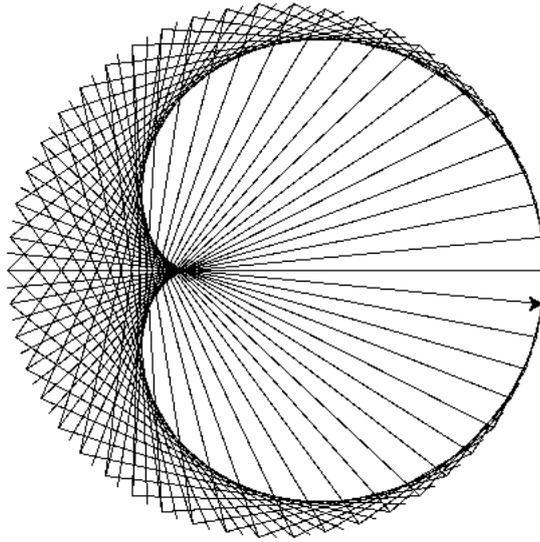Here is the layout, from the table of 2, modulo $n = 10$.
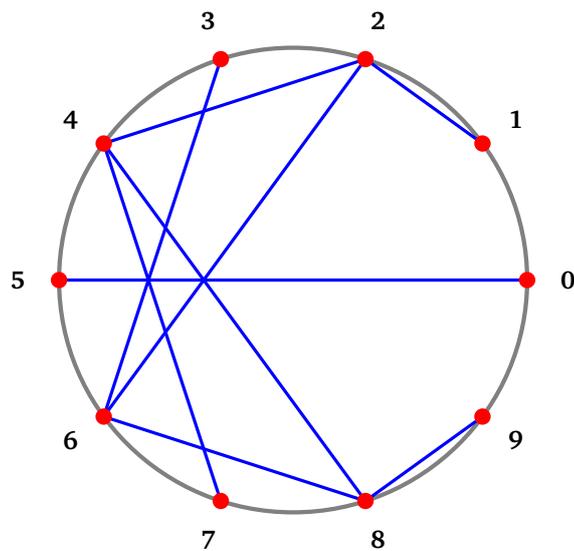
For example:

- the 3 point is linked to the 6 point, because $2 \times 3 = 6$;
- the 4 point is linked to the 8 point, because $2 \times 4 = 8$;
- the 7 point is linked to the 4 point, because $2 \times 7 = 14 = 4 \pmod{10}$.

Draw the table of 2 modulo n, for different values of n.

Here is what it gives for $n = 100$.

Hints. For calculations modulo n, use the expression `(2*k) % n`.

Here's how to get the coordinates of the vertices. This is done with the sine and cosine functions (available from the `math` module). The coordinates (x_i, y_i) of the vertex number i, can be calculated by the formula:

$$x_i = r \cos\left(\frac{2i\pi}{n}\right) \qquad \text{and} \qquad y_i = r \sin\left(\frac{2i\pi}{n}\right)$$

These points will be located on a circle of radius r, centered at $(0, 0)$. You will have to choose r rather large (for example $r = 200$).
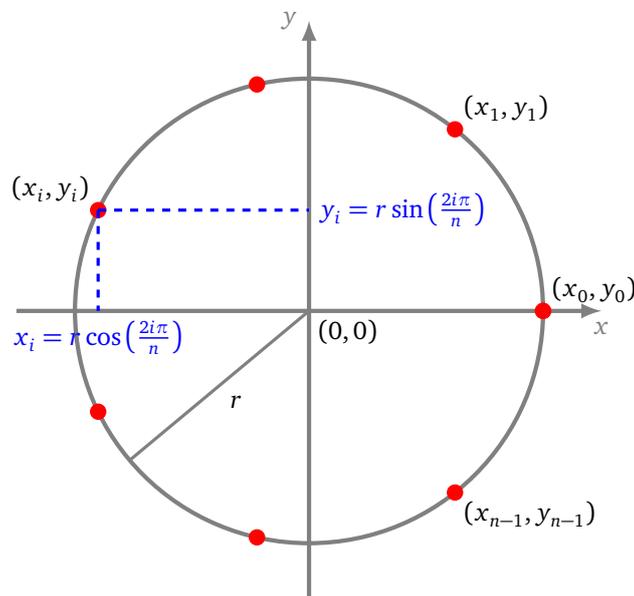
Lesson 2 (Several turtles)**.**

Several turtles can be defined and move independently. Here's how to define two turtles (one red and one blue) and move them.

```
turtle1 = Turtle()    # with capital 'T'!
turtle2 = Turtle()
```

```
turtle1.color('red')
turtle2.color('blue')

turtle1.forward(100)
turtle2.left(90)
turtle2.forward(100)
```

Activity 6 (The pursuit of turtles).

Goal: draw tracking curves.

Program four turtles running one after the other:
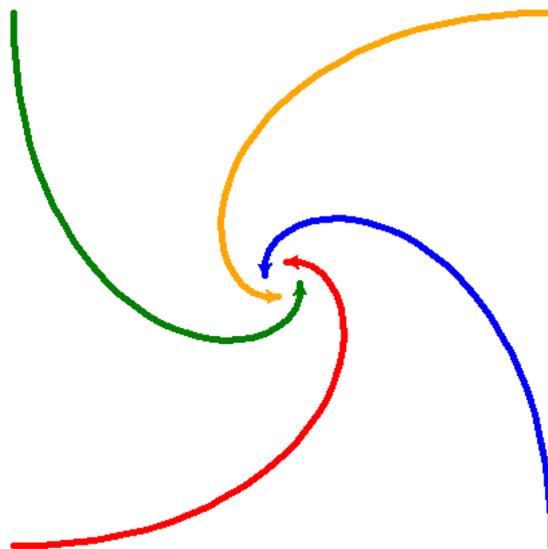
- turtle 1 runs after turtle 2,
- turtle 2 runs after turtle 3,
- turtle 3 runs after turtle 4,
- turtle 4 runs after turtle 1.

Here are the starting positions and orientations:
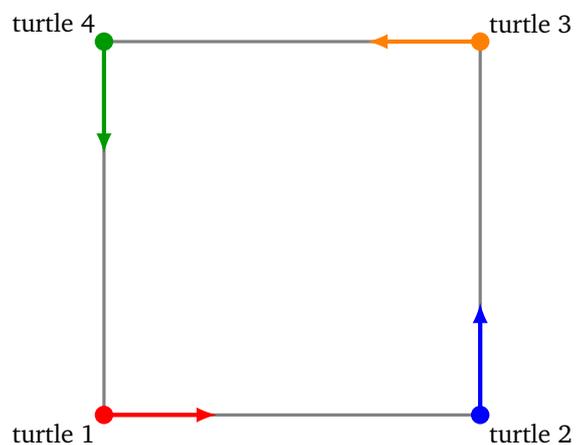
Hints. Use the following piece of code:

```
position1 = turtle1.position()
position2 = turtle2.position()
angle1 = turtle1.towards(position2)
turtle1.setheading(angle1)
```

- You place turtles at the four corners of a square, for example at $(-200, -200)$, $(200, -200)$, $(200, 200)$ and $(-200, 200)$.
- You get the position of the first turtle by using `position1 = turtle1.position()`. Same for the other turtles.
- You calculate the angle between turtle 1 and turtle 2 by the command `angle1 = turtle1.towards(position2)`.
- You orient the first turtle according to this angle: `turtle1.setheading(angle1)`.
- You advance the first turtle by 10 steps.

Improve your program by drawing a segment between the chasing turtle and the chased turtle each time.
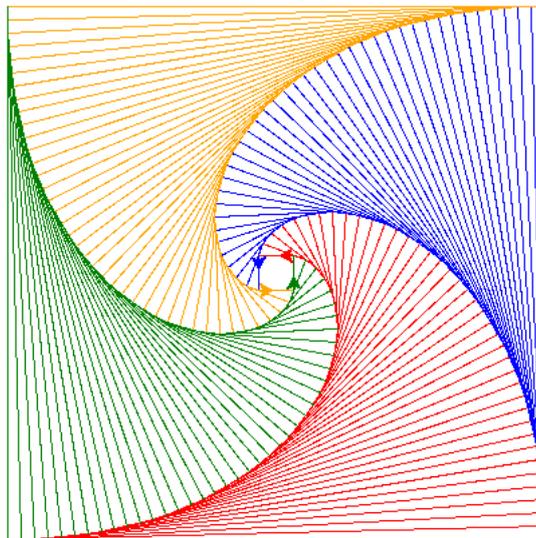
Part II
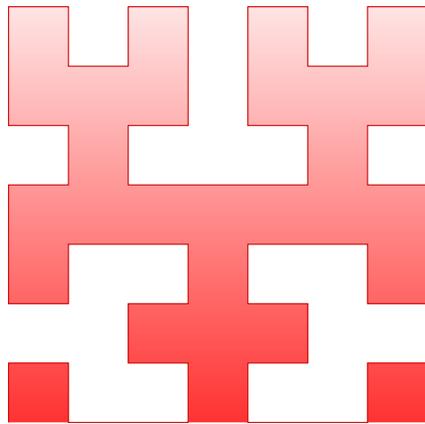
Basics

If ... then ...

The computer can react according to a situation. If a condition is met, it acts in a certain way, otherwise it does something else.

Lesson 1 (If ... then ...)**.**
Here's how to use the "`if`" test with `Python`:
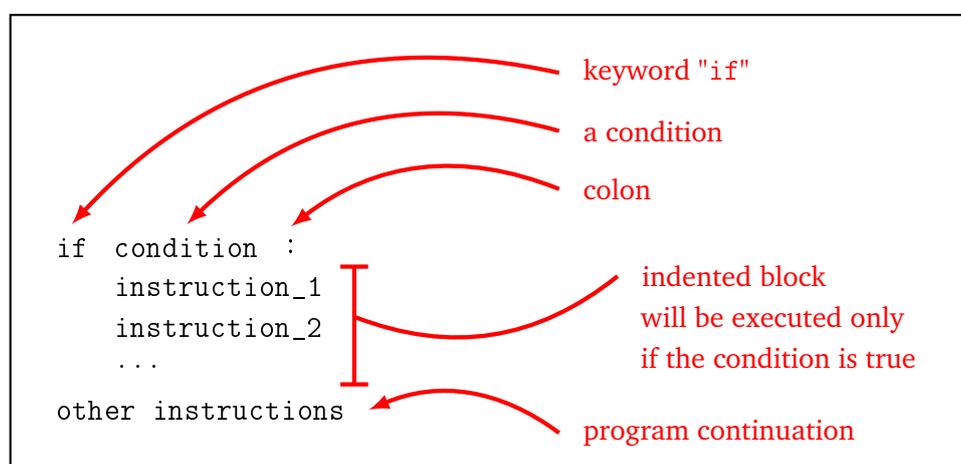
Here is an example, which warns a driver if a variable `speed` is too large.

```
if speed > 110:
    print("Warning, you are driving too fast.")
```

Instructions can also be executed if the condition is not met using the keyword "`else`".
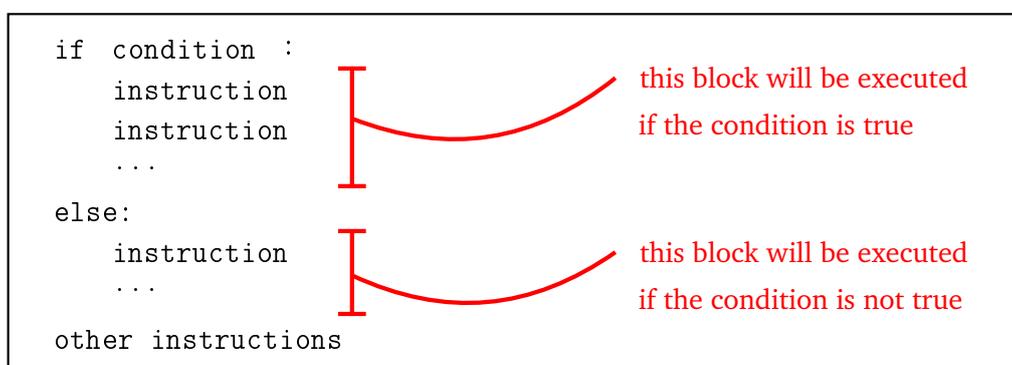
Once again, it is the indentation that delimits the different blocks of instructions. Here is an example that displays the sign of a number `x`.

```
        if x >= 0:
            print("The number is positive (or zero).")
        else:
            print("The number is negative.")
```

Lesson 2 (Keyboard entry).

To be able to interact with the user, you can ask him to enter text on the keyboard. Here is a small program that asks for the user's first name and age and displays a message like "Hello Kevin" then "You are a minor/adult!" according to age.

```
        first_name = input ("What's your name? ")
        print("Hello",first_name)

        age_str = input("How old are you? ")
        age = int(age_str)

        if age >= 18:
            print("You're an adult!")
        else:
            print("You're a minor!")
```

Explanations.
- The command `input()` pauses the execution of the program and waits for the user to send a text input (ended by pressing the "Enter" key).
- This command returns a string.
- If you want an integer, you have to convert the string. For example, here `age_str` can be equal to `"17"` (it is not a number but a sequence of characters), while `int(age_str)` is now the integer 17.
- The reverse operation is also possible, `str()` converts a number into a string. For example `str(17)` returns the string `"17"`; if you set `age = 17`, then `str(age)` also returns `"17"`.

Lesson 3 (The "random" module).

The `random` module generates numbers as if they were randomly drawn.
- Here is the command to place at the beginning of the program to call this module:
$$\text{from random import *}$$
- The `randint(a,b)` command returns a random integer between a and b.
  For example after the instruction `n = randint(1,6)`, n is a random integer such that $1 \leqslant n \leqslant 6$. If you repeat the instruction `n = randint(1,6)`, n takes a new value. It's like rolling a 6-face dice.
- The `random()` command, without argument, returns a floating number (i.e. a decimal number) between 0 and 1. For example, after the instruction `x = random()`, then x is a floating point number with $0 \leqslant x < 1$.

Activity 1 (Multiplication quiz).

   Goal: program a small multiplication tables test.

- Define a variable a, to which you assign a random value between 1 and 12.
- Same thing for a variable b.
- Display the question on the screen: "What is the product $a \times b$?". (Replace a and b by their value!)
- Retrieve the user's answer and transform it into an integer.
- If the answer is correct, display "Well done!", otherwise display "Lost! The correct answer was ...".

Test of equality. To test if two numbers x and y are equal, the instruction is:

```
if x == y:
```

The equality test is written with the double equal symbol "==". For example "x == 3" returns "True" if x is equal to 3 and "False" otherwise.

Attention! The instruction "x = 3" has nothing to do with testing equality, this instruction stores 3 in the variable x.

Activity 2 (The words of the turtle).

   Goal: guide the turtle with a word, each character corresponding to an instruction.

You give the turtle a word, for example **FlFrfflFrfF**, in which each character (read from left to right) corresponds to an instruction that the turtle must execute.

- **F** : go forward of 100 by tracing,
- **f** : go forward of 100 without tracing,
- **l** : turn left by 90 degrees,
- **r** : turn right by 90 degrees.

Example. Here is the drawing that the turtle must trace when

```
word = "FflFlFrFlFFlfFF"
```


Hints. Here's how to scan the letters of a word and test if a letter is the character **F**:

```
for c in word:
    if c == "F":
        instructions...
```

Finally remember that `up()` and `down()` set the position of the pen.

Lesson 4 (Booleans)**.**

- A *boolean* is a type of data that is either equal to the value "True" or the value "False". In `Python` the values are `True` and `False` (with a capital letter).

- We can obtain a boolean as a result of the comparison of two numbers. For example `7 < 4` is equal to `False` (because 7 is not smaller than 4). Check that `print(7 < 4)` displays `False`.
  Here are the main comparisons:
  - **Test of equality:** `a == b`
  - **Strict lower test:** `a < b`
  - **Large lower test:** `a <= b`
  - **Higher test:** `a > b` or `a >= b`
  - **Test of non equality:** `a != b`

  For example `6*7 == 42` is equal to `True`.

-

> **ATTENTION!** The classic mistake is to confuse "a = b" and "a == b".
> - **Assignment.** `a = b` puts the content of the variable b in the variable a.
> - **Test of equality.** `a == b` tests if the contents of a and b are the same, and is equal to `True` or `False`.

- We can compare something other than numbers. For example, "`char == "A"`" tests if the variable `char` is equal to `"A"`; "`its_raining == True`" tests if the variable `its_raining` is true...

- Booleans are useful in "if … then …" tests and in "while … then …" loops.

- **Operations between booleans.** If P and Q are two booleans, new booleans can be defined.
  - **Logical and.** "`P and Q`" is true if and only if P and Q are true.
  - **Logical or.** "`P or Q`" is true if and only if P or Q is true.
  - **Negation.** "`not P`" is true if and only if P is false.

  For example "`(2+2 == 2*2) and (5 < 3)`" returns `False`, because even if we have $2+2 = 2\times2$, the other condition is not satisfied because $5 < 3$ is wrong.

Activity 3 (Digits of an integer)**.**

Goal: find numbers whose digits verify certain properties.

1. The following program displays all integers from 0 to 99. Understand this program. What do the variables u and t represent?

```
for t in range(10):
    for u in range(10):
        n = 10*t + u
        print(n)
```

2. Find all integers between 0 and 999 that satisfy all the following properties:
   - the integer ends with a 3,
   - the sum of the digits is greater than or equal to 15,
   - the tens digit is even.

3. Modify your previous program to count and display the number of integers that satisfy these properties.

Activity 4 (Triangles)**.**

Goal: determine the properties of a triangle from the three lengths of the sides.

We give ourselves the three lengths a, b and c. You will determine the properties of the triangle whose lengths would be a, b, c.
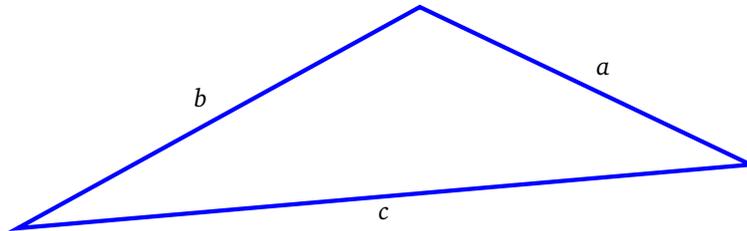
Define three variables a, b and c with integer values and $a \leqslant b \leqslant c$ (or ask the user for three values).

1. **Order.** Ask `Python` to test if the lengths satisfy $a \leqslant b \leqslant c$. Display a sentence for the answer.
2. **Existence.** There is a triangle corresponding to these lengths if and only if:
$$a + b \geqslant c.$$
   Ask `Python` to test if this is the case and display the answer.
3. **Right triangle.** Ask `Python` to test if the triangle is a right triangle. (Think of Pythagorean theorem.)
4. **Equilateral triangle.** Test if the triangle is equilateral.
5. **Isosceles triangle.** Test if the triangle is isosceles.
6. **Acute triangle.** Test if all angles are acute (i.e. less than or equal to 90 degrees).
   Hints.
   - The cosine law allows us to calculate an angle according to the lengths:
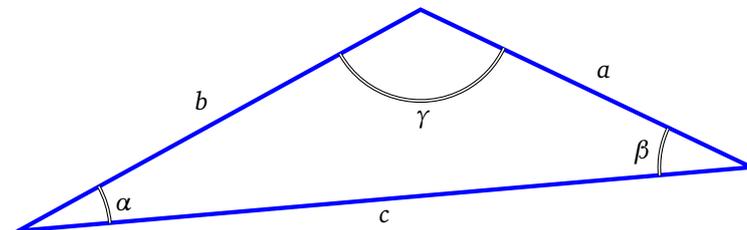
$$\cos \alpha = \frac{-a^2 + b^2 + c^2}{2bc}, \qquad \cos \beta = \frac{a^2 - b^2 + c^2}{2ac}, \qquad \cos \gamma = \frac{a^2 + b^2 - c^2}{2ab}.$$

   - To test if the angle α is acute just check $\cos \alpha \geqslant 0$ (in the end we never calculate α, but only $\cos \alpha$).

Find examples of lengths a, b, c to illustrate the different properties.

Activity 5 (The mystery number)**.**

Goal: code the classic game when learning to program. The computer chooses a random number, the user must guess this number by following the indications "larger" or "smaller" given by the computer. As this game is quickly boring, we introduce variants where the computer is allowed to lie or cheat!

1. **The classic game.**
   - The computer randomly chooses a mystery number between 0 and 99.

- The player offers an answer.
- The computer replies "the number to find is greater" or "the number to find is smaller" or "bravo, it's the right number!".
- The player has seven attempts to find the right answer.

Program this game!

Hints. To leave a `for` loop before the last proposal, you can use the command `break`. Use this when the player finds the right answer.

2. **The computer is lying.**

   To complicate the game, the computer has the right to lie from time to time. For example, about one in four times the computer can give the wrong hint of "larger" or "smaller".

   Hints. To decide when the computer is lying, each turn, draw a random number between 1 and 4, if it is 4 the computer will lie!

3. **The computer is cheating.**

   Now the computer is cheating (but it no longer lies)! Each turn the computer changes the mystery number a little bit.

   Hints. Each round, draw a random number, between −3 and +3 for example, and add it to the mystery number. (Be careful not to exceed the 0 and 99 limits.)

Functions

Writing a function is the easiest way to group code for a particular task, in order to execute it once or several times later.

Lesson 1 (Function (start)).

A computer function is a portion of code that performs a specific task and can be used one or more times during the rest of the program. Defining a function in `Python` is very simple. Here are two examples:

```
def say_hello():
    print("Hello world!")
    return
```

```
def print_squares():
    for i in range(20):
        print(i**2)
    return
```

The instructions are grouped into an indented block. The word `return` (optional) indicates the end of the function. These instructions are executed only if I call the function. For example, each time I execute the `say_hello()` command, `Python` displays the sentence "Hello world!". Each time I execute the `print_squares()` command, `Python` displays $0, 1, 4, 9, 16, \ldots$, i.e. the numbers i^2 for $i = 0, \ldots, 19$.
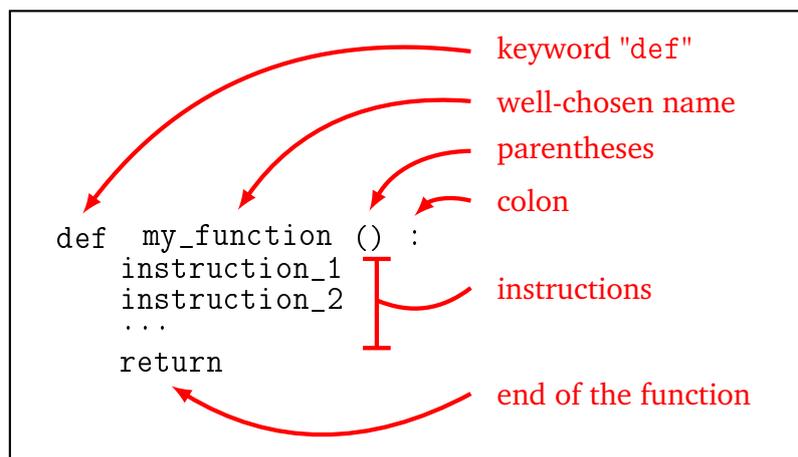
Lesson 2 (Function (continued)).

Functions achieve their full potential with:

- an *input*, which defines variables that serve as *parameters*,

- an *output*, which is a result returned by the function (and which will often depend on the input parameters).

Here are two examples:

```
def display_month(number):
    if number == 1:
        print("We are in January.")
    if number == 2:
        print("We are in February.")
    if number == 3:
        print("We are in March.")
    # etc.
    return
```

When called, this function displays the name of the month based on the number provided as input. For example `display_month(3)` will display `"We are in March."`.

```
def compute_cube(a):
    cube = a * a * a     # or a**3
    return cube
```

This function calculates the cube of a number, for example `compute_cube(2)` does not display anything but returns the value 8. This value can be used elsewhere in the program. For example, what do the following instructions do?

```
x = 3
y = 4
z = compute_cube(x) + compute_cube(y)
print(z)
```

In mathematical terms, we set $x = 3$, $y = 4$, then we calculated the cube of x, the cube of y and added them up:

$$z = x^3 + y^3 = 3^3 + 4^3 = 27 + 64 = 91$$

Thus the program displays 91.

The advantages of programming using functions are as follows:

- you write the code for a function only once, but you can call the function several times;

- by dividing our program into small blocks, each with its own use, the program is easier to write, read, correct, and modify;

- you can use a function written by someone else (such as the `sqrt()` function) without knowing all the internal details of its programming.

Activity 1 (First functions).

Goal: write very simple functions.

1. **Functions without parameters or outputs.**
   (a) Define a function called `print_table_of_7()` that displays the multiplication table by 7: $1 \times 7 = 7$, $2 \times 7 = 14$...
   (b) Define a function called `print_hello()` that asks the user for their first name and then displays "Hello" followed by their name.
   Hint. Use `input()`.

2. **Functions with one parameter and no outputs.**
   (a) Define a function called `print_a_table(n)` that depends on a parameter `n` and displays the multiplication table of this integer n. For example, the command `print_a_table(5)` must display: $1 \times 5 = 5$, $2 \times 5 = 10$...
   (b) Define a function called `say_greeting(sentence)` that depends on a parameter `sentence`. This function asks for the user's first name and displays the sentence followed by the first name. For example, `say_greeting("Hi")` would display "Hi" followed by the first name given by the user.

3. **Functions without parameters and with an output.**
   Define a function called `ask_full_name()` that first asks for the user's first name, then their last name and returns the complete identity with the last name in upper case as a result. For example, if the user enters "Darth" then "Vader", the function returns the string `"Darth VADER"` (the function displays nothing).
   Hints.
   - If `word` is a string, then `word.upper()` is the transformed string with characters in capital letters. Example: if `word = "Vader"` then `word.upper()` returns `"VADER"`.
   - You can merge two strings by using the operator "+". Example: `"Darth" + "Vader"` is equal to `"DarthVader"`. Another example: if `string1 = "Darth"` and `string2 = "Vader"` then `string1 + " " + string2` is equal to `"Darth Vader"`.
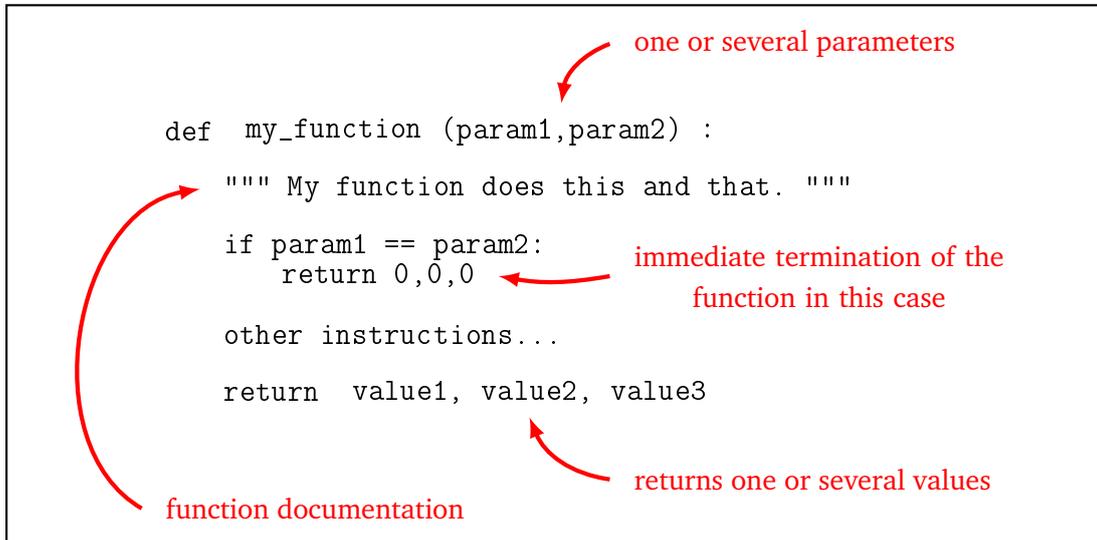
Lesson 3 (Function (continuation and end for now)).
A function can have several parameters and can return several results. For example, here is a function that calculates and returns the sum and product of two given input numbers.

```
def sum_product(a,b):
    """ Computes the sum and product of two numbers. """
    s = a + b
    p = a * b
    return s, p
```

```
mysum, myprod = sum_product(6,7)
```

The last line calls the function with arguments 6 (for parameter `a`) and 7 (for parameter `b`). This function returns two values, the first one is assigned to `mysum` (which is therefore equal to 13) and the second one to `myprod` (which is equal to 42).

```
def  my_function (param1,param2) :          one or several parameters
    """ My function does this and that. """
    if param1 == param2:
        return 0,0,0          immediate termination of the
                              function in this case
    other instructions...
    return  value1, value2, value3
                              returns one or several values
    function documentation
```

So let's remember:

- There can be several input parameters.
- There can be several results at the output.
- Very important! Do not confuse displaying and returning a value. Display (by the command `print()`) just displays something on the screen. Most functions do not display anything, but instead return one (or more) value. This is much more useful because this value can then be used elsewhere in the program.
- As soon as the program encounters the instruction `return`, the function stops and returns the result. There may be several instances of the `return` instruction in a function but only one will be executed. It is also possible not to put an instruction `return` if the function returns nothing.
- You can, of course, call other functions in the body of your function!
- It is important to write lots of comments about your code. To document a function, you can describe what it does starting with a *docstring*, i.e. a description (in English) surrounded by three quotation marks:

```
""" My function does this and that. """
```

to be placed just after the header.

- In the definition of a function, the variables that appear between the parentheses are called the *parameters*; in a function call, however, the values between the parentheses are called the *arguments*. There is of course a correspondence between the two.

Activity 2 (More features)**.**

Goal: build functions with different types of input and output.

1. **Trinomials.**
    (a) Write a function `trinomial_1(x)` that depends on a parameter `x` and returns the value of

the trinomial $3x^2 - 7x + 4$. For example, `trinomial_1(7)` returns 102.

(b) Write a function `trinomial_2(a,b,c,x)` that depends on four parameters a, b, c and x and returns the value of the trinomial $ax^2 + bx + c$. For example, `trinomial_2(2,-1,0,6)` returns 66.

2. **Currencies.**

(a) Write a function `conversion_dollars_to_euros(amount)` which takes in a sum of money `amount`, expressed in dollars and returns its value in euros (for example 1 dollar = 0.89 euro).

(b) Write a function `conversion_dollars(amount,currency)` which depends on two parameters `amount` and `currency` and converts the amount given in dollars, to the desired currency. Examples of currencies: 1 dollar = 0.89 euros; 1 dollar = 0.77 pounds; 1 dollar = 110 yen. For instance, `conversion_dollars(100,"pound")` returns 77.

Make sure to give meaningful names to your functions and variables. Don't forget to document each function by adding a small explanatory text between triple quotation marks at the very beginning of your function.

3. **Volumes.**

Build functions that calculate and return volumes:

- the volume of a cube according to the length of one side,

- the volume of a ball according to its radius,

- the volume of a cylinder according to the radius of its base and its height,

- the volume of a rectangular parallelepiped box according to its three dimensions.

For the value of π, you should use either the approximate value 3.14, or the approximate value provided by the constant `pi` in the `math` module.

4. **Perimeters and areas.**

(a) Write a function whose use is `perimeter_area_rectangle(a,b)` and which returns the perimeter and area of a rectangle with dimensions a and b.

(b) Same question with `perimeter_area_disc(r)` for the perimeter and area of a disc of radius r.

(c) Use your previous function to guess the radius, for which the area of a disc is larger than the perimeter of that disc.

Hint. If you want to scan the radius by incrementing the value of 0.1 each time, you can build a loop as follows:

```
for R in range(0,30):
```

then make a call to the function by `perimeter_area_disc(R/10)`.

Activity 3 (Turtle).

Goal: define functions that draw geometric shapes. Creating a function is similar to creating a block in Scratch.
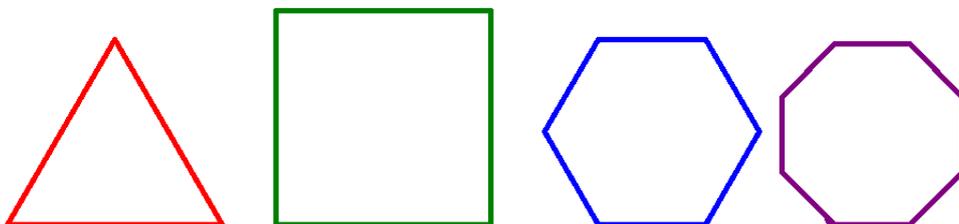
1. Program a `triangle()` function that draws a triangle (in red, each side measuring 200).
2. Program a `square()` function that draws a square (in green, each side measuring 200). Use a "for" loop so you don't have to rewrite the same instructions several times.
3. Program a `hexagon(length)` function that draws a hexagon (in blue) of a given side length (the angle to turn is 60 degrees).
4. Program a `polygon(n,length)` function that draws a regular polygon of n sides and a given side length (the angle to rotate is $360/n$ degrees).

Activity 4 (Functions again).

Goal: create new functions.

1. (a) Here is the discount for the price of a train ticket based on the age of the passenger:
   - reduction of 50% for those under 10 years old;
   - reduction of 30% for 10 to 18 years old;
   - reduction of 20% for 60 years old and over.

   Write a function `reduction()` that returns the reduction according to age. The function properties are described in the box below:

reduction()
Use: `reduction(age)` Input: an integer corresponding to age Output: an integer corresponding to the reduction
Examples: • `reduction(17)` returns 30. • `reduction(23)` returns 0.

   Deduce an `amount()` function that calculates the amount to be paid based on the normal fare and the traveler's age.

amount()
Use: `amount(normal_rate,age)` Input: a number `normal_rate` corresponding to the price without discount and `age` (an integer) Output: a number corresponding to the amount to be paid after reduction Note: uses the function `reduction()`
Example: `amount(100,17)` returns 70.

   A family buys tickets for different trips, here is the normal fare for each trip and the ages of the passengers:
   - normal price 30 dollars, child of 9 years old;
   - normal price 20 dollars, for each of the twins of 16 years old;
   - normal price 35 dollars, for each parent of 40 years old.

   What is the total amount paid by the family?

2. We want to program a small multiplication tables quiz.
   (a) Program a function `is_calculation_correct()` that decides if the answer given to a multiplication is right or not.

> **is_calculation_correct()**
>
> Use: `is_calculation_correct(a,b,answer)`
> Input: three integers, `answer` being the proposed answer to the calculation of $a \times b$.
> Output: "True" or "False", depending on whether the answer is correct or not
>
> Examples:
> - `is_calculation_correct(6,7,35)` returns `False`.
> - `is_calculation_correct(6,7,42)` returns `True`.

(b) Program a function that displays a multiplication, asks for an answer and displays a short concluding sentence. All this in English or a foreign language!

> **test_multiplication()**
>
> Use: `test_multiplication(a,b,lang)`
> Input: two integers, the chosen language (for example `"english"` or `"french"`)
> Output: nothing (but display a sentence)
> Note: uses the function `is_calculation_correct()`
>
> Example: `test_multiplication(6,7,"french")` asks, in French, for the answer to the calculation 6×7 and answers if it is correct or not.

Bonus. Improve your program so that the computer offers random operations to the player. (Use the `randint()` function of the `random` module.)

Activity 5 (Experimental equality).

Goal: use the computer to experiment with equality of functions.

1. (a) Build an `absolute_value(x)` function that returns the absolute value of a number (without using the `abs()` function in `Python`).
   (b) Build a `root_of_square(x)` function which corresponds to the calculation of $\sqrt{x^2}$.
   (c) Two mathematical functions (of one variable) f and g are said to be *experimentally equal* if $f(i) = g(i)$ for $i = -100, -99, \ldots, 0, 1, 2, \ldots, 100$. Check by computer that the two functions defined by
   $$|x| \quad \text{and} \quad \sqrt{x^2}$$
   are experimentally equal.

2. (a) Build a two-parameter function `F(a,b)` that returns $(a+b)^2$. Same thing with `G(a,b)` that returns $a^2 + 2ab + b^2$.
   (b) Two functions of two variables F and G are said to be *experimentally equal* if $F(i,j) = G(i,j)$ for all $i = -100, -99, \ldots, 100$ and for all $j = -100, -99, \ldots, 100$. Check by computer that the functions $(a+b)^2$ and $a^2 + 2ab + b^2$ you defined are experimentally equal.
   (c) I know that one of the following two identities is true:
   $$(a-b)^3 = a^3 - 3a^2b - 3ab^2 + b^3 \quad \text{or} \quad (a-b)^3 = a^3 - 3a^2b + 3ab^2 - b^3.$$
   Use the computer to help you to decide which one it is!

3. (a) Build a function `sincos(x)` that returns $(\sin(x))^2 + (\cos(x))^2$ and another `one(x)` that always returns 1. Are these two functions experimentally equal (in the sense of the first question)? Find out why that is.

(b) Fix $\epsilon = 0.00001$. It is said that two functions (of one variable) f and g are *experimentally approximately equal* if $|f(i) - g(i)| \leqslant \epsilon$ for $i = -100, -99, \ldots, 100$. Do the two functions defined by `sincos(x)` and `one(x)` now satisfy this criterion?

(c) Experimentally check and experimentally approximately check the identities:
$$\sin(2x) = 2\sin(x)\cos(x), \qquad \cos\left(\frac{\pi}{2} - x\right) = \sin(x).$$

(d) **Bonus. A counter-example.** Show that the functions defined by $g_1(x) = \sin(\pi x)$ and $g_2(x) = 0$ are experimentally equal (with our definition given above). But also show that you don't get $g_1(x) = g_2(x)$ for every $x \in \mathbb{R}$.

Lesson 4 (Local variable).

Here is a very simple function that takes a number as an input and returns the number increased by one.
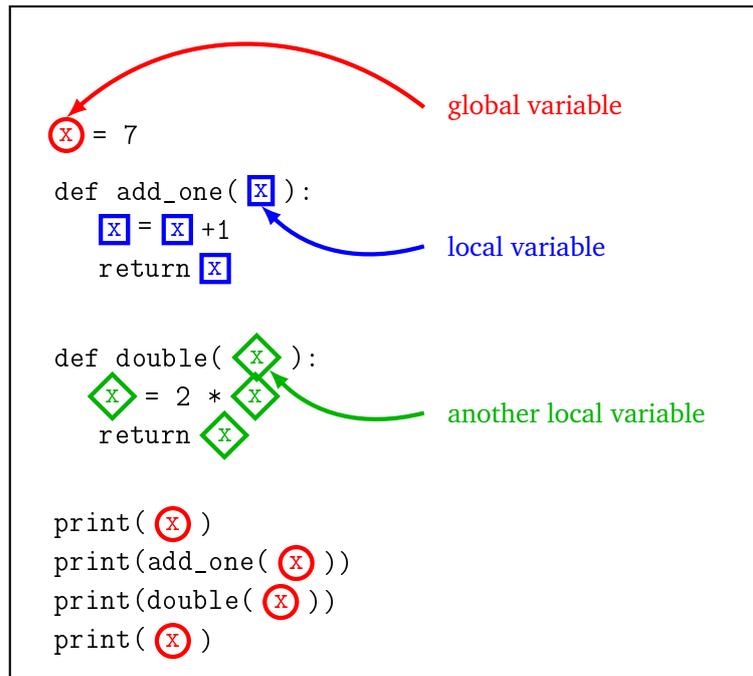
```
def my_function(x):
    x = x + 1
    return x
```

- Of course `my_function(3)` returns 4.
- If I define a variable by `y = 5` then `my_function(y)` returns 6. And the value of `y` has not changed, it is still equal to 5.
- Here is the delicate situation that you must understand:

```
x = 7
print(my_function(x))
print(x)
```

  - The variable `x` is initialized to 7.
  - The call of the function `my_function(x)` is therefore the same as `my_function(7)` and logically returns 8.
  - What is the value of `x` at the end? The variable `x` is unchanged and is still equal to 7! Even if in the meantime there has been an instruction `x = x + 1`. This instruction changed the `x` inside the function, but not the `x` outside the function.

- Variables defined within a function are called *local variables*. They do not exist outside the function.
- If there is a variable in a function that has the same name as a variable in the program (like the `x` in the example above), it is as if there were two distinct variables; the local variable only exists inside the function.

To understand the scope of the variables, you can color the global variables of a function in red, and the local variables with one color per function. The following small program defines two functions. The first adds one to a number and the second calculates the double.

```
x = 7

def add_one( x ):
    x = x +1
    return x

def double( x ):
    x = 2 * x
    return x

print( x )
print(add_one( x ))
print(double( x ))
print( x )
```

global variable

local variable

another local variable

The program first displays the value of x, or 7, then it increases it by 1, so it displays 8, then it displays twice as much as x, so 14. The global variable x has never changed, so the last display of x is still 7.

It is still possible to force the hand of Python and modify a global variable in a function using the keyword global. See the "Polish calculator – Stacks" chapter.

Arithmetic – While loop – I

The activities in this sheet focus on arithmetic: long division, prime numbers . . . This is an opportunity to use the "while" loop intensively.

Lesson 1 (Arithmetic)**.**

Let us recall what Euclidean division is. Here is the division of a by b, a is a positive integer, b is a strictly positive integer (with the example of 100 divided by 7):

quotient

q

b $\quad a$

remainder

r

14

7 $\quad$ 100

2
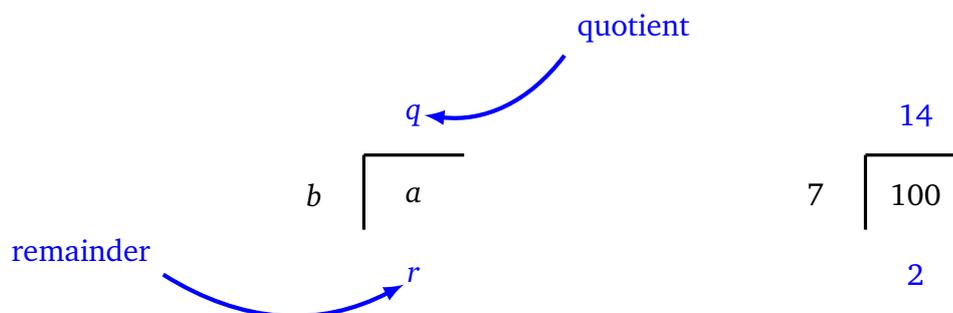
We have the two fundamental properties that define q and r:

$$a = b \times q + r \qquad \text{and} \qquad 0 \leqslant r < b$$

For example, for the division of $a = 100$ by $b = 7$: we have the quotient $q = 14$ and the remainder $r = 2$ that verify $a = b \times q + r$ because $100 = 7 \times 14 + 2$ and also $r < b$ because $2 < 7$.

> With `Python`:
> - `a // b` returns the quotient,
> - `a % b` returns the remainder.

It is easy to check that:

b is a divisor of a if and only if $r = 0$.

Activity 1 (Quotient, remainder, divisibility)**.**

Goal: use the remainder to find out if one integer divides another.

1. Program a function named `quotient_remainder(a,b)` that does the following tasks for two integers $a \geqslant 0$ and $b > 0$:
    - It displays the quotient q of the Euclidean division of a per b,
    - it displays the remainder r of this division,

- it displays `True` if the remainder r is positive or zero and strictly less than b, and `False` otherwise,

- it displays `True` if you have equality $a = bq + r$, and `False` if not.

Here is an example of what the call should display for `quotient_remainder(100,7)`:

```
Division of a = 100 by b = 7
The quotient is q = 14
The remainder is r = 2
Check remainder: 0 <= r < b? True
Check equality: a = bq + r? True
```

Note. You have to check without cheating that we have $0 \leqslant r < b$ and $a = bq + r$, but of course it must always be true!

2. Program a function called `is_even(n)` that tests if the integer n is even or not. The function should return `True` or `False`.

   Hints.

   - First possibility: calculate `n % 2`.

   - Second possibility: calculate `n % 10` (which returns the digit of units).

   - The smartest people will be able to write the function with only two lines (one for `def`... and the other for `return`...).

3. Program a function called `is_divisible(a,b)` that tests if b divides a. The function should return `True` or `False`.

Lesson 2 ("while" loop).

The "while" loop executes instructions as long as a condition is true. As soon as the condition becomes false, it proceeds to the next instructions.
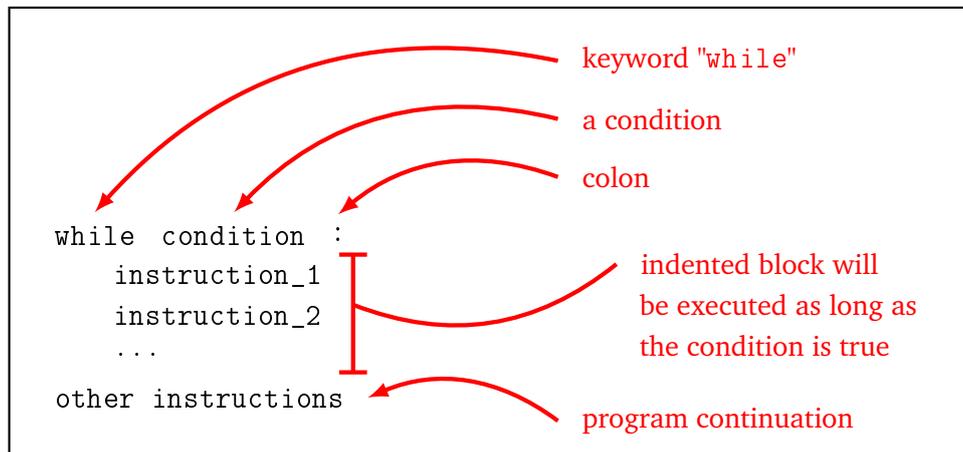
Example.

Here is a program that displays the countdown $10, 9, 8, \ldots 3, 2, 1, 0$. As long as the condition $n \geqslant 0$ is true, we reduce n by 1. The last value displayed is $n = 0$, because then $n = -1$ and the condition "$n \geqslant 0$" becomes false so the loop stops.

```
n = 10
while n >= 0:
    print(n)
    n = n - 1
```

This is summarized in the form of a table:

Input: $n = 10$

n	"$n \geqslant 0$" ?	new value of n
10	yes	9
9	yes	8
...	...	...
1	yes	0
0	yes	-1
-1	no	

Display: $10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0$

Example.

This piece of code looks for the first power of 2 greater than a given integer n. The loop prints the values 2, 4, 8, 16,... It stops as soon as the power of 2 is higher or equal to n, so this program displays 128.

```
n = 100
p = 1
while p < n:
    p = 2 * p
print(p)
```

Inputs: $n = 100$, $p = 1$

p	"$p < n$" ?	new value of p
1	yes	2
2	yes	4
4	yes	8
8	yes	16
16	yes	32
32	yes	64
64	yes	128
128	no	

Display: 128

Example.
For this last loop we have already prepared a function
called `is_even(n)` which returns `True` if the integer n
is even and `False` otherwise. The loop does this: as long
as the integer n is even, n becomes $n/2$. This amounts to
removing all factors 2 from the integer n. As $n = 56 =$
$2 \times 2 \times 2 \times 7$, this program displays 7.

```
n = 56
while is_even(n) == True:
    n = n // 2
print(n)
```

Input: $n = 56$

n	"is n even" ?	new value of n
56	yes	28
28	yes	14
14	yes	7
7	no	

Display: 7

For the latter example, it is much more natural to start the loop with

```
while is_even(n):
```

Indeed `is_even(n)` is already a value "True" or "False". Therefore we're getting closer to the English
sentence "while n is even…"

Operation "+=". To increment a number you can use these two methods:

```
nb = nb + 1      or      nb += 1
```

The second writing is shorter but makes the program less readable.

Activity 2 (Prime numbers).

Goal: test if an integer is (or not) a prime number.

1. **Smallest divisor.**
   Program a function called `smallest_divisor(n)` that returns, the smallest divisor $d \geqslant 2$ of the
   integer $n \geqslant 2$.
   For example `smallest_divisor(91)` returns 7, because $91 = 7 \times 13$.

   Method.
   - We remind you that d divides n if and only if `n % d` is equal to 0.
   - It is a bad idea to use a loop "for d ranging from 2 to n", since, if for example we know that 7
     is a divisor of 91 it is useless to test if $8, 9, 10\ldots$ are also divisors because we have already
     found a smaller one.
   - A good idea is to use a "while" loop! The principle is: "as long as I haven't got my divisor, I
     should keep looking for". (And so, as soon as I find it, I stop looking.)
   - In practice here are the main lines:
     - Begin with $d = 2$.
     - As long as d does not divide n move on to the next candidate (d becomes $d + 1$).
     - At the end d is the smallest divisor of n (in the worst case $d = n$).

2. **Prime numbers (1).**

Slightly modify your `smallest_divisor(n)` function to write your first prime function `is_prime_1(n)` which returns "True" if n is a prime number and "False" otherwise.
For example `is_prime_1(13)` returns `True`, `is_prime_1(14)` returns `False`.

3. **Fermat numbers.**
Pierre de Fermat ($\sim$1605–1665) thought that all integers of the form $F_n = 2^{(2^n)} + 1$ were prime numbers. Indeed $F_0 = 3$, $F_1 = 5$ and $F_2 = 17$ are prime numbers. If he had known `Python` he would probably have changed his mind! Find the smallest integer F_n which is not prime.
Hint. With `Python` b^c is written `b ** c` and therefore $a^{(b^c)}$ is written `a ** (b ** c)`.

We will improve our function which tests if a number is prime or not, it will allow us to test lots of numbers or very large numbers more quickly.

4. **Prime numbers (2).**
Enhance your previous function to become `is_prime_2(n)`. It should not test all the divisors d from 2 to n, but only up to $\sqrt{n}$.
Explanations.
   - For example, to test if 101 is a prime number, just see if it divisible by $2, 3, \ldots, 10$. It is faster!
   - This improvement is due to the following proposal: if an integer is not prime then it admits a divisor d that verifies $2 \leqslant d \leqslant \sqrt{n}$.
   - Instead of testing if $d \leqslant \sqrt{n}$, it is easier to test if $d^2 \leqslant n$.

5. **Prime numbers (3).**
Improve your function to become `is_prime_3(n)` using the following idea. We test if n is divisible by $d = 2$, but from $d = 3$, we just test the odd divisors (we test d, then $d + 2 \ldots$).
   - For example to test if $n = 419$ is a prime number, we first test if n is divisible by $d = 2$, then $d = 3$ and then $d = 5$, $d = 7 \ldots$
   - This allows you to do about half less tests!
   - Explanations: if an even number d divides n, then we already know that 2 divides n.

6. **Calculation time.**
Compare the calculation times of your different functions `is_prime()` by repeating the call `is_prime(97)`, for example, a million times. See the course below for more information on how to do this.

Lesson 3 (Calculation time).
There are two ways to make programs run faster: a good way and a bad way. The bad way is to buy a more powerful computer. The good method is to find a more efficient algorithm!
With `Python`, it is easy to measure the execution time of a function in order to compare it with the execution time of another. Just use the module `timeit`.
Here is an example: we measure the computation time of two functions that have the same purpose, test if an integer n is divisible by 7.

```python
# First function (not very clever)
def my_function_1(n):
    divis = False
    for k in range(n):
        if k*7 == n:
            divis = True
    return divis
```

```
# Second function (faster)
def my_function_2(n):
    if n % 7 == 0:
        return True
    else:
        return False


# Measurement of execution times
import timeit

print(timeit.timeit("my_function_1(1000)",
    setup="from __main__ import my_function_1",
    number=100000))
print(timeit.timeit("my_function_2(1000)",
    setup="from __main__ import my_function_2",
    number=100000))
```

Results.

The result depends on the computer, but allows the comparison of the execution times of the two functions.

- The measurement for the first function (called $100\,000$ times) returns 5 seconds. The algorithm is not very clever. We're testing if $7 \times 1 = n$, then test $7 \times 2 = n$, $7 \times 3 = n$...

- The measurement for the second function returns 0.01 second! We test if the remainder of n divided by 7 is 0. The second method is therefore 500 times faster than the first.

Explanations.

- The module is named `timeit`.

- The function `timeit.timeit()` returns the execution time in seconds. The function takes the following parameters:

  - a string for the call of the function to be tested (here we ask if 1000 is divisible by 7),

  - an argument `setup="..."` which indicates where to find this function,

  - the number of times you have to repeat the call to the function (here `number=100000`).

- The number of repetitions must be large enough to avoid uncertainties.

Activity 3 (More prime numbers).

Goal: program more "while" loops and study different kinds of prime numbers using your `is_prime()` *function.*

1. Write a `prime_after(n)` function that returns the first prime number p greater than or equal to n.
   For example, the first prime number after $n = 60$ is $p = 61$. What is the first prime number after $n = 100\,000$?

2. Two prime numbers p and $p + 2$ are called *twin prime numbers*. Write a `twin_prime_after(n)` function that returns the first pair p, $p + 2$ of twin prime numbers, with $p \geqslant n$.
   For example, the first pair of twin primes after $n = 60$ is $p = 71$ and $p + 2 = 73$. What is the first pair of twin primes after $n = 100\,000$?

3. An integer p is a *Germain prime number* if p and $2p + 1$ are prime numbers. Write a

`germain_after(n)` function that returns the pair p, $2p + 1$ where p is the first Germain prime number $p \geqslant n$.

For example, the first Germain prime number after $n = 60$ is $p = 83$, with $2p + 1 = 167$. What is the first Germain prime number after $n = 100\,000$?

Strings – Analysis of a text

You're going to do some fun activities by manipulating strings and characters.

Lesson 1 (Characters and strings).

1. A *character* is a unique symbol, some examples of characters include: a lowercase letter "a", a capital letter "B", a special symbol "&", a symbol representing a number "7", a space " " that we will also note "␣".
   To designate a character, it must be put in single quotation marks 'z' or double quotation marks "z".

2. A *string* is a sequence of characters, such as a word "Hello", a sentence 'It is sunny.' or a password "N[w5ms}e!".

3. The type of a character or string is str.

Lesson 2 (Operations on strings).

1. The *concatenation*, i.e. the end-to-end placing of two strings, is done using the operator +. For example "umbr" + "ella" gives the string "umbrella".

2. The empty string "" is useful when you want to initialize a string before adding other characters.

3. The *length* of a string is the number of characters it contains. It is obtained by calling the function len(). For example len("Hello␣World") returns 11 (a space counts as a character).

4. If word is a string then you can retrieve each character by the command word[i]. For example, if word = "plane" then:
   - word[0] is the character "p",
   - word[1] is the character "l",
   - word[2] is the character "a",
   - word[3] is the character "n",
   - word[4] is the character "e".

Letter	p	l	a	n	e
Rank	0	1	2	3	4

Note that there are 5 letters in the word "plane" and that you access it through the ranks starting with 0. The indices are therefore 0, 1, 2, 3, and 4 for the last letter. More generally, if word is a string, characters are obtained by word[i] for i varying from 0 to len(word)-1.

Lesson 3 (Substrings)**.**

You can extract several characters from a string using the syntax `word[i:j]` which returns a string formed by characters ranked i to $j-1$ (beware the character number j is not included).

For example if `word = "wednesday"` then:

- `word[0:4]` returns the `"wedn"` substring formed by the characters of ranks 0, 1, 2 and 3 (but not 4),

- `word[3:6]` returns `"nes"` corresponding to indices 3, 4 and 5.

Letter	w	e	d	n	e	s	d	a	y
Rank	0	1	2	3	4	5	6	7	8

Another example: `word[1:len(word)-1]` returns the word but with its first and last letter cut off.

Activity 1 (Plurals of words)**.**

Goal: write a step by step program that returns the plural of a given word.

1. For a string `word`, for example `"cat"`, the program should display the plural of this word by adding an `"s"`.

2. For a word, for example `"bus"`, it should display the last letter of this string (here `"s"`). Improve your program for the first question, by testing if the last letter is already an `"s"`:
   - if this is the case, then add `"es"` to form the plural (`"bus"` becomes `"buses"`),
   - otherwise you have to add `"s"`.

3. Check if a word ends with a `"y"`. If so, display the plural with `"ies"` (the plural of `"city"` is `"cities"`). (Exceptions are not taken into account.)

4. Wrap all your work from the first three questions in a function called `plural()`. The function displays nothing, but returns the word in its plural form.

---

plural()

Use: `plural(word)`
Input: a word (a string)
Output: the plural of the word

Examples:
- `plural("cat")` returns `"cats"`
- `plural("bus")` returns `"buses"`
- `plural("city")` returns `"cities"`

---

5. Write a function `conjugation()` that conjugates a verb to the present continuous tense.

---

conjugation()

Use: `conjugation(verb)`
Input: a verb (a string, exceptions are not taken into account)
Output: no result but displays conjugation in the present continuous tense

Example: `conjugation("sing")`, prints `"I am singing, you are singing,..."`

Lesson 4 (A little more on strings)**.**

1. A `for ... in ...` loop allows you to browse a string, character by character:

```
for charac in word:
    print(charac)
```

2. You can test if a character belongs to a given list of characters. For example:

```
if charac in ["a", "A", "b", "B", "c", "C"]:
```

allows you to execute instructions if the character `charac` is one of the letters a, A, b, B, c, C.
To avoid some letters, we would use:

```
if charac not in ["X", "Y", "Z"]:
```

Activity 2 (Word games)**.**

Goal: manipulate words in a fun way.

1. **Distance between two words.**
   The Hamming distance between two words of the same length is the number of places where the letters are different.
   For example:

   S**N**AKE   S**T**A**CK**

   The second letter of **SNAKE** is different from the second letter of **STACK**, the fourth and fifth ones are also different. The Hamming distance between **SNAKE** and **STACK** is therefore equal to 3.
   Write a function `hamming_distance()` that calculates the Hamming distance between two words of the same length.

hamming_distance()
Use: `hamming_distance(word1,word2)` Input: two words (strings) Output: the Hamming distance (an integer)
Example: `hamming_distance("SHORT","SKIRT")` returns 2

2. **Upside down.**
   Write a function `upsidedown()` that returns a word backwards: **HELLO** becomes **OLLEH**.

upsidedown()
Use: `upsidedown(word)` Input: a word (a string) Output: the word backwards
Example: `upsidedown("PYTHON")` returns `"NOHTYP"`

3. **Palindrome.**
   Deduce a function that tests whether a word is a palindrome or not. A *palindrome* is a word that can be written from left to right or right to left; for example **RADAR** is a palindrome.

> ### is_palindrome()
>
> Use: `is_palindrome(word)`
> Input: a word (a string)
> Output: "True" if the word is a palindrome, "False" otherwise.
>
> Example: `is_palindrome("KAYAK")` returns `True`

4. **Pig latin.** Pig latin is a made up language, here are the rules according to *Wikipedia*:
   - For words that begin with vowel sounds, one just adds "way" to the end. Examples are:
     - **EAT** becomes **EATWAY**
     - **OMELET** becomes **OMELETWAY**
     - **EGG** becomes **EGGWAY**
   - For words that begin with consonant sounds, all letters before the initial vowel are placed at the end of the word sequence. Then, "ay" is added, as in the following examples:
     - **PIG** becomes **IGPAY**
     - **LATIN** becomes **ATINLAY**
     - **BANANA** becomes **ANANABAY**
     - **STUPID** becomes **UPIDSTAY**
     - **GLOVE** becomes **OVEGLAY**

   Write a `pig_latin()` function that translates into pig latin a word according to this procedure.

> ### pig_latin()
>
> Use: `pig_latin(word)`
> Input: a word (a string)
> Output: the word transformed into pig latin.
>
> Examples:
>   - `pig_latin("DUCK")` returns `"UCKDAY"`
>   - `pig_latin("ALWAYS")` returns `"ALWAYSWAY"`

Activity 3 (DNA)**.**
   *A DNA molecule is made up of about six billion nucleotides. The computer is therefore an essential tool for DNA analysis. In a DNA strand there are only four types of nucleotides that are noted **A**, **C**, **T** or **G**. A DNA sequence is therefore a long word in the form: **TAATTACAGACACCTGAA**...*

1. Write a `presence_of_A()` function that tests if a sequence contains the nucleotide **A**.

> ### presence_of_A()
>
> Use: `presence_of_A(sequence)`
> Input: a DNA sequence (a string whose characters are among A, C, T, G)
> Output: "True" if the sequence contains "A", "False" otherwise.
>
> Example: `presence_of_A("CTTGCT")` returns `False`

2. Write a `position_of_AT()` function that tests if a sequence contains the nucleotide **A** followed by the nucleotide **T** and returns the position of the first occurrence found.

> **position_of_AT()**
>
> Use: `position_of_AT(sequence)`
> Input: a DNA sequence (a string whose characters are among A, C, T, G)
> Output: the position of the first "AT" sequence found (starting at 0); `None` if not found.
>
> Example:
> - `position_of_AT("CTTATGCT")` returns 3
> - `position_of_AT("GATATAT")` returns 1
> - `position_of_AT("GACCGTA")` returns `None`

Hint. `None` is assigned to a variable to indicate the absence of a value.

3. Write a `position()` function that tests if a sequence contains a given code and returns the position of the first occurrence.

> **position()**
>
> Use: `position(code,sequence)`
> Input: a code and a DNA sequence
> Output: the position of the beginning of the code found; `None` if not found
>
> Example: `position("CCG","CTCCGTT")` returns 2

4. A crime has been committed in the castle of Adeno. You recovered two strands of the culprit's DNA, from two distant positions in the DNA sequence. There are four suspects, whose DNA you sequenced. Can you find out who did it?
   First code of the culprit: **CATA**
   Second code of the culprit: **ATGC**

   DNA of Colonel Mustard:
   CCTGGAGGGTGGCCCCACCGGCCGAGACAGCGAGCATATGCAGGAAGCGGCAGGAATAAGGAAAAGCAGC
   Miss Scarlet's DNA:
   CTCCTGATGCTCCTCGCTTGGTGGTTTGAGTGGACCTCCCAGGCCAGTGCCGGGCCCCTCATAGGAGAGG
   Mrs. Peacock's DNA:
   AAGCTCGGGAGGTGGCCAGGCGGCAGGAAGGCGCACCCCCCCAGTACTCCGCGCGCCGGGACAGAATGCC
   Pr. Plum's DNA:
   CTGCAGGAACTTCTTCTGGAAGTACTTCTCCTCCTGCAAATAAAACCTCACCCATGAATGCTCACGCAAG

Lesson 5 (Character encoding).
A character is stored by the computer as an integer. For ASCII/unicode encoding, the capital letter "A" is encoded by 65, the lowercase letter "h" is encoded by 104, and the symbol "#" by 35.
Here is the table of the first characters. Numbers 0 to 32 are not printable characters. However, the number 32 is the space character " ".

33 !	43 +	53 **5**	63 ?	73 **I**	83 **S**	93]	103 **g**	113 **q**	123 **{**
34 "	44 ,	54 **6**	64 @	74 **J**	84 **T**	94 ^	104 **h**	114 **r**	124 **|**
35 #	45 -	55 **7**	65 **A**	75 **K**	85 **U**	95 _	105 **i**	115 **s**	125 **}**
36 $	46 .	56 **8**	66 **B**	76 **L**	86 **V**	96 '	106 **j**	116 **t**	126 ~
37 %	47 /	57 **9**	67 **C**	77 **M**	87 **W**	97 **a**	107 **k**	117 **u**	127 -
38 &	48 **0**	58 :	68 **D**	78 **N**	88 **X**	98 **b**	108 **l**	118 **v**	
39 '	49 **1**	59 ;	69 **E**	79 **O**	89 **Y**	99 **c**	109 **m**	119 **w**	
40 (	50 **2**	60 <	70 **F**	80 **P**	90 **Z**	100 **d**	110 **n**	120 **x**	
41)	51 **3**	61 =	71 **G**	81 **Q**	91 [	101 **e**	111 **o**	121 **y**	
42 *	52 **4**	62 >	72 **H**	82 **R**	92 \	102 **f**	112 **p**	122 **z**	

1. The `chr()` function is a `Python` function that returns the character associated to a code.

> **python: chr()**
>
> Use: `chr(code)`
> Input: a code (an integer)
> Output: a character
>
> Example:
> - `chr(65)` returns `"A"`
> - `chr(123)` returns `"{"`

2. The `ord()` function is a `Python` function corresponding to the reverse operation: it returns the code associated with a character.

> **python: ord()**
>
> Use: `ord(charac)`
> Input: a character (a string of length 1)
> Output: an integer
>
> Example:
> - `ord("A")` returns 65
> - `ord("*")` returns 42

Activity 4 (Upper case/lower case).

Goal: convert a word to upper or lower case.

1. Decode by hand the encrypted hidden message:

   80-121-116-104-111-110   105-115   99-64-64-108

2. Write a loop that displays characters encoded by integers from 33 to 127.
3. What does the command `chr(ord("a")-32)` return? And `chr(ord("B")+32)`?

4. Write an `upper_letter()` function that transforms a lowercase letter into its uppercase letter.

upper_letter()
Use: `upper_letter(charac)` Input: a lowercase character among "a",...,"z" Output: the same letter in upper case Example: `upper_letter("t")` returns `"T"`

5. Write an `uppercase()` function that takes a sentence written in lower case and returns the same sentence written in upper case. Characters that are not lowercase letters remain unchanged.

uppercase()
Use: `uppercase(sentence)` Input: a sentence Output: the same sentence in upper case Example: `uppercase("Hello world!")` returns `"HELLO WORLD!"`

Do the same thing for a `lowercase()` function.

6. Write a function `format_full_name()` that returns the first name and last name formatted according to the style **First_name LAST_NAME**.

format_full_name()
Use: `format_full_name(somebody)` Input: a person's first name and surname (separated by a space) Output: the full name following the format `"Firstname LASTNAME"` Example: • `format_full_name("harry Potter")` returns `"Harry POTTER"` • `format_full_name("LORD Voldemort")` returns `"Lord VOLDEMORT"`

Activity 5.

Goal: determine the language of a text from the analysis of letter frequencies.

1. Write a function called `occurrences_letter()` that counts the number of times the given letter appears in a sentence (in upper case).

occurrences_letter()
Use: `occurrences_letter(letter,sentence)` Input: a letter (a character) and a sentence in capital letters (a string) Output: the number of occurrences of the letter (an integer) Example: `occurrences_letter("E","IS THERE ANYBODY OUT THERE")` returns 4

2. Write a function called `number_letters()` that counts the total number of letters that appear in a sentence (in upper case). Do not count spaces or punctuation.

> ### number_letters()
>
> Use: `number_letters(sentence)`
> Input: a sentence in capital letters (a string)
> Output: the total number of letters from "A" to "Z" (an integer)
>
> Example: `number_letters("IS THERE ANYBODY OUT THERE")` returns 22

3. The *frequency of appearance* of a letter in a text or sentence is the percentage given according to the formula:

$$\text{frequency of appearance of a letter} = \frac{\text{number of occurrences of the letter}}{\text{total number of letters}} \times 100.$$

For example, the sentence **IS THERE ANYBODY OUT THERE** contains 22 letters; the letter **E** appears there 4 times. The frequency of appearance of **E** in this sentence is therefore:

$$f_E = \frac{\text{number of occurrences of E}}{\text{total number of letters}} \times 100 = \frac{4}{22} \times 100 \simeq 16.66$$

The frequency is therefore about 17%.

Write a function called `percentage_letter()` that calculates this frequency of appearance.

> ### percentage_letter()
>
> Use: `percentage_letter(letter,sentence)`
> Input: a letter (a character) and a sentence in capital letters (a string)
> Output: the frequency of appearance of the letter (a number lower than 100)
>
> Example: `percentage_letter("E","IS THERE ANYBODY OUT THERE")` returns `16.66...`

Use this function to properly display the frequency of appearance of all letters in a sentence.

4. Here is the frequency of appearance of letters according to the language used (source: en.wikipedia.org/wiki/Letter_frequency). For example, the most common letter in English is "E" with a frequency of more than 12%. The "W" represents about 2% of letters in English and German, but almost does not appear in French and Spanish. These frequencies also vary according to the text analyzed.

Letter	English	French	German	Spanish
a	8.167%	8.173%	7.094%	12.027%
b	1.492%	0.901%	1.886%	2.215%
c	2.782%	3.345%	2.732%	4.019%
d	4.253%	3.669%	5.076%	5.010%
e	12.702%	16.734%	16.396%	12.614%
f	2.228%	1.066%	1.656%	0.692%
g	2.015%	0.866%	3.009%	1.768%
h	6.094%	0.737%	4.577%	0.703%
i	6.966%	7.579%	6.550%	6.972%
j	0.153%	0.613%	0.268%	0.493%
k	0.772%	0.049%	1.417%	0.011%
l	4.025%	5.456%	3.437%	4.967%
m	2.406%	2.968%	2.534%	3.157%
n	6.749%	7.095%	9.776%	7.023%
o	7.507%	5.819%	3.037%	9.510%
p	1.929%	2.521%	0.670%	2.510%
q	0.095%	1.362%	0.018%	0.877%
r	5.987%	6.693%	7.003%	6.871%
s	6.327%	7.948%	7.577%	7.977%
t	9.056%	7.244%	6.154%	4.632%
u	2.758%	6.429%	5.161%	3.107%
v	0.978%	1.838%	0.846%	1.138%
w	2.360%	0.074%	1.921%	0.017%
x	0.150%	0.427%	0.034%	0.215%
y	1.974%	0.128%	0.039%	1.008%
z	0.074%	0.326%	1.134%	0.467%

In your opinion, which languages were written the following four texts written in (the letters of each word were mixed)?

TMAIER BERACUO RSU NU REBRA PRCEEH EIANTT NE ONS EBC NU GAOFREM EIMATR RERNAD APR L RDUOE LAHECLE UIL TTNI A EUP SREP EC LGNGAEA TE RBONUJO ERMNOUSI DU UBRACEO QUE OVSU EEST LIJO UQE OUVS EM MSZELBE BAEU ASNS MIERNT IS RVETO AGRAME ES PRARPTOE A OEVTR AMGUPLE VUOS SEET EL PNIHXE DSE OSHET ED CSE BIOS A ESC MSOT LE OUBRCEA NE ES ESTN ASP DE IEJO TE OUPR ERRNOTM AS BELEL XOVI IL OREVU NU RGLEA ECB ILESSA EBOMTR AS PIOER EL NRDAER S EN ISIAST TE ITD MNO NOB EUSRMNOI NRPEEAZP QEU UTOT EUTLRFTA IVT XUA SPNEDE DE UECIL UQI L TECEOU TECET NEOCL VATU BNEI UN GMAEORF SNAS TUOED LE EOABURC OHENTXU TE NSCOFU UJRA SMIA UN EPU TRDA UQ NO EN L Y ARRPEIDNT ULSP

WRE TREITE SO TSPA CUDHR AHNCT UND WIND SE STI RED AEVRT MTI ESEIMN IDNK RE ATH END NEABNK WLOH IN EMD AMR ER AFTSS HIN IHSERC RE AHTL HIN MRWA EINM SHNO SAW SRTIBG UD SO NGBA DNEI EIHSGTC ESISTH RAETV UD DEN LERNIOKG NITHC NDE LOENINKGRE TIM OKRN UDN CHWFSEI NEIM NSOH ES STI IEN BIFTRLSEEEN DU BILESE IKDN OMKM EHG MIT MIR RAG ECHNOS EPELSI EIPSL IHC ITM RDI HNCMA BEUTN MBLUNE DINS NA DEM TNDRAS NMIEE UTETMR AHT CAMHN UDNGEL GDAWEN MIEN EATRV MENI VEART DUN OSTHER DU CINTH SAW KNNOEIREGL RIM ILEES PRSTVRCIEH ISE IHGRU BEEILB RIGUH MNEI KNDI NI RDNEUR NATBRLET STAESUL EDR WNID

DSNOACAIF ORP ANU DAEDALRI DNAAEIMTI EQU NNCOSETE EL RSTEOUL SMA AACTFAITNS UQE EL TSVAO OINSRVUE DE US ANIGIICANOM EIORDP TOOD RTEIENS RPO LE ITOABOLRROA ED QIUAMALI USOP A NSSRCAEAD LA TMREAAI NXTADAUEE ROP GOARLS EMESS DE NNAMICLUIAPO Y LOVOIV A

RES LE RHMEOB EOMDNEERPRD DE LOS RSOPMRIE OMTSIPE UEQ CIIDADE LE RTDAAOZ ED LSA CELSAL Y LA NICOIOPS ED LAS UESVNA SSACA Y ES ITRMNEEOD QEU AERFU EL UEQIN IIIRDEGAR LA NAIORTREICP DE AL RRTEIA

IMTRUESMME DNA TEH LNGIIV SI EYAS SIFH REA GJPNUIM DNA HET TTNOCO IS GHIH OH OUYR DDADY SI IRHC DAN ROUY MA SI DOGO GKOILON OS USHH LTLIET BBYA NDOT OUY CYR NEO OF HESET GNSRONIM YUO RE NANGO SIER PU SNIGING NAD OULLY EPADRS YUOR GINSW DAN LYOLU KATE OT HET KSY TUB ITLL TATH MGNIRNO EREHT NATI INTGOHN ACN AHMR OYU TWIH DADYD NDA MYMMA NSTIDGAN YB

Lists I

A list is a way to group elements into a single object. After defining a list, you can retrieve each item of the list one by one, but also add new ones. . .

Lesson 1 (List (1)).

A *list* is a series of elements. This can be a list of integers, for example $[5,-7,12,99]$, or a list of strings, for example `["March","April","May"]` or objects of different types `[3.14,"pi",10e-3,"x",True]`.

- **Construction of a list.** A list is defined by elements between square brackets:
    - `mylist1 = [5,4,3,2,1]` a list of 5 integers,
    - `mylist2 = ["Friday","Saturday","Sunday"]` a list of 3 strings,
    - `mylist3 = []` the empty list (very useful if you intend to complete the list later).

- **Get an item.** To get an item from the list, simply write `mylist[i]` where i is the rank of the desired item.
  Beware! The trap is that you start counting from the rank 0.
  For example after the instruction `mylist = ["A","B","C","D","E","F"]` then
    - `mylist[0]` returns `"A"`
    - `mylist[1]` returns `"B"`
    - `mylist[2]` returns `"C"`
    - `mylist[3]` returns `"D"`
    - `mylist[4]` returns `"E"`
    - `mylist[5]` returns `"F"`

"A"	"B"	"C"	"D"	"E"	"F"

rank :   0   1   2   3   4   5

- **Add an element.** To add an item at the end of a list, just use the command `mylist.append(element)`. For example if `primes = [2,3,5,7]` then `primes.append(11)` adds 11 to the list, if you then execute the instruction `primes.append(13)` then the list `primes` becomes `[2,3,5,7,11,13]`.

- **Example of construction.** Here is how to build the list that contains the first ten squares:

```
list_squares = []              # Start from the empty list
for i in range(10):
    list_squares.append(i**2)  # Add squares one by one
```

At the end `list_squares` is:
$$[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]$$

Lesson 2 (List (2)).

- **Length of a list.** The length of a list is the number of elements it contains. The command `len(mylist)` returns the length. The list $[5,4,3,2,1]$ is 5 elements long, the list `["Friday","Saturday","Sunday"]` has length 3, the empty list `[]` has length 0.

- **Browse a list.** Here is the easiest way to scan a list (and in this case, to display each item):

```
for item in mylist:
    print(item)
```

- **Browse a list (again).** Sometimes you need to know the index of the elements. Here is another way to do it (which here displays the index and the element).

```
n = len(mylist)
for i in range(n):
    print(i,mylist[i])
```

- To get a list from `range()` you have to write:
$$list(range(n))$$

- It's a bad idea to name your list "`list`" because this word is already used by `Python`.

Activity 1 (Simple or compound interests).

Goal: create two lists to compare two types of interests.

1. **Simple interest.** We have an amount of S_0. Each year this investment earns interest based on the initial amount.
   For example, with an initial amount of $S_0 = 1000$ and simple interest of $p = 10\%$. The interest is 100. So after one year, I have a sum of $S_1 = 1100$, after two years $S_2 = 1200\ldots$
   Program a `simple_interest(S0,p,n)` function that returns the list of amounts for the n first years. For example `simple_interest(1000,10,3)` returns $[1000,\ 1100,\ 1200,\ 1300]$.
2. **Compound interest.** An amount of S_0 brings in compound interest. This time the interest is calculated each year on the basis of the sum of the previous year, i.e. according to the formula:
$$I_{n+1} = S_n \times \frac{p}{100}$$
   Program a function `compound_interest(S0,p,n)` which returns the list of amounts of the n first years. For example `compound_interest(1000,10,3)` returns $[1000,\ 1100,\ 1210,\ 1331]$.
3. I have the choice between a simple interest investment of 10% or a compound interest investment of 7%. What is the most advantageous solution depending on the duration of the placement?

Lesson 3 (List (3)).

- **Concatenate two lists.** If you have two lists, you can merge them by the operator "+". For example with `mylist1 = [4,5,6]` and `mylist2 = [7,8,9]`
$$\text{mylist1 + mylist2} \quad \text{is} \quad [4,5,6,7,8,9].$$

- **Add an item at the end.** The operator "+" provides another method to add an item to a list:
$$\text{mylist = mylist + [element]}$$
   For example $[1,2,3,4]$ + $[5]$ is $[1,2,3,4,5]$. Attention! The element to be added must be surrounded by square brackets. It is an alternative method to `mylist.append(element)`.

- **Add an element at the beginning.** With:
$$\texttt{mylist = [element] + mylist}$$
the item is added at the beginning of the list. For example $[5] + [1,2,3,4]$ is $[5,1,2,3,4]$.

- **Slicing lists.** You can extract a whole part of the list at once: `mylist[a:b]` returns the sublist of items with indices a to $b-1$.

"A"	"B"	"C"	"D"	"E"	"F"	"G"

rank :   0   1   2   3   4   5   6

For example if `mylist = ["A","B","C","D","E","F","G"]` then
  - `mylist[1:4]` returns `["B","C","D"]`
  - `mylist[0:2]` returns `["A","B"]`
  - `mylist[4:7]` returns `["E","F","G"]`

Once again, it is important to remember that the index of a list starts at 0 and that slicing `mylist[a:b]` stops at the rank $b-1$.

Activity 2 (Manipulate lists).

  Goal: program small routines that manipulate lists.

1. Program a `rotate(mylist)` function that shifts all the elements of a list by one index (the last element becoming the first). The function returns a new list.
   For example, `rotate([1,2,3,4])` returns the list $[4,1,2,3]$.
2. Program an `inverse(mylist)` function that inverts the order of the elements in a list.
   For example, `inverse([1,2,3,4])` returns the list $[4,3,2,1]$.
3. Program a `delete_rank(mylist,rank)` function that returns a list of all elements, except the one at the given index.
   For example, `delete_rank([8,7,6,5,4],2)` returns the list $[8,7,5,4]$ (item 6 that was at index 2 is deleted).
4. Program a `delete_element(mylist,element)` function returning a list that contains all items except those equal to the specified element.
   For example, `delete_element([8,7,4,6,5,4],4)` returns the list $[8,7,6,5]$ (all items equal to 4 have been deleted).

Lesson 4 (Manipulate lists).
You can now use the `Python` functions which do some of these operations.
- **Invert a list.** Here are three methods:
  - `mylist.reverse()` modifies the list in place (i.e. `mylist` is now reversed, the command returns nothing);
  - `list(reversed(mylist))` returns a new list;
  - `mylist[::-1]` returns a new list.
- **Delete an item.** The command `mylist.remove(element)` deletes the first occurrence found (the list is modified). For example if `mylist = [2,5,3,8,5]` the call `mylist.remove(5)` modifies the list to become $[2,3,8,5]$ (the first 5 has disappeared).
- **Delete an element (again).** The command `del mylist[i]` deletes the element of rank i (the

list is modified).

Activity 3 (Bubble sort).

Goal: order a list from the smallest to the largest element.

The bubble sort is a simple way to order a list, here it will be from the smallest to the largest element. The principle is as follows:

- We go through the list from the beginning. As soon as you encounter two consecutive elements in the wrong order, you exchange them.
- At the end of the first pass, the largest element is at the end and it will not move anymore.
- We restart from the beginning (until the penultimate element), this time the last two elements are well placed.
- We continue this way. There is a total of $n-1$ passages if the list is of length n.
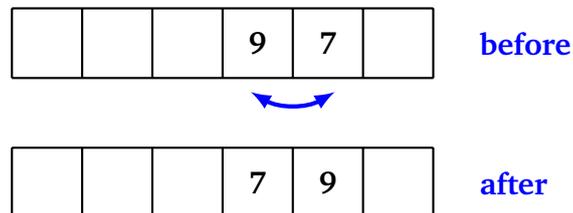
Here is the bubble sort algorithm:

Algorithm.
-
    – Input: a list ℓ of n numbers
    – Output: the ordered list from the smallest to the largest
- For i ranging from $n-1$ to 0:
      For j ranging from 0 to $i-1$:
            If $\ell[j+1] < \ell[j]$ then exchange $\ell[j]$ and $\ell[j+1]$.
- Return the list ℓ.

Program the bubble sort algorithm into a `bubble_sort(mylist)` function that returns the ordered list of elements. For example `bubble_sort([13,11,7,4,6,8,12,6])` returns the list `[4,6,6,7,8,11,12,13]`.

Hints.
- Begin by defining `new_mylist = list(mylist)` and work only with this new list.
- For the index i to run backwards from $n-1$ to 0, you can use the command:

$$\texttt{for i in range(n-1,-1,-1):}$$

Indeed `range(a,b,-1)` corresponds to the decreasing list of integers i satisfying $a \geqslant i > b$ (as usual the right bound is not included).

Lesson 5 (Sorting).
You can now use the `sorted()` function from `Python` which orders lists.

┌───┐
│   python: `sorted()`   │
├───┤
│ Use: `sorted(mylist)`   │
│ Input: a list   │
│ Output: the ordered list of elements   │
│   │
│ Example: `sorted([13,11,7,4,6,8,12,6])`   returns   the   list │
│ `[4,6,6,7,8,11,12,13]`.   │
└───┘

Attention! There is also a `mylist.sort()` method that works a little differently. This command returns nothing, but on the other hand the list `mylist` is now ordered. We are talking about a modification *in place*.

Activity 4 (Arithmetic).

Goal: improve some of the "Arithmetic – While loop – I" chapter functions.

1. **Prime factors.** Program a `prime_factors(n)` function that returns a list of all the prime factors of an integer $n \geqslant 2$. For example, for $n = 12\,936$, its decomposition into prime factors is $n = 2^3 \times 3 \times 7^2 \times 11$, the function returns `[2, 2, 2, 3, 7, 7, 11]`.
   Hints. Consult the "Arithmetic – While loop – I" chapter. The core of the algorithm is as follows:

   > As long as $d \leqslant n$:
   > > If d is a divisor of n, then:
   > > > add d to the list,
   > > > n becomes n/d.
   > > Otherwise increment d by 1.

2. **List of prime numbers.** Write a `list_primes(n)` function that returns the list of all prime numbers less than n. For example `list_primes(100)` returns the list:
   `[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]`
   To do this, you will program an algorithm that is a simple version of the sieve of Eratosthenes:

   > **Algorithm.**
   > - – Input: an integer $n \geqslant 2$.
   >   – Ouput: the list of prime numbers $< n$.
   > - Initialize `mylist` with a list that contains all integers from 2 to $n - 1$.
   > - For d ranging from 2 to $n - 1$:
   >   > For k in `mylist`:
   >   > > If d divides k and $d \neq k$, then remove the element k from `mylist`.
   > - Return `mylist`.

   Hints.
   - Start from `mylist = list(range(2,n))`.
   - Use `mylist.remove(k)`.

   Explanations. Let's see how the algorithm works with $n = 30$.
   - At the beginning the list is

     $[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]$

- We start with $d = 2$, we eliminate all the numbers divisible by 2, unless it is the number 2: so we eliminate 4, 6, 8,..., the list is now $[2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]$.

- We continue with $d = 3$, we eliminate multiples of 3 (except 3), after these operations the list is: $[2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29]$.

- With $d = 4$, we eliminate multiples of 4 (but there are no more).

- With $d = 5$ we eliminate multiples of 5 (here we just eliminate 25), the list becomes $[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]$.

- We continue (here nothing happens anymore).

- At the end, the list is $[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]$.

Lesson 6 (Plot a list).
With the `matplotlib` module it is very easy to visualize the elements of a list of numbers.
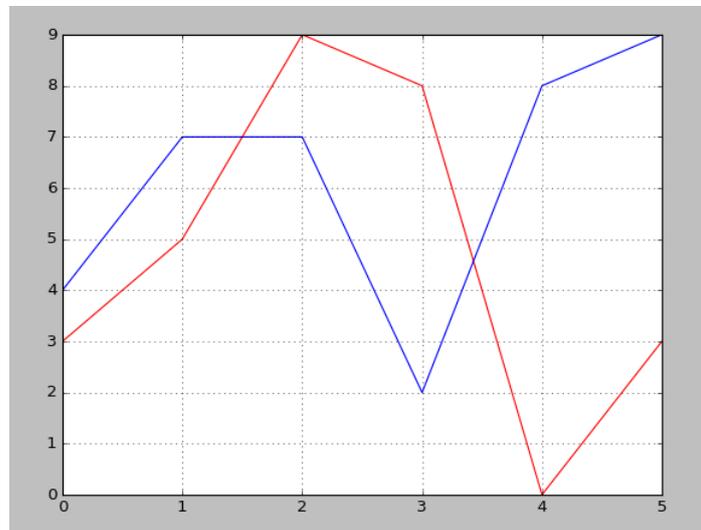

```
import matplotlib.pyplot as plt

mylist1 = [3,5,9,8,0,3]
mylist2 = [4,7,7,2,8,9]

plt.plot(mylist1,color="red")
plt.plot(mylist2,color="blue")
plt.grid()
plt.show()
```

Explanations.

- The module is named `matplotlib.pyplot` and is given the new simpler name of `plt`.

- Attention! The `matplotlib` module is not always installed by default with `Python`.

- `plt.plot(mylist)` traces the points of a list (in the form of (i, ℓ_i)) that are linked by segments.

- `plt.grid()` draws a grid.

- `plt.show()` displays everything.

To display points (x_i, y_i) you must provide the list of x-values then the list of y-values:

```
plt.plot(mylist_x,mylist_y,color="red")
```

Here is an example of a graph obtained by displaying coordinate points of the type (x, y) with $y = x^2$.
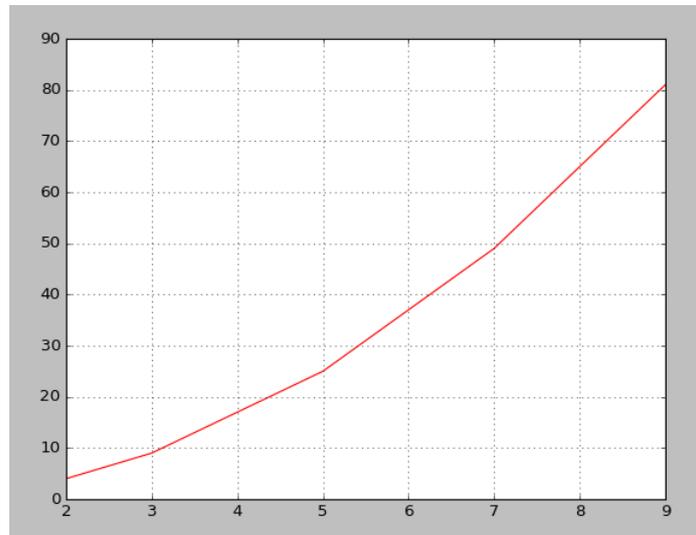

```
import matplotlib.pyplot as plt

mylist_x = [2, 3, 5, 7, 9]
mylist_y = [4, 9, 25, 49, 81]
plt.plot(mylist_x,mylist_y,color="red")
plt.grid()
plt.show()
```

Activity 5 (Ballistics).

Goal: visualize the firing of a cannonball.
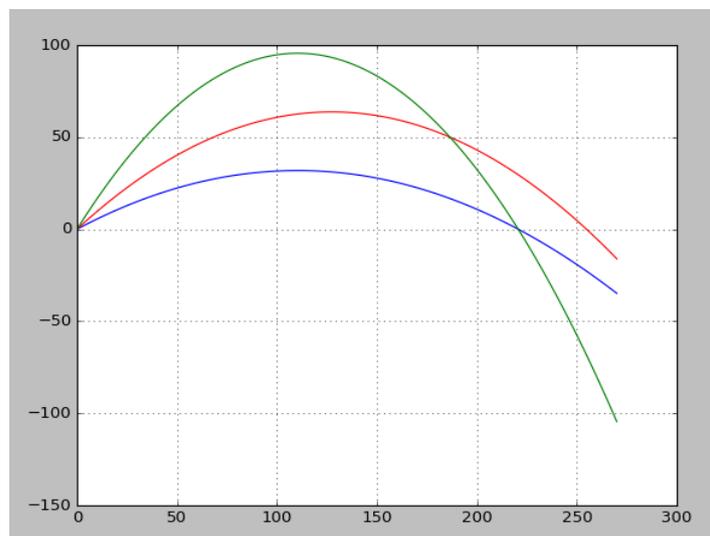
A cannonball has been fired from the point $(0,0)$. The trajectory equation is given by the formula:

$$y(x) = -\frac{1}{2}g\frac{1}{v^2\cos^2(\alpha)}x^2 + \tan(\alpha)x$$

where

- α is the angle of the shot,
- v is the initial speed,
- g is the gravitational constant: we will take $g = 9.81$.
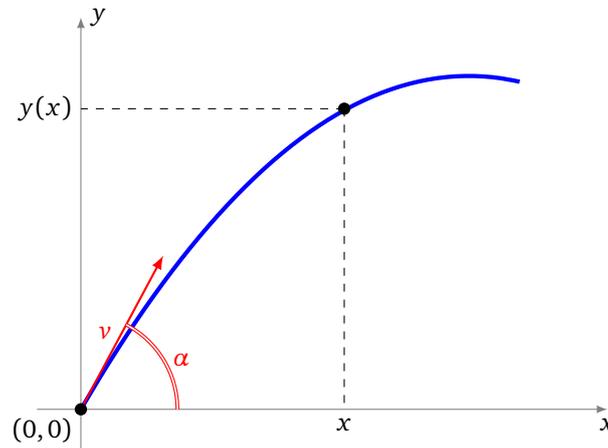
1. Program a `parabolic_shot(x,v,alpha)` function which returns the value $y(x)$ given by the formula.
   Hint. Be careful with the units for the angle α. If for example you choose that the unit for the angle is degrees, then to apply the formula with `Python` you must first convert the angles to radians:

   $$\alpha_{\text{radian}} = \frac{2\pi}{360}\alpha_{\text{degree}}$$

2. Program a `list_trajectory(xmax,n,v,alpha)` function that calculates the list of y-values of the $n+1$ points of the trajectory whose x-values are regularly spaced between 0 and $x_{\max}$.
   Method. For i ranging from 0 to n:
   - calculate $x_i = i \cdot \frac{x_{\max}}{n}$,
   - calculate $y_i = y(x_i)$ using the trajectory formula,
   - add y_i to the list.

3. For $v = 50$, $x_{\max} = 270$ and $n = 100$, display different trajectories according to the values of α. What angle α allows to reach the point $(x,0)$ at ground level as far away from the shooting point as possible?

PART III
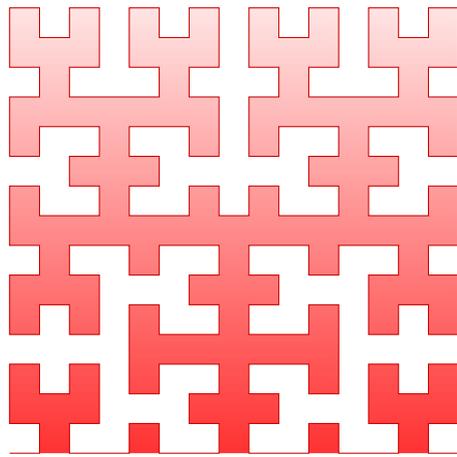
ADVANCED CONCEPTS

Statistics – Data visualization

It's good to know how to calculate the minimum, maximum, average and quartiles of a series. It's even better to visualize them all on the same graph!

Activity 1 (Basic statistics)**.**

> *Goal: calculate the main characteristics of a series of data: minimum, maximum, mean and standard deviation.*

In this activity `mylist` refers to a list of numbers (integer or floating point numbers).

1. Write your own function `mysum(mylist)` which calculates the sum of the elements of a given list. Compare your result with the `sum()` function described below which already exists in `Python`. Especially for an empty list, check that your result is 0.

> **python: sum()**
>
> Use: `sum(mylist)`
> Input: a list of numbers
> Output: a number
>
> Example: `sum([4,8,3])` returns 15

You can now use the function `sum()` in your programs!

2. Write a `mean(mylist)` function that calculates the average of the items in a given list (and returns 0 if the list is empty).

3. Write your own `minimum(mylist)` function that returns the smallest value of the items in a given list. Compare your result with the `Python min()` function described below (which can also calculate the minimum of two numbers).

> **python: min()**
>
> Use: `min(mylist)` or `min(a,b)`
> Input: a list of numbers or two numbers
> Output: a number
>
> Example:
> - `min(12,7)` returns 7
> - `min([10,5,9,12])` returns 5

You can now use the `min()` function and of course also the `max()` function in your programs!

4. The *variance* of a data series $(x_1, x_2, \ldots, x_n)$ is defined as the average of the squares of deviations from the mean. That is to say:

$$v = \frac{1}{n}\big((x_1 - m)^2 + (x_2 - m)^2 + \cdots + (x_n - m)^2\big)$$

where m is the average of $(x_1, x_2, \ldots, x_n)$.

Write a `variance(mylist)` function that calculates the variance of the elements in a list. For example, for the series $(6, 8, 2, 10)$, the average is $m = 6.5$, the variance is

$$v = \frac{1}{4}\big((6 - 6.5)^2 + (8 - 6.5)^2 + (2 - 6.5)^2 + (10 - 6.5)^2\big) = 8.75.$$

5. The *standard deviation* of a series $(x_1, x_2, \ldots, x_n)$ is the square root of the variance:

$$\sigma = \sqrt{v}$$

where v is the variance. Program a `standard_deviation(mylist)` function. With the example above we find $\sigma = \sqrt{v} = \sqrt{8.75} = 2.95\ldots$

6. Here are the average monthly temperatures (in Celsius degrees) in London and Chicago.

```
temp_london = [4.9,5,7.2,9.7,13.1,16.6,18.7,18.2,15.5,11.6,7.7,5.6]
temp_chicago = [-5,-2.7,2.8,9.2,15.2,20.7,23.5,22.6,18.4,12.1,4.8,-1.9]
```

Calculate the average temperature over the year in London and then in Chicago. Calculate the standard deviation of the temperatures in London and then in Chicago. What conclusions do you draw from this?

Lesson 1 (Graphics with tkinter).

To display this:

The code is:

```
# tkinter window
root = Tk()

canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(fill="both", expand=True)

# A rectangle
canvas.create_rectangle(50,50,150,100,width=2)

# A rectangle with thick blue edges
canvas.create_rectangle(200,50,300,150,width=5,outline="blue")

# A rectangle filled with purple
```

```
canvas.create_rectangle(350,100,500,150,fill="purple")

# An ellipse
canvas.create_oval(50,110,180,160,width=4)

# Some text
canvas.create_text(400,75,text="Bla bla bla bla",fill="blue")

# Launch of the window
root.mainloop()
```

Some explanations:

- The `tkinter` module allows us to define variables `root` and `canvas` that determine a graphic window (here width 800 and height 600 pixels). Then describe everything you want to add to the window. And finally the window is displayed by the command `root.mainloop()` (at the very end).

- Attention! The window's graphic marker has its y-axis pointing downwards. The origin $(0,0)$ is the top left corner (see figure below).

- Command to draw a rectangle: `create_rectangle(x1,y1,x2,y2)`; just specify the coordinates (x_1, y_1), (x_2, y_2) of two opposite vertices. The option `width` adjusts the thickness of the line, `outline` defines the color of this line, `fill` defines the filling color.

- An ellipse is traced by the command `create_oval(x1,y1,x2,y2)`, where (x_1, y_1), (x_2, y_2) are the coordinates of two opposite vertices of a rectangle framing the desired ellipse (see figure). A circle is obtained when the corresponding rectangle is a square!

- Text is displayed by the command `canvas.create_text(x,y,text="My text")` specifying the (x, y) coordinates of the point from which you want to display the text.
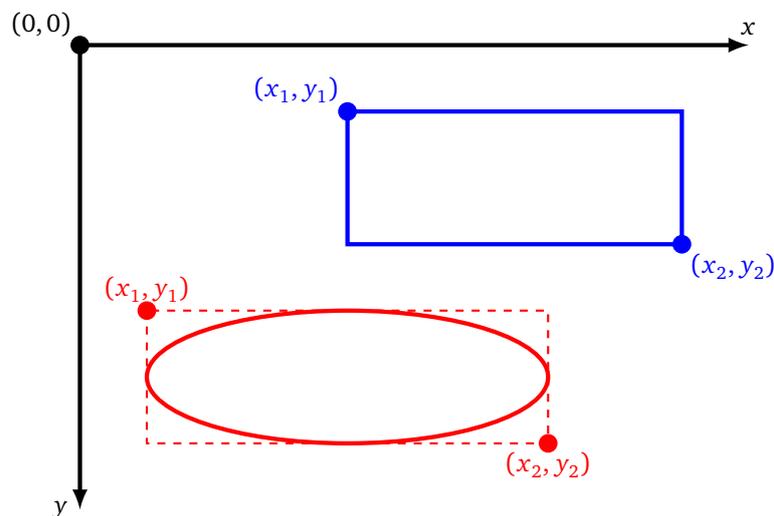
Activity 2 (Graphics).

Goal: visualize data by different types of graphs.
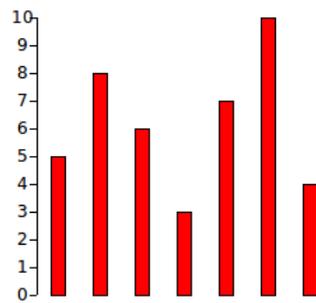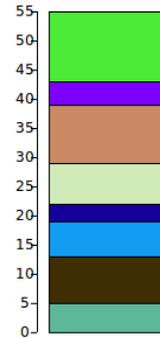
Bar graphics   **Cumulative graph**
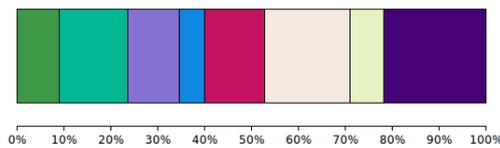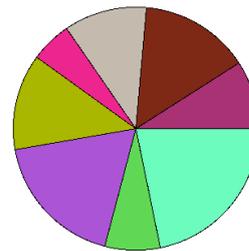
Percentage graphics   **Pie chart**

1. **Bar graphics.** Write a `bar_graphics(mylist)` function that displays the values of a list as vertical bars.

   Hints.

   - First of all, don't worry about drawing the vertical axis of the coordinates with the figures.

   - You can define a variable `scale` that allows you to enlarge your rectangles, so that they have a size adapted to the screen.

   - If you want to test your graph with a random list, here is how to build a random list of 10 integers between 1 and 20:

   ```
   from random import *
   mylist = [randint(1,20) for i in range(10)]
   ```

2. **Cumulative graph.** Write a `cumulative_graphics(mylist)` function that displays the values of a list in the form of rectangles one above the other.

3. **Graphics with percentage.** Write a `percentage_graphics(mylist)` function that displays the values of a list in a horizontal rectangle of fixed size (for example 500 pixels) and is divided into sub-rectangles representing the values.

4. **Pie chart.** Write a `sector_graphics(mylist)` function that displays the values of a list as a pie chart (a fixed size disk divided into sectors representing the values).

   The `tkinter create_arc()` function, which allows you to draw arcs of circles, is not very intuitive. Imagine that we draw a circle, by specifying the coordinates of the corners of a square that surrounds it, then by specifying the starting angle and the angle of the sector (in degrees).

   ```
   canvas.create_arc(x1,y1,x2,y2,start=start_angle,extent=my_angle)
   ```
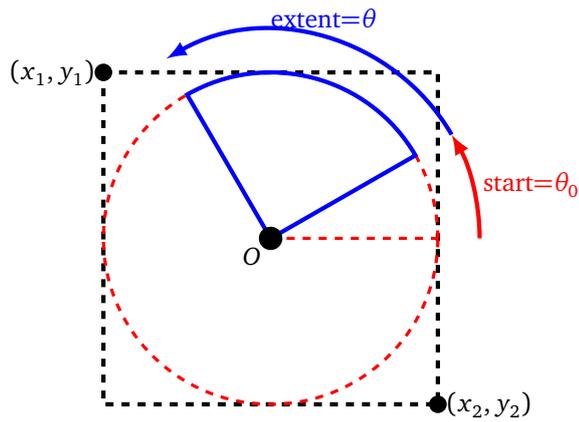
The option `style=PIESLICE` displays a sector instead of an arc.

5. **Bonus.** Gather your work into a program that allows the user to choose the diagram he wants by clicking on buttons, and also the possibility to get a new random series of data. *To create and manage buttons with* `tkinter`*, see the lesson below.*
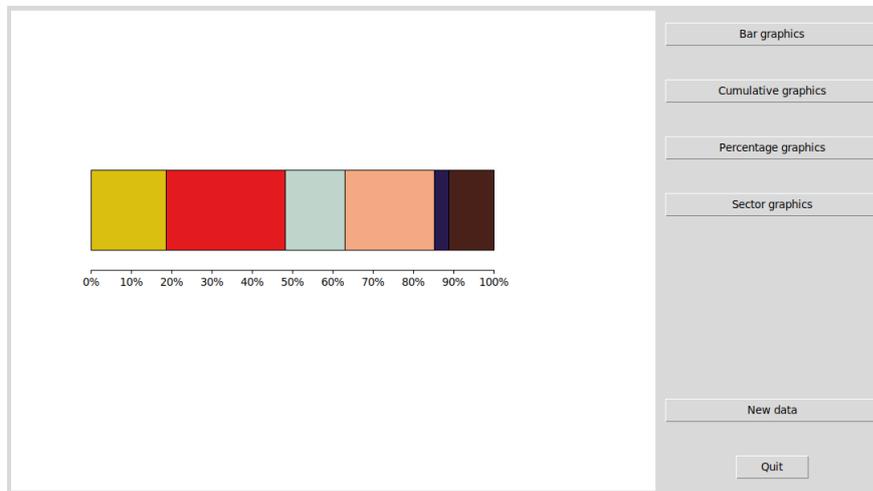
Lesson 2 (Buttons with tkinter).

It is more ergonomic to display windows where actions are performed by clicking on buttons. Here is the window of a small program with two buttons. The first button changes the color of the rectangle, the second button ends the program.

The code is:

```
from tkinter import *
from random import *

root = Tk()
canvas = Canvas(root, width=400, height=200, background="white")
canvas.pack(fill="both", expand=True)

def action_button():
    canvas.delete("all")        # Clear all
    colors = ["red","orange","yellow","green","cyan","blue","purple"]
    col = choice(colors)        # Random color
    canvas.create_rectangle(100,50,300,150,width=5,fill=col)
    return

button_color = Button(root,text="View",width=20,command=action_button)
button_color.pack(pady=10)

button_quit = Button(root,text="Quit", width=20, command=root.quit)
button_quit.pack(side=BOTTOM, pady=10)

root.mainloop()
```

Some explanations:

- A button is created by the command `Button`. The `text` option customizes the text displayed on the button. The button created is added to the window by the method `pack`.

- The most important thing is the action associated with the button! It is the option `command` that receives the name of the function to be executed when the button is clicked. For our example `command=action_button`, associates the click on the button with a change of color.

- Attention! You have to give the name of the function without brackets: `command=my_function` and not `command=my_function()`.

- To associate the button with "Quit" and close the window, the argument is `command=root.quit`.

- The instruction `canvas.delete("all")` deletes all drawings from our graphic window.

Activity 3 (Median and quartiles).

Goal: calculate the median and quartiles of some data.

1. Program a function `median(mylist)` which calculates the median value of the items in a given list. By definition, half of the values are less than or equal to the median, the other half are greater than or equal to the median.
   Background. We note n the length of the list and we assume that the list is ordered (from the smallest to the largest element).
   - **Case n odd.** The median is the value of the list at the index $\frac{n-1}{2}$. Example with `mylist = [12,12,14,15,19]`:
     – the length of the list is $n = 5$ (indices range from 0 to 4),
     – the middle value at index 2,

- the median is the value `mylist[2]`, so it is 14.
  - **Case n even.** The median is the average between the value of the list at index $\frac{n}{2} - 1$ and index $\frac{n}{2}$. Example with `mylist = [10,14,19,20]`:
    - the length of the list is $n = 4$ (indices range from 0 to 3),
    - the middle indices are 1 and 2,
    - the median is the average between `mylist[1]` and `mylist[2]`, so it is $\frac{14+19}{2} = 16.5$.

2. The results of a class are collected in the following form of a number of students per grade:

$$\text{grade\_count} = [0,0,1,2,5,2,3,5,4,1,2]$$

The index i range is from 0 to 10. And the value at index i indicates the number of students who received the grade i. For example here, 1 student got the grade 2, 2 students got the grade 3, 5 students got 4, ... Write a `grades_to_list(grade_count)` function that takes the list of numbers of students for each grade as input and returns the list of all grades. For our example the function must return `[2,3,3,4,4,4,4,4,5,5,6,6,6,7,...]`.

Deduce a function that calculates the median of a class's scores from the numbers of students for each grade.

3. Write a function `quartiles(mylist)` that calculates the quartiles Q_1, Q_2, Q_3 of the items in a given list. The quartiles divide the values into: one quarter below Q_1, one quarter between Q_1 and Q_2, one quarter between Q_2 and Q_3, one quarter above Q_3. For the calculation, we will use the fact that:

- Q_2 is simply the median of the entire list (assumed ordered),
- Q_1 is the median of the sublist formed by the first half of the values,
- Q_3 is the median of the sublist formed by the second half of the values.

For the implementation, it is necessary to consider again whether the length n of the list is even or not. Deduce a function that calculates the quartiles of a class's grades from a list of the numbers of students per grade.

Activity 4 (Box plot).

Goal: draw box plots.

A *box plot* is a diagram that represents the main characteristics of a statistical series: minimum, maximum, median and quartiles. The schematic diagram is as follows:
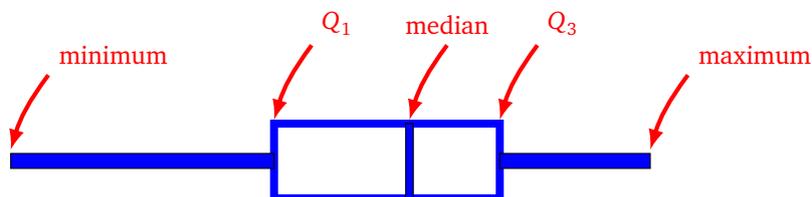
Write a `box_plot(grade_count)` function that draws the box plot of a class's grades from a list of the numbers of students per grade (see previous activity).
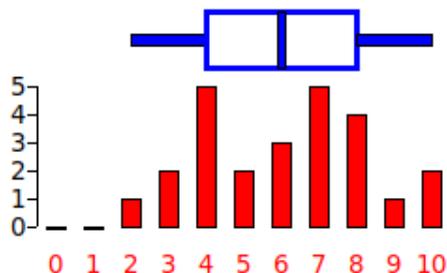
Activity 5 (Moving average).

Goal: calculate moving averages in order to draw "smooth" curves.

1. Simulate the stock market price of the *NISDUQ* index over 365 days. At the beginning, day $j = 0$, the index is equal to 1000. Then the index for a day is determined by adding a random value (positive or negative) to the value of the previous day's index:
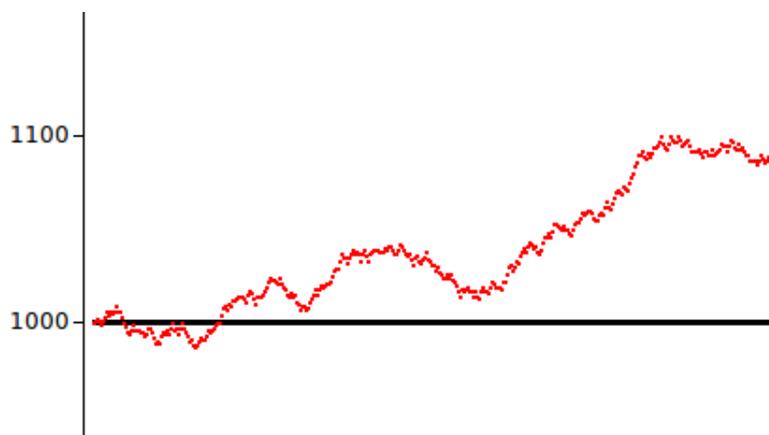
   index of the day j = index of the day $(j - 1)$ + random value.

   For this random value, you can try a formula like:

   ```
   value = randint(-10,12)/3
   ```

   Write an `index_stock_exchange()` function, without parameters, which returns a list of 365 index values using this method.

2. Trace point by point the index curve over a year. (To draw a point, you can display a square with a size of 1 pixel.)

3. *Since the daily index curve is very chaotic, we want to smooth it out in order to make it more readable. For this we can calculate moving averages.*

   The 7-day moving average at the day j, is the average of the last 7 indices. For example: the 7-day moving average for the day $j = 7$ is the average of the day's indices $j = 1, 2, 3, 4, 5, 6, 7$. You can change the duration: for example the 30-day moving average is the average of the last 30 indices. Write a `moving_average(mylist,duration)` function that returns a list of all moving averages in a data set with respect to a fixed time.

4. Trace these data point by point on the same graph: the index curve over a year (shown in red below), the 7-day moving average curve (shown in blue below) and the 30-day moving average curve (shown in brown below). Note that the longer the duration, the more the curve is "smooth". (Of course the 30-day moving average curve only starts from the thirtieth day.)
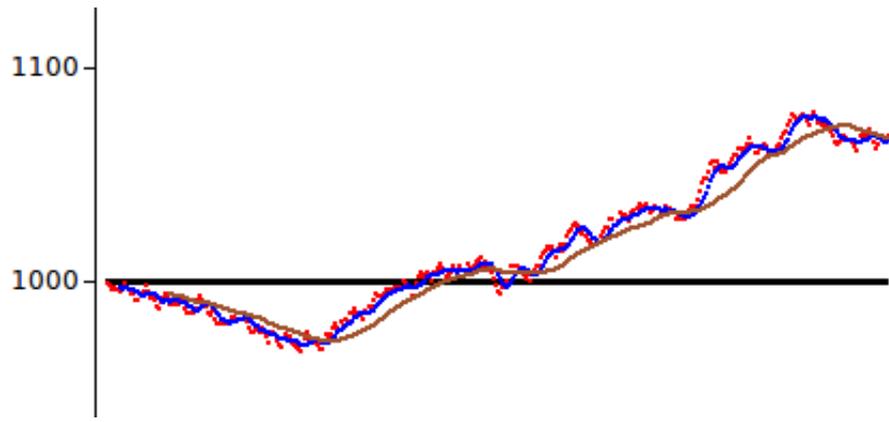
Files

You will learn to read and write data with files.

Lesson 1 (Write to a file).
Writing to a file is almost as easy as displaying a sentence on the screen. On the left is a program that writes two lines to a file called `my_file.txt`; on the right is the resulting file that is displayed in a text editor.

```
fi = open("my_file.txt","w")

fi.write("Hello world!\n")

line = "Hi there.\n"
fi.write(line)

fi.close()
```

```
Hello world!
Hi there.
```

Explanations.
- The command `open` allows you to open a file. The first argument is the name of the file. The second argument here is `"w"` to say that you want to write to the file.
- We do not work with the file name, but with the value returned by the function `open`. Here we have named this object file `fi`. From now on we will work with this variable `fi`.
- We now write in the file almost as we would display a sentence on the screen. The instruction is `fi.write()` where the argument is a string of characters.
- To switch to the next line, you must add the line terminator character `"\n"`.
- It is important to close your file when you are finished writing. The instruction is `fi.close()`.
- The data to be written is composed of strings, so to write a number, you must first transform it using `str(number)`.

Lesson 2 (Read a file).
It is just as easy to read a file. Here is how to do it (left) and the display by `Python` on the screen (right).

```
fi = open("my_file.txt","r")

for line in fi:
    print(line)

fi.close()
```

```
Hello world!

Hi there.
```

Explanations.
- The command `open` is called with the argument `"r"` (for read) this time, it opens the file in reading.
- We work again with an object file here named `fi`.
- A loop goes through the entire file line by line. Here we just ask to display each line.
- We close the file with `fi.close()`.
- The data read is a string, so if you want a number, you must first transform it with `int(string)` (for an integer) or `float(string)` (for a decimal number).

Activity 1 (Read and write a file).

Goal: write a file of grades, then read it to calculate the averages.

1. Generate at random a grades file, named `grades.txt`, which is composed of lines with the structure:

    Firstname Lastname grade1 grade2 grade3

    For example:

    ```
    Tintin Vador 15.0 5.0 19.0
    Bill Croft 15.0 14.5 10.5
    Hermione Skywalker 10.5 7.0 19.5
    Lara Parker 12.5 13.0 14.5
    Hermione Croft 11.5 18.5 9.5
    Robin Vador 8.0 8.0 11.0
    ```

    Hints.
    - Build a list of first names `list_firstnames = ["Alice","Tintin","James",...]`. Then choose a first name at random by the instruction `firstname = choice(list_firstnames)` (you have to import the module `random`).
    - Do the same thing for last names!
    - For a grade, you can choose a random number with the command `randint(a,b)`.
    - **Please note!** Don't forget to convert the numbers into a string to write it to the file: `str(grade)`.

2. Read the `grades.txt` file that you produced. Calculate the average of each person's grades and write the result to an `averages.txt` file where each line is of the form:

    Firstname Lastname average

    For example:

```
Tintin Vador 13.00
Bill Croft 13.33
Hermione Skywalker 12.33
Lara Parker 13.33
Hermione Croft 13.17
Robin Vador 9.00
```

Hints.
- For each line read from the `grades.txt` file, you get the data as a list by using the command `line.split()`.
- **Please note!** The data read is a string. You can convert a string `"12.5"` to the number `12.5` by using the `float(string)` instruction.
- To convert a number to a string with only two decimal places after the dot, you can use the `'{0:.2f}'.format(number)` command.
- Don't forget to close all your files.

Lesson 3 (Files with *csv* format)**.**

The *csv* format (for *comma-separated values*) is a very simple text file format containing data. Each line of the file contains a series of values (numbers or text). On the same line the values are separated by a comma (hence the name of the format, even if other separators are possible).

Example. Here is a file that contains the names, first names, years of birth, height and number of Nobel Prize medals:

```
CURIE,Marie,1867,1.55,2
EINSTEIN,Albert,1879,1.75,1
NOBEL,Alfred,1833,1.70,0
```

Activity 2 (csv format)**.**

Goal: write a data file, with the csv *format, then read it for a graphic display.*

1. Generate a `sales.csv` file of sales figures (randomly drawn) from a sports brand.
   Here is an example:

```
Best sales of the brand 'Pentathlon'

,2015,2016,2017,2018,2019,2020

Mountain bike,560,890,500,550,650,970
Surfboard,550,690,750,640,710,790
Running shoes,850,740,790,1000,680,540
Badminton racket,710,640,620,550,790,880
Volley ball,900,550,790,510,930,800
```

- The data starts from the fifth line.

- The produced file respects the format *csv* and can be readable by *LibreOffice Calc* for example.

	A	B	C	D	E	F	G	
1	Best sales of the brand 'Pentathlon'							
2								
3		2015	2016	2017	2018	2019	2020	
4								
5	Mountain bike	560	890	500	550	650	970	
6	Surfboard	550	690	750	640	710	790	
7	Running shoes	850	740	790	1000	680	540	
8	Badminton racket	710	640	620	550	790	880	
9	Volley ball	900	550	790	510	930	800	
10								

2. Read the `sales.csv` file to display the sales curves.
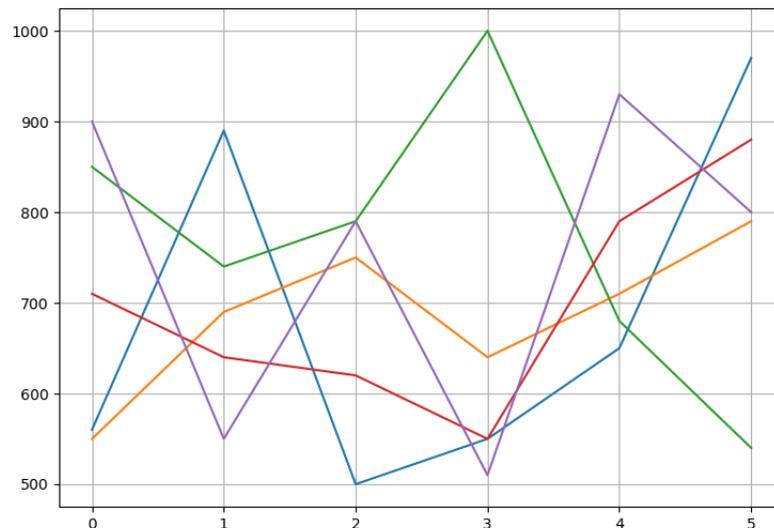
Hints.

- The `matplotlib` package allows you to easily display graphics, it is often called with the instruction:

$$\text{import matplotlib.pyplot as plt}$$

- Here is how to view two data lists `mylist1` and `mylist2`:

```
plt.plot(mylist1)
plt.plot(mylist2)
plt.grid()
plt.show()
```

Lesson 4 (*Bitmap* images)**.**

There is a simple file format, called the *bitmap* format, which describes an image pixel by pixel. This format is available in three variants depending on whether the image is in black and white, grayscale, or color.

Black and white picture, "pbm" format.

The image is described by 1's and 0's.

Here is an example: the `image_bw.pbm` file is on the left (read as a text file) and on the right is its visualization (using an image reader, here very enlarged).

```
P1
4 5
1 1 1 1
1 0 0 0
1 1 1 0
1 0 0 0
1 1 1 1
```


Here is the description of the format:

- First line: the id P1.

- Second line: the number of columns, then the number of rows (here 4 columns and 5 rows).

- Then the color of each pixel line by line: 1 for a black pixel, 0 for a white pixel. Warning: this is contrary to the usual convention!

Grayscale image, "pgm" format.

The image is described by different values for different grayscales. Here is an example: the `image_gray.pbm` file is on the left and on the right is its visualization.

```
P2
4 5
255
  0   0   0   0
192 192 192 192
192 255 128 128
192 255  64  64
192   0   0   0
```


Here is the description of the format:

- First line: the identifier is this time P2.

- Second line: the number of columns, then the number of rows.

- Third line: the maximum value of the grayscale (here 255).

- Then the grayscale of each pixel line by line: this time 0 for a black pixel, the maximum value for a white pixel and intermediate values give intermediate grays.

Image in colors, "ppm" format.

The image is described by three values per pixel: one for red, one for green, and one for blue.

Here is an example: the file `image_col.ppm` on the left and on the right its visualization.

```
P3
3 2
255
255   0   0     0 255 0     0   0 255
  0 128 255   255 128 0   128 255   0
```
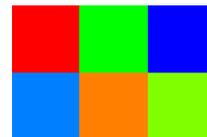

Here is the description of the format:

- First line: the id is now P3.

- Second line: the number of columns, then the number of rows.

- Third line: the maximum value of the color levels (here 255).

- Then each pixel is described by 3 numbers: the level of red, green and blue (RGB system). For example, the first pixel is encoded by $(255, 0, 0)$ so it is a red pixel.

Activity 3 (*Bitmap* images).

Goal: define your own images pixel by pixel.

1. Generate an `image_bw.pbm` file that represents a black and white image (e.g. of size 300×200) according to the following pattern:
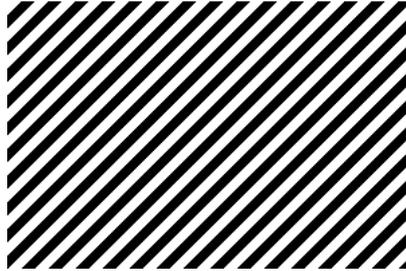
   Hints. If i designates the line number and j the column number (from the top left), then the pixel in position (i, j) is white if $i + j$ is between 0 and 9, between 20 and 29, or between 40 and 49,... This is obtained by the formula:

   ```
   col = (i+j)//10 % 2
   ```

   which returns 0 or 1 as desired.

2. Generate an `image_gray.pgm` file that represents a grayscale image (for example of size 200×200 with 256 grayscale) according to the following pattern:
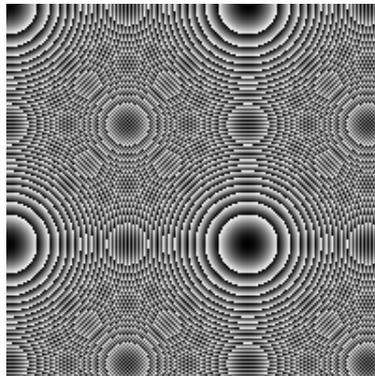
   Hints. This time the formula is:

   ```
   col = (i**2 + j**2) % 256
   ```

   which returns an integer between 0 and 255.

3. Generate an `image_col.ppm` file that represents a color image (e.g. of size 200×200, with 256 red, green and blue levels) according to the following pattern:
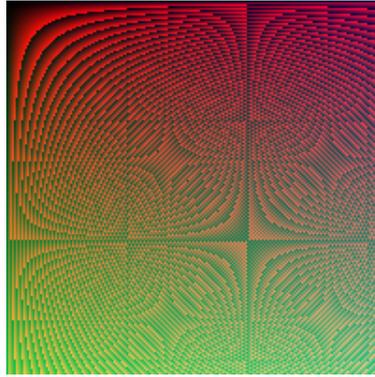
Hints. This time the formulas are:

```
R = (i*j) % 256
  G = i % 256
B = (i + j)//3 % 256
```

which gives the red, green and blue levels of the pixel (i, j).

4. Write an `inverse_black_white(filename)` function that reads a black and white image `.pbm` file and creates a new file in which the white pixels have turned black and vice versa.
   Example: on the left the start image, on the right the finish image.
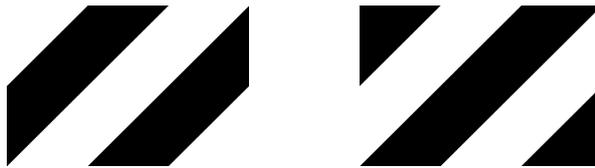
5. Write a `colors_to_gray(filename)` function that reads a color image file in `.ppm` format and creates a new file in `.pgm` format in which the color pixels are transformed into grayscale.
   You can use the formula:

$$g = 0,21 \times R + 0,72 \times G + 0,07 \times B$$

where
   - R, G, B are the red, green and blue levels of the colored pixel,
   - g is the grayscale of the transformed pixel.

   Example: on the left the starting image in color, on the right the arrival image in grayscale.
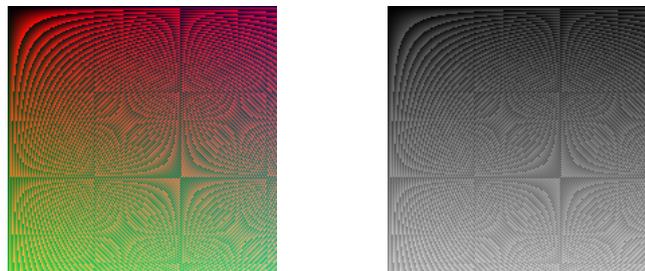
Activity 4 (Distance between two cities).

Goal: read the coordinates of the cities and write the distances between them.

1. **Distance in the plan.**
   Write a program that reads a file containing the coordinates (x, y) of cities, then calculates and writes in another file the distances (on the map) between two cities.

The formula for the distance between two points (x_1, y_1) and (x_2, y_2) of the plane is:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

Example. Here is an example of an input file:

```
Paris 500 800
Lille 500 1000
Nantes 100 700
Marseille 900 0
```

And here is the output file produced by the program:

```
            Paris      Lille     Nantes  Marseille
Paris           0        200        412        894
Lille         200          0        500       1077
Nantes        412        500          0       1063
Marseille     894       1077       1063          0
```

We can read from this file that the distance between Lille and Marseille is 1077 kilometers.
Below is the map of France that provides (very approximate) data for the input file. The origin is at the bottom left, each side of a square represents 100 km. For example, in this diagram, Paris has the coordinates $(500, 800)$.
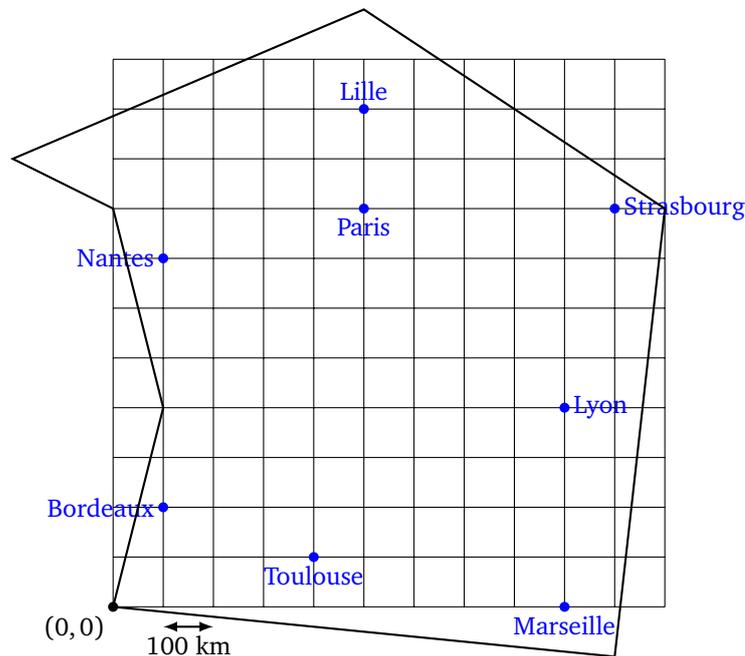
2. **Distance on the sphere.**
   On Earth, the distance between two cities is a distance on a *great circle* along the surface of the sphere and not along a straight line. This is the distance a plane travels to connect two cities.
   Write a program that reads the latitudes and longitudes of cities, then calculates and writes to another file the distances (on the Earth's surface) between two cities.
   Example. Here is an example of an input file:

```
Paris 48.853 2.350
New-York 40.713 -74.006
Vancouver 49.250 -123.119
Lima -12.043 -77.0282
Hong-Kong 22.286 114.158
Addis-Abeba 9.0250 38.747
```

And here is the output file produced by the program:

	Paris	New-York	Vancouver	Lima	Hong-Kong	Addis-Abeba
Paris	0	5837	7924	10253	9629	5573
New-York	5837	0	3905	5874	12959	11207
Vancouver	7924	3905	0	8168	10257	13298
Lima	10253	5874	8168	0	18371	13001
Hong-Kong	9629	12959	10257	18371	0	8135
Addis-Abeba	5573	11207	13298	13001	8135	0

Implementation and explanations.

- The input file contains the latitude (noted as φ) and longitude (noted as λ) in degrees for each city. For example, Paris has a latitude of $\varphi = 48.853$ degrees and a longitude of $\lambda = 2.350$ degrees.

- It will be necessary to use the angles in radians for the formulas. The formula for converting degrees to radians is:

$$\text{angle in radians} = \frac{2\pi}{360} \times \text{angle in degrees}$$

- **Formula for an approximate distance.**
  There is a simple formula that gives a good estimate for the shortest distance between two points on a sphere with a radius of R. First, let:

$$x = (\lambda_2 - \lambda_1) \cdot \cos\left(\frac{\varphi_1 + \varphi_2}{2}\right) \quad \text{and} \quad y = \varphi_2 - \varphi_1$$

  The approximate distance is then

$$\tilde{d} = R\sqrt{x^2 + y^2}$$

  where (φ_1, λ_1) and (φ_2, λ_2) are the latitudes/longitudes of two cities expressed in radians.

- **Exact distance formula.**
  The bravest can use the exact formula to calculate the distance. First, set:

$$a = \left(\sin\left(\frac{\varphi_1 + \varphi_2}{2}\right)\right)^2 + \cos(\varphi_1) \cdot \cos(\varphi_2) \cdot \left(\sin\left(\frac{\lambda_2 - \lambda_1}{2}\right)\right)^2$$

  The exact distance is then:

$$d = 2 \cdot R \cdot \text{atan2}\left(\sqrt{a}, \sqrt{1-a}\right)$$

  where atan2(y, x) is the function "arctangent" which is obtained by the `atan2(y,x)` command from the `math` module.

- For the Earth radius we will take $R = 6371$ km.

Arithmetic – While loop – II

Our study of numbers is further developed with the "while" loop. For this chapter you will need the `is_prime()` *function you wrote in the "Arithmetic – While loop – I" part.*

Activity 1 (Goldbach's conjecture(s)).

Goal: study two Goldbach conjectures. A conjecture is a statement that you think is true but you not know how to prove it.

1. **Goldbach's good guess:** *Every even integer greater than 4 is the sum of two prime numbers.*
   For example $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5$, $10 = 3 + 7$ (but also $10 = 5 + 5$), $12 = 5 + 7$,... For $n = 100$ there are 6 solutions: $100 = 3 + 97 = 11 + 89 = 17 + 83 = 29 + 71 = 41 + 59 = 47 + 53$. No one can prove this conjecture, but you will see that there are good reasons to believe it is true.
   (a) Program a `number_solutions_goldbach(n)` function which for a given even integer n, finds how many decompositions $n = p + q$ there are where p and q are prime numbers and $p \leqslant q$.
   For example, for $n = 8$, there is only one solution $8 = 3 + 5$, but for $n = 10$ there are two solutions $10 = 3 + 7$ and $10 = 5 + 5$.
   Hints.
   - It is therefore necessary to test all p including 2 and $n/2$;
   - set $q = n - p$;
   - we have a solution when $p \leqslant q$ and p and q are both prime numbers.
   (b) Prove with the machine that the Goldbach conjecture is verified for all even integers n between 4 and 10 000.
2. **Goldbach's bad guess:** *Every odd integer n can be written as*
$$n = p + 2k^2$$
   where p is a prime number and k is an integer (possibly zero).
   (a) Program an `is_decomposition_goldbach(n)` function that returns "True" when there is a decomposition of the form $n = p + 2k^2$.
   (b) Show that Goldbach's second guess is wrong! There are two integers smaller than 10 000 that do not have a decomposition of this form. Find them!

Activity 2 (Numbers with 4 or 8 divisors).

Goal: disprove a conjecture by doing a lot of calculations.

Conjecture: *Between 1 and N, there are more integers that have exactly 4 divisors than integers that have exactly 8 divisors.*

You will see that this conjecture looks true for N that are rather small, but you will show that this conjecture is false by finding a large N that contradicts this statement.

1. **Number of divisors.**
   Program a `number_of_divisors(n)` function that returns the number of integers dividing n.
   For example: `number_of_divisors(100)` returns 9 because there are 9 divisors of $n = 100$:

   $$1, 2, 4, 5, 10, 20, 25, 50, 100$$

   Hints.
   - Don't forget 1 and n as divisors.
   - Try to optimize your function because you will use it intensively: for example, there are no divisors strictly larger than $\frac{n}{2}$ (except n).

2. **4 or 8 divisors.**
   Program a `four_and_eight_divisors(Nmin,Nmax)` function that returns two numbers: (1) the number of integers n with $N_{\min} \leqslant n < N_{\max}$ that admit exactly 4 divisors and (2) the number of integers n with $N_{\min} \leqslant n < N_{\max}$ that admit exactly 8 divisors.
   For example `four_and_eight_divisors(1,100)` returns $(32,10)$ because there are 32 integers between 1 and 99 that admit 4 divisors, but only 10 integers that admit 8.

3. **Proof that the conjecture is false.**
   Check that for "small" values of N (up to $N = 10\,000$ for example) there are more integers with 4 divisors than 8. But check that for $N = 300\,000$ this is no longer the case.
   Hints. As there are many calculations, you can split them into slices (the slice of integers $1 \leqslant n < 50\,000$, then $50\,000 \leqslant n < 100\,000$,...) and then add them up. This allows you to split your calculations between several computers.

Activity 3 (121111... is never prime?).

   Goal: study a new false conjecture!

We call U_k the following integer:
$$U_k = 1\,2\,\underbrace{1\,1\,1\ldots1\,1\,1}_{k \text{ occurrences of } 1}$$
formed by the digit 1, then the digit 2, then k times the digit 1.
For example $U_0 = 12$, $U_1 = 121$, $U_2 = 1211$, ...

1. Write a function `one_two_one(k)` that returns the integer U_k.
   Hint. You can notice that starting with $U_0 = 12$, we have the relationship $U_{k+1} = 10 \cdot U_k + 1$. So you can start with `u = 12` and repeat a number of times `u = 10*u + 1`.

2. Check with your machine that $U_0, \ldots, U_{20}$ are not prime numbers.

   You might think it's still the case, but it's not true. The integer U_{136} is a prime number! Unfortunately it is too big to be verified with our algorithms. In the following point we will define what is an almost prime number *to be able to push the calculations further.*

3. Program a function `is_almost_prime(n,r)` that returns "True" if the integer n does not admit any divisor d such that $1 < d \leqslant r$ (we assume $r < n$).
   For example: $n = 143 = 11 \times 13$ and $r = 10$, then `is_almost_prime(n,r)` is "True" because n does not allow any divisor less than or equal to 10. (But of course, n is not a prime number.)
   Hint. Adapt your `is_prime(n)` function!

4. Find all the integers U_k with $0 \leqslant k \leqslant 150$ which are almost prime for $r = 1\,000\,000$ (i.e. they are not divisible by any integer d with $1 < d \leqslant 1\,000\,000$).

Hint. In the list you must find U_{136} (which is a prime number) but also U_{34} which is not prime but whose smallest divisor is $10\,149\,217\,781$.

Activity 4 (Integer square root).

Goal: calculate the integer square root of an integer.

Let $n \geqslant 0$ be an integer. The ***integer square root of*** n is the largest integer $r \geqslant 0$ such as $r^2 \leqslant n$. Another definition is to say that the integer square root of n is $\sqrt{n}$ rounded down to the nearest integer.
Examples:
- $n = 21$, then the integer square root of n is 4 (because $4^2 \leqslant 21$, but $5^2 > 21$). In other words, $\sqrt{21} = 4.58\ldots$, and we round down to the nearest integer, so it is 4.
- $n = 36$, then the integer square root of n is 6 (because $6^2 \leqslant 36$, but $7^2 > 36$). In other words, $\sqrt{36} = 6$ and the integer square root is of course also 6.
1. Write a first function that calculates the integer square root of an integer n, first by calculating $\sqrt{n}$, then rounding down.
   Hints.
   - For this question only, you can use the `math` module of `Python`.
   - In this module `sqrt()` returns the real square root.
   - The `floor()` function of the same module returns the number rounded down to the nearest integer.
2. Write a second function that calculates the integer square root of an integer n, but this time according to the following method:
   - Start with $p = 0$.
   - As long as $p^2 \leqslant n$, increment the value of p by 1.
   Think carefully about what the returned value should be (beware of the offset!).
3. Write a third function that still calculates the integer square root of an integer n with the algorithm described below. This algorithm is called the Babylonian method (or Heron's method or Newton's method).

   Algorithm.
   Input: a positive integer n
   Output: its integer square root
   - Start with $a = 1$ and $b = n$.
   - As long as $|a - b| > 1$, repeat:
       - $a \leftarrow (a + b)//2$,
       - $b \leftarrow n//a$.
   - Return the minimum between a and b: this is the integer square root of n.

   We do not explain how this algorithm works, but it is one of the most effective methods to calculate square roots. The numbers a and b provide, during execution, an increasingly precise interval containing of $\sqrt{n}$.
   Here is a table that details an example calculation for the integer square root of $n = 1664$.

Step	a	b
$i = 0$	$a = 1$	$b = 1664$
$i = 1$	$a = 832$	$b = 2$
$i = 2$	$a = 417$	$b = 3$
$i = 3$	$a = 210$	$b = 7$
$i = 4$	$a = 108$	$b = 15$
$i = 5$	$a = 61$	$b = 27$
$i = 6$	$a = 44$	$b = 37$
$i = 7$	$a = 40$	$b = 41$

In the last step, the difference between a and b is less than or equal to 1, so the integer square root is 40. We can verify that this is correct because: $40^2 = 1600 \leqslant 1664 < 41^2 = 1681$.

Bonus. Compare the execution speeds of the three methods using `timeit()`. See the "Functions" chapter.

Lesson 1 (Exit a loop).

It is not always easy to find the right condition for a "while" loop. `Python` has a command to immediately exit a "while" loop or a "for" loop: this is the `break` command.

Here are some examples that use this `break` command. As it is rarely an elegant way to write your program, alternatives are also presented.

Example.

Here are different codes for a countdown from 10 to 0.

```
# Countdown
n = 10
while True: #Infinite loop
    print(n)
    n = n - 1
    if n < 0:
        break #Immediate stop
```

```
# Better (with a flag)
n = 10
finished = False
while not finished:
    print(n)
    n = n - 1
    if n < 0:
        finished = True
```

```
# Even better
n = 10
while n >= 0:
    print(n)
    n = n - 1
```

Example.

Here are programs that search for the integer square root of 777, i.e. the largest integer i that satisfies $i^2 \leqslant 777$. In the script on the left, the search is limited to integers i between 0 and 99.

```
# Integer square root
n = 777
for i in range(100):
    if i**2 > n:
        break
print(i-1)
```

```
# Better
n = 777
i = 0
while i**2 <= n:
    i = i + 1
print(i-1)
```

Example.

Here are programs that calculate the real square roots of the elements in a list, unless of course the number is negative. The code on the left stops before the end of the list, while the code on the right handles the problem properly.

```
# Square root of the elements
# of a list
mylist = [3,7,0,10,-1,12]
for element in mylist:
    if element < 0:
        break
    print(sqrt(element))
```

```
# Better with try/except
mylist = [3,7,0,10,-1,12]
for element in mylist:
    try:
        print(sqrt(element))
    except:
        print("Warning, I don't know
        how to compute the
        square root of",element)
```

Binary I

The computers transform all data into numbers and manipulate only those numbers. These numbers are stored in the form of lists of 1's and 0's. It's the binary numeral system of numbers. To better understand this binary numeral system, you must first need to understand the decimal numeral system better.

Lesson 1 (Base 10 system).

We usually denote integers with the decimal numeral system (based on 10). For example, $70\,685$ is $7 \times 10\,000 + 0 \times 1\,000 + 6 \times 100 + 8 \times 10 + 5 \times 1$:

7	0	6	8	5
10000	1000	100	10	1
10^4	10^3	10^2	10^1	10^0

(we can see that 5 is the number of units, 8 the number of tens, 6 the number of hundreds...).

It is necessary to understand the powers of 10. We denote $10 \times 10 \times \cdots \times 10$ (with k factors) by 10^k.

d_{p-1}	d_{p-2}	$\cdots$	d_i	$\cdots$	d_2	d_1	d_0
10^{p-1}	10^{p-2}	$\cdots$	10^i	$\cdots$	10^2	10^1	10^0

We calculate the integer corresponding to the digits $[d_{p-1}, d_{p-2}, \ldots, d_2, d_1, d_0]$ by the formula:

$$n = d_{p-1} \times 10^{p-1} + d_{p-2} \times 10^{p-2} + \cdots + d_i \times 10^i + \cdots + d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0$$

Activity 1 (From decimal system to integer).

Goal: from the decimal notation, find the integer.

Write a `decimal_to_integer(list_decimal)` function which takes a list representing the decimal notation and calculates the corresponding integer.

> ### decimal_to_integer()
>
> Use: `decimal_to_integer(list_decimal)`
> Input: a list of numbers between 0 and 9
> Output: the integer whose decimal notation is the list
>
> Example: if the input is `[1,2,3,4]`, the output is `1234`.

Hints. It is necessary to sum up elements of type:

$$d_i \times 10^i \quad \text{for } 0 \leqslant i < p$$

where p is the length of the list and d_i is the digit at index i *counting from the end* (i.e. from right to left). To manage the fact that the index i used for the power of 10 does not correspond to the index in the list, there are two solutions:

- understand that $d_i = \texttt{mylist}[p-1-i]$ where `mylist` is the list of p digits,

- or start by reversing `mylist`.

Lesson 2 (Binary).

- **Power of 2.** We denote 2^k for $2 \times 2 \times \cdots \times 2$ (with k factors). For example, $2^3 = 2 \times 2 \times 2 = 8$.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

- **Binary system: an example.**
  Integers can be represented using a binary numeral system, i.e. a notation where the only numbers are 0 or 1. In binary, the digits are called ***bits***. For example, 1.0.1.1.0.0.1 (pronounce the numbers one by one) is the binary numeral system of the integer 89. How to do you do this calculation? The same way we did using the base 10, but instead using the powers of 2.

1	0	1	1	0	0	1
64	32	16	8	4	2	1
2^6	2^5	2^4	2^3	2^2	2^1	2^0

So the system 1.0.1.1.0.0.1 represents the integer:
$$1 \times 64 + 0 \times 32 + 1 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 = 64 + 16 + 8 + 1 = 89.$$

- **Binary system: formula.**

b_{p-1}	b_{p-2}	$\cdots$	b_i	$\cdots$	b_2	b_1	b_0
2^{p-1}	2^{p-2}	$\cdots$	2^i	$\cdots$	2^2	2^1	2^0

We can calculate the integer corresponding to the bits $[b_{p-1}, b_{p-2}, \ldots, b_2, b_1, b_0]$ as a sum of terms $b_i \times 2^i$, by the formula:
$$n = b_{p-1} \times 2^{p-1} + b_{p-2} \times 2^{p-2} + \cdots + b_i \times 2^i + \cdots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

- `Python` **and the binary.** `Python` accepts the binary decimal system directly as long as we use the prefix "0b". Examples:
  - with `x = 0b11010`, then `print(x)` displays 26,
  - with `y = 0b11111`, then `print(y)` displays 31,
  - and `print(x+y)` displays 57.

Activity 2 (From binary numeral system to integer).

Goal: convert binary values to integers.

1. Calculate integers whose binary numeral system is given below. You can do it by hand or get help from `Python`. For example 1.0.0.1.1 is equal to $2^4 + 2^1 + 2^0 = 19$ as confirmed by the command `0b10011` which returns 19.
   - 1.1, 1.0.1, 1.0.0.1, 1.1.1.1
   - 1.0.0.0.0, 1.0.1.0.1, 1.1.1.1.1
   - 1.0.1.1.0.0, 1.0.0.0.1.1

   • 1.1.1.0.0.1.0.1
2. Write a `binary_to_integer(list_binary)` function which takes in a list representing a binary system value and calculates the corresponding integer.

> ### binary_to_integer()
>
> Use: `binary_to_integer(list_binary)`
> Input: a list of bits, 0 and 1
> Output: the integer whose binary numeral system is the list
>
> Examples:
>   • input `[1,1,0]`, output 6
>   • input `[1,1,0,1,1,1]`, output 55
>   • input `[1,1,0,1,0,0,1,1,0,1,1,1]`, output 3383

Hints. This time it is necessary to sum up elements of type:

$$b_i \times 2^i \quad \text{for } 0 \leqslant i < p$$

where p is the length of the list and b_i is the bit (0 or 1) at index i of the list counting from the end.

3. Here is an algorithm that does the same conversion: it allows the calculation of the integer from the binary notation, but it has the advantage of never directly calculating powers of 2. Program this algorithm into a `binary_to_integer_bis()` function that has the same characteristics as the previous function.

> **Algorithm.**
> Input: `mylist`, a list of 1's and 0's
> Output: the associated binary number
>   • Initialize a variable n to 0.
>   • For each element b from `mylist`:
>   – if b is 0, then make $n \leftarrow 2n$,
>   – if b is 1, then make $n \leftarrow 2n + 1$.
>   • The result is the value of n.

Lesson 3 (Base 10 system (again))**.**
Of course, when you see the number 1234 you can immediately find the list of its digits $[1, 2, 3, 4]$. But how to do it in general from an integer n?

1. We will need to use the Euclidean division by 10:
   • calculate the *remainder* left over after dividing by 10 using `n % 10`, we also call this *n modulo 10*
   • calculate `n // 10` or the *quotient* of this division.
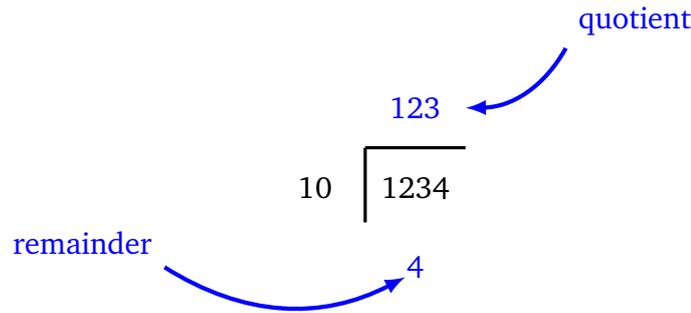
2. The Python commands are simply `n % 10` and `n // 10`.
   Example: `1234 % 10` is equal to 4; `1234 // 10` is equal to 123.

3. • The digit of the units of n is obtained using modulo 10: it is $n\%10$. For example $1234\%10 = 4$.

   • The tens figure is obtained from the quotient of n divided by 10 and then taking modulo 10:
   it is therefore $(n//10)\%10$. Example: $1234//10 = 123$, then we have $123\%10 = 3$; the figure
   of tens of 1234 is indeed 3.

   • For the figure of hundreds, we divide n by 100, then take modulo 10. Example $1234//100 = 12$;
   2 is the figure of the units of 12 and it is the figure of the hundreds of 1234. Note: dividing by
   100 means dividing by 10, then again by 10.

   • For the figure of thousands we calculate the quotient of the division by 1000 then we take the
   figure of the units...

Activity 3 (Find the digits of an integer).

   Goal: decompose an integer into a list of its digits (based on 10*).*

Program the function `integer_to_decimal()` using the following algorithm.

`integer_to_decimal()`
Use: `integer_to_decimal(n)` Input: a positive integer Output: the list of its digits Example: if the input is 1234, the output is `[1,2,3,4]`.

Algorithm.
Input: an integer $n > 0$
Output: the list of its digits
   • Start from an empty list.

   • As long as n is not zero:
        – add $n\%10$ at the beginning of the list,

        – set $n \leftarrow n//10$.

   • The result is the desired list.

Lesson 4 (Binary system with `Python`).

`Python` calculates the binary value of an integer very well using the `bin()` function.

python: bin()
Use: `bin(n)`   Input: an integer   Output: the binary numeral system of n in the form of a string starting with `'0b'`   Example: • `bin(37)` returns `'0b100101'`   • `bin(139)` returns `'0b10001011'`

Lesson 5 (Binary system calculation).

Calculating the binary value of an integer n, is the same as calculating the decimal value except replace the divisions by 10 with a divisions by 2.

So we need:

• $n\%2$: the remainder of the Euclidean division of n by 2 (also called n modulo 2); the remainder is either 0 or 1.

• and $n//2$: the quotient of this division.

Note that the remainder $n\%2$ is either 0 (when n is even) or 1 (when n is odd).

Here is a general method to calculate the binary value of an integer:

• We start from the integer whose binary value we want.

• A series of Euclidean divisions by 2 are performed:
  – for each division, you get a remainder 0 or 1;
  – we get a quotient that we divide again by 2... We stop when this quotient is zero.

• The binary system is read as the sequence of the remainders, but starting from the last one.

Example.

Calculation of the binary value of 14.

- We divide 14 by 2, the quotient is 7, the remainder is 0.
- We divide 7 (the previous quotient) by 2: the new quotient is 3, the new remainder is 1.
- We divide 3 by 2: quotient 1, remainder 1.
- We divide 1 by 2: quotient 0, remainder 1.
- It's over (the last quotient is zero).
- The successive remainders are 0, 1, 1, 1. we read the binary numeral system backwards, it is 1.1.1.0.

The divisions are done from left to right, but the remainders are read from right to left.

```
       7                3                1                0
    ┌─────          ┌─────          ┌─────          ┌─────
 2  │ 14         2  │ 7          2  │ 3          2  │ 1
      (0)              (1)              (1)              (1)
              ←─────────────────────────────────────────
```
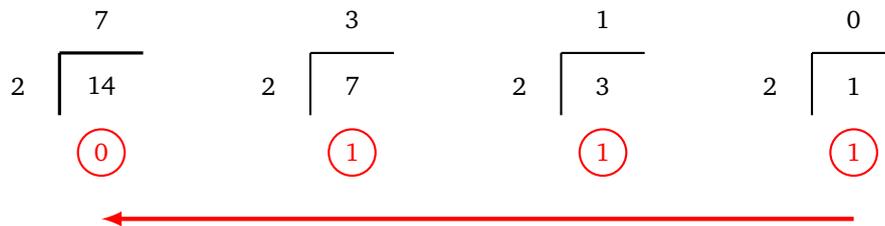
Example.

Binary value of 50.

```
      25               12               6                3                1                0
   ┌─────          ┌─────          ┌─────          ┌─────          ┌─────          ┌─────
 2 │ 50         2  │ 25         2  │ 12         2  │ 6          2  │ 3          2  │ 1
     (0)             (1)             (0)             (0)             (1)             (1)
```
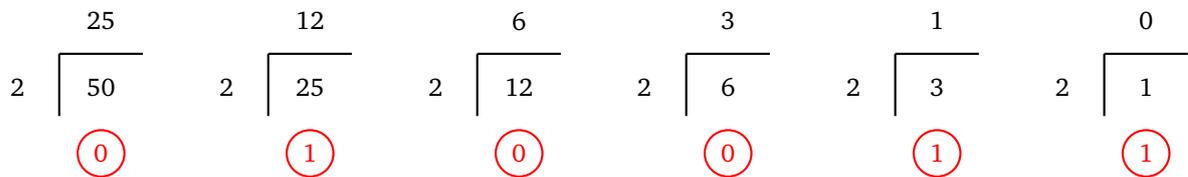
The successive remainders are 0, 1, 0, 0, 1, 1, so the binary notation of 50 is 1.1.0.0.1.0.

Activity 4.

Goal: find the binary value of an integer.

1. Calculate the binary value of the following integers by hand. Check your results using the Python `bin()` function.
   - 13, 18, 29, 31,
   - 44, 48, 63, 64,
   - 100, 135, 239, 1023.
2. Program an `integer_to_binary()` function using the following algorithm.

Algorithm.

Input: an integer $n > 0$

Output: its binary value in the form of a list

- Start from an empty list.

- As long as n is not zero:
    - add $n\%2$ at the beginning of the list,
    - set $n \leftarrow n//2$.

- The result is the desired list.

---

integer_to_binary()

Use: `integer_to_binary(n)`
Input: a positive integer
Output: its binary notation in the form of a list

Example: if the input is 204, the output is `[1,1,0,0,1,1,0,0]`.

Make sure your functions are working properly:

- start with any integer n,

- calculate its binary value,

- calculate the integer associated with this binary notation,

- you should find the starting integer!

Lists II

The lists are so useful that you have to know how to handle them in a simple and efficient way. That's the purpose of this chapter!

Lesson 1 (Manipulate lists efficiently).

- **Slicing lists.**
    - You already know `mylist[a:b]` that returns the sublist of elements from the rank a to the rank $b-1$.
    - `mylist[a:]` returns the list of elements from rank a until the end.
    - `mylist[:b]` returns the list of elements from the beginning to the rank $b-1$.
    - `mylist[-1]` returns the last element, `mylist[-2]` returns the penultimate element, ...
    - **Exercise.**

7	2	4	5	3	10	9	8	3

rank : 0   1   2   3   4   5   6   7   8

    With `mylist = [7,2,4,5,3,10,9,8,3]`, what do the following instructions return?
    - `mylist[3:5]`
    - `mylist[4:]`
    - `mylist[:6]`
    - `mylist[-1]`

- **Find the rank of an element.**
    - `mylist.index(element)` returns the first position at which the item was found. Example: with `mylist = [12, 30, 5, 9, 5, 21]`, `mylist.index(5)` returns 2.
    - If you just want to know if an item belongs to a list, then the statement:

                        element in mylist

    returns True or False. Example: with `mylist = [12, 30, 5, 9, 5, 21]`, "9 in mylist" is true, while "8 in mylist" is false.

- **List comprehension.**
    A set can be defined by listing all its elements, for example $E = \{0, 2, 4, 6, 8, 10\}$. Another way is to say that the elements of the set must verify a certain property. For example, the same set E can be defined by:
    $$E = \{x \in \mathbb{N} \mid x \leqslant 10 \text{ and } x \text{ is even}\}.$$
    With `Python` there is a way to define lists this way. It is an extremely powerful and efficient syntax. Let's look at some examples:
    - Let's start from a list, for example `mylist = [1,2,3,4,5,6,7,6,5,4,3,2,1]`.

– The command `mylist_doubles = [ 2*x for x in mylist ]` returns a list that contains the double of each item of the `mylist` list. So this is the list $[2,4,6,8,...]$.

– The command `mylist_squares = [ x**2 for x in mylist ]` returns the list of squares of the items in the initial list. So this is the list $[1,4,9,16,...]$.

– The command `mylist_partial = [x for x in mylist if x > 2]` extracts the list composed only of elements greater than 2. So this is the list $[3,4,5,6,7,6,5,4,3]$.

- **List of lists.**
  A list can contain other lists, for example:

  `mylist = [ ["Harry", "Hermione", "Ron"], [101,103] ]`

  contains two lists. We will be interested in lists that contain lists of integers, which we will call *arrays*. For example:

  `array = [ [2,14,5], [3,5,7], [15,19,4], [8,6,5] ]`

  Then `array[i]` returns the sublist of index i, while `array[i][j]` returns the integer located at the index j in the sublist of index i. For example:

  – `array[0]` returns the list $[2,14,5]$,

  – `array[1]` returns the list $[3,5,7]$,

  – `array[0][0]` returns the integer 2,

  – `array[0][1]` returns the integer 14,

  – `array[2][1]` returns the integer 19.

Activity 1 (Lists comprehension).

Goal: practice list comprehension. In this activity the lists are lists of integers.

1. Program a `multiplication(mylist,k)` function that multiplies each item in the list by k. For example, `multiplication([1,2,3,4,5],2)` returns $[2,4,6,8,10]$.
2. Program a `power(mylist,k)` function that raises each element of the list to the power k. For example, `power([1,2,3,4,5],3)` returns $[1,8,27,64,125]$.
3. Program an `addition(mylist1,mylist2)` function that adds together the elements of two lists of the same length. For example, `addition([1,2,3],[4,5,6])` returns $[5,7,9]$.
   Hint. This is an example of a task where list comprehension is not used!
4. Program a `non_zero(mylist)` function that returns a list of all non-zero elements. For example, `non_zero([1,0,2,3,0,4,5,0])` returns $[1,2,3,4,5]$.
5. Program an `even(mylist)` function that returns a list of all even elements. For example, `even([1,0,2,3,0,4,5,0])` returns $[0,2,0,4,0]$.

Activity 2 (Reach a fixed amount).

Goal: try to reach the total of 100 *in a list of numbers.*

We consider a list of n integers between 1 and 99 (included). For example, this list of 25 integers:

$$[16,2,85,27,9,45,98,73,12,26,46,25,26,49,18,99,10,86,7,42]$$

which was obtained at random by the command:

`mylist_20 = [randint(1,99) for i in range(20)]`

We are looking for different ways to find numbers from the list whose sum is exactly 100.

1. Program a `sum_twoinarow_100(mylist)` function that tests if there are two consecutive elements in the list whose sum is exactly 100. The function returns `True` or `False` (but it can also

display numbers and their position for verification). For the example given, this function returns `False`.

2. Program a `sum_two_100(mylist)` function that tests if there are two items in the list, located at different positions, whose sum is equal to 100. For the example given, this function returns `True` and can display the integers 2 and 98 (at ranks 1 and 6 of the list).

3. Program a `sum_seq_100(mylist)` function that tests if there are consecutive elements in the list whose sum is equal to 100. For the example given, this function returns `True` and can display the sequence of integers 25, 26, 49 (at ranks 11, 12 and 13).

4. *(Optional.)* The larger the size of the list, the more likely it is to have values in the list whose sum is 100. For each of the three previous situations, determine the size n of the list such that the probability of obtaining a sum of 100 is greater than $1/2$.
   Hints. For each case, you can get an estimate of this integer n, by writing a `proba(n,N)` function that performs a large number N of random draws of lists having n items ($N = 10\,000$ for example). The probability is approximated by the number of successful cases (where the function returns `True`) divided by the total number of cases (here N).

Activity 3 (Arrays).

   Goal: working with lists of lists.

In this activity we work with arrays of size $n \times n$ containing integers. The object `array` is therefore a list of n lists, each having n elements.
For example ($n = 3$):

$$\text{array = [[1,2,3], [4,5,6], [7,8,9]]}$$

represents the array:

$$
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{array}
$$

1. Write a `sum_diagonal(array)` function that calculates the sum of the elements located on the main diagonal of an array. The main diagonal of the example given is 1, 5, 9, so the sum is 15.

2. Write a `sum_antidiagonal(array)` function that calculates the sum of the elements located on the other diagonal. The anti-diagonal of the example given is composed of 3, 5, 7, the sum is still 15.

3. Write a `sum_all(array)` function that calculates the total sum of all elements. In this example the total sum is 45.

4. Write a `print_array(array)` function that displays an array properly on the screen. You can use the command:

$$\text{print('\{:>3d\}'.format(array[i][j]), end="")}$$

   Explanations.
   - The command `print(string,end="")` allows you to display a string of characters without going to the next line.
   - The command `'{:>3d}'.format(k)` displays the integer k on three characters (even if there is only one digit to display).

Activity 4 (Magic Squares).

   Goal: build magic squares as big as you want! You must first have done the previous activity.

A *magic square* is a square array of size $n \times n$ that contains all the integers from 1 to n^2 and satisfies that: the sum of each row, the sum of each column, the sum of the main diagonal and the sum of the anti-diagonal all have the same value.

Here is an example of a magic square with a size of 3×3 and one of size 4×4.
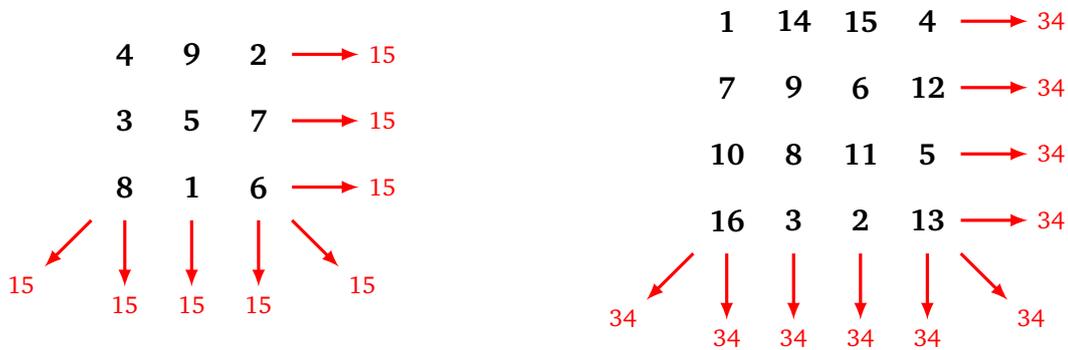
For a magic square of size $n \times n$, the value of the sum is:

$$S_n = \frac{n(n^2 + 1)}{2}.$$

1. **Examples.** Define an array for each of the 3×3 and 4×4 examples above and display them on the screen (use the previous activity).

2. **To be or not to be.** Define an `is_magic_square(square)` function that tests whether a given array is (or isn't) a magic square (use the previous activity for diagonals).

3. **Random squares.** *(Optional.)* Randomly generate squares containing integers from 1 to n^2 using a `random_square(n)` function. Experimentally verify that it is rare to obtain a magic square in this way!

   Hints. For a list `mylist`, the command `shuffle(mylist)` (from the `random` module) randomly mixes the list (the list is modified in place).

   The purpose of the remaining questions is to create large magic squares.

4. **Addition.** Define an `addition_square(square,k)` function which adds an integer k to all the elements of the array. With the example of the 3×3 square, the command `addition_square(square,-1)` subtracts 1 from all the elements and returns an array that would look like this:

   $$\begin{array}{ccc} 3 & 8 & 1 \\ 2 & 4 & 6 \\ 7 & 0 & 5 \end{array}$$

   Hints. To define a new square, start by filling it with 0's:

   ```
   new_square = [[0 for j in range(n)] for i in range(n)]
   ```

   then fill it with the correct values using commands of the type:

   ```
   new_square[i][j] = ...
   ```

5. **Multiplication.** Define a `multiplication_square(square,k)` function which multiplies all the elements of the array by k. With the example of the 3×3 square, the `multiplication_square(square,2)` command multiplies all the elements by 2 and thus returns an array that would be displayed as follows:

   $$\begin{array}{ccc} 8 & 18 & 4 \\ 6 & 10 & 14 \\ 16 & 2 & 12 \end{array}$$

6. **Homothety.** Define a `homothety_square(square,k)` function which enlarges the array by a factor of k as shown in the examples below. Here is an example of the 3×3 square with a homothety ratio of $k = 3$.

4	9	2
3	5	7
8	1	6

$\longrightarrow$

4	4	4	9	9	9	2	2	2
4	4	4	9	9	9	2	2	2
4	4	4	9	9	9	2	2	2
3	3	3	5	5	5	7	7	7
3	3	3	5	5	5	7	7	7
3	3	3	5	5	5	7	7	7
8	8	8	1	1	1	6	6	6
8	8	8	1	1	1	6	6	6
8	8	8	1	1	1	6	6	6

Here is an example of a 4×4 square with a homothety ratio of $k = 2$.

1	14	15	4
7	9	6	12
10	8	11	5
16	3	2	13

$\longrightarrow$

1	1	14	14	15	15	4	4
1	1	14	14	15	15	4	4
7	7	9	9	6	6	12	12
7	7	9	9	6	6	12	12
10	10	8	8	11	11	5	5
10	10	8	8	11	11	5	5
16	16	3	3	2	2	13	13
16	16	3	3	2	2	13	13

7. **Block addition.** Define a `block_addition_square(big_square,small_square)` function that adds a small array of size $n \times n$ to each block of the larger $nm \times nm$ sized array as shown in the example below with $n = 2$ and $m = 3$ (hence $nm = 6$). The small 2×2 square on the left is added to the large square in the center to give the result on the right. For this addition the large square is divided into 9 blocks, there is a total of 36 additions.

1	2
3	4

4	4	9	9	2	2
4	4	9	9	2	2
3	3	5	5	7	7
3	3	5	5	7	7
8	8	1	1	6	6
8	8	1	1	6	6

$\longrightarrow$

5	6	10	11	3	4
7	8	12	13	5	6
4	5	6	7	8	9
6	7	8	9	10	11
9	10	2	3	7	8
11	12	4	5	9	10

8. **Products of magic squares.** Define a `product_squares(square1,square2)` function which from two magic squares, calculates a larger magic square called the product of the two squares. The algorithm is as follows:

Algorithm.

-
  - Inputs: a magic square C_1 of size $n \times n$ and a magic square C_2 of size $m \times m$.
  - Output: a magic square C of size $(nm) \times (nm)$.
- Create square C_{3a} by subtracting 1 from all elements of C_2. (Use the `addition_square(square2,-1)` command.)
- Define square C_{3b} as the homothety of square C_{3a} of ratio n. (Use the `homothety(square3a,n)` command.)
- Define square C_{3c} by multiplying all the terms of square C_{3b} by n^2. (Use the `multiplication_square(square3b,n**2)` command.)
- Define square C_{3d} by adding square C_1 to square C_{3c} block by block. (Use the `block_addition_square(square3c,square1)` command.)
- Return the square C_{3d}.

- Implement this algorithm.
- Test it on examples, checking that the square obtained is indeed a magic square.
- Build a magic square of size 36×36.
- Also confirm that the order of the product is important: $C_1 \times C_2$ is not the same square as $C_2 \times C_1$.

Binary II

We continue our exploration of the world of 1's and 0's.

Activity 1 (Palindromes)**.**

Goal: find palindromes in the binary and decimal numeral systems.

In English a palindrome is a word (or a sentence) that can be read in both directions, for example "**RADAR**" or "**A MAN, A PLAN, A CANAL: PANAMA**". In this activity, a *palindrome* will be a list, which has the same elements when browsing from left to right or right to left.
Examples:

- [1,0,1,0,1] is a palindrome (in the binary numeral system),

- [2,9,4,4,9,2] is a palindrome (in the decimal numeral system).

1. Program an is_palindrome(mylist) function that tests if a list is a palindrome or not.
   Hints. You can compare the items at index i and $p-1-i$ or use list(reversed(mylist)).
2. We are looking for integers n such that their binary value is a palindrome. For example, the binary notation of $n = 27$ is the palindrome [1,1,0,1,1]. This is the tenth integer n that has this property.
   What is the thousandth integer $n \geqslant 0$ whose binary value is a palindrome?
3. What is the thousandth integer $n \geqslant 0$ whose decimal value is a palindrome?
   For example, the digits of $n = 909$ in the decimal system, form the palindrome [9,0,9]. This is the hundredth integer n that has this property.
4. An integer n is a *bi-palindrome* if its binary notation *and* its decimal notation are palindromes. For example $n = 585$ has a decimal value which is a palindrome and so is its binary value [1,0,0,1,0,0,1,0,0,1]. This is the tenth integer n that has this property.
   What is the twentieth integer $n \geqslant 0$ that is a bi-palindrome?

Lesson 1 (Logical operations)**.**
We consider that 0 represents "False" and 1 represents "True".

- With the logical operation "OR", the result is true as soon as at least one of the two terms is true. Case by case:
    - 0 OR 0 = 0
    - 0 OR 1 = 1
    - 1 OR 0 = 1
    - 1 OR 1 = 1

- With the logical operation "AND", the result is true only when both terms are true. Case by case:

≫
≫
≫
≫
≫
≫
≫
≫
≫
≫
≫
≫
≫
≫
≫
≫
≫
≫
≫
≫
≫

- – 0 AND 0 = 0
- – 0 AND 1 = 0
- – 1 AND 0 = 0
- – 1 AND 1 = 1
- The logical operation "NOT", inverts true and false values:
  - – NOT 0 = 1
  - – NOT 1 = 0
- For numbers in binary notation, these operations range from bit to bit, i.e. digit by digit (starting with the digits on the right) as one would with adding (without carrying).
  For example:

$$
\begin{array}{rl}
 & 1.0.1.1.0 \\
\text{AND} & 1.1.0.1.0 \\
\hline
 & 1.0.0.1.0
\end{array}
\qquad
\begin{array}{rl}
 & 1.0.0.1.0 \\
\text{OR} & 0.0.1.1.0 \\
\hline
 & 1.0.1.1.0
\end{array}
$$

If the two systems do not have the same number of bits, we add non-significant 0's on the left (as shown in the example of 1.0.0.1.0 OR 1.1.0 in the figure on the right).

Activity 2 (Logical operations).

Goal: program the main logical operations.

1. (a) Program a `NOT()` function which corresponds to the negation for a given list. For example, `NOT([1,1,0,1])` returns `[0,0,1,0]`.
   (b) Program an `OReq()` function which corresponds to "OR" with two lists of equal length. For example, given `mylist1 = [1,0,1,0,1,0,1]` and `mylist2 = [1,0,0,1,0,0,1]`, the function returns `[1,0,1,1,1,0,1]`.
   (c) Do the same for `ANDeq()` for two lists having the same length.
2. Write a `zero_padding(mylist,p)` function that adds zeros at the beginning of the list to get a list of length p. Example: if `mylist = [1,0,1,1]` and p = 8, then the function returns `[0,0,0,0,1,0,1,1]`.
3. Write two functions `OR()` and `AND()` which correspond to the logical operations, but with two lists that do not necessarily have the same length.
   Example:
   - `mylist1 = [1,1,1,0,1]` and `mylist2 = [1,1,0]`,
   - it should be considered that `mylist2` is equivalent to the list `mylist2bis = [0,0,1,1,0]` of the same length as `mylist1`,
   - so `OR(mylist1,mylist2)` returns `[1,1,1,1,1]`,
   - then `AND(mylist1,mylist2)` returns `[0,0,1,0,0]` (or `[1,0,0]` depending on your choice).

   Hints. You can use the content of your functions `OReq` and `ANDeq`, or you can first add zeros to the shortest list.

Activity 3 (De Morgan's laws)**.**

Goal: generate all possible lists of 0's and 1's to check a proposition.

1. **First method: use binary notation.**
   We want to generate all possible lists of 0's and 1's of a given size p. Here's how to do it:
     - We define an integer n, that ranges from 0 to $2^p - 1$.
     - For each of these integers n, we calculate its binary value (in the form of a list).
     - We add (if necessary) 0 at the beginning of the list, in order to get a list of length p.
   Program this method into a function.
   Example: $n = 36$ in binary notation is $[1,0,0,1,0,0]$. If you want a list of $p = 8$ bits, you should add two 0: $[0,0,1,0,0,1,0,0]$.

2. **Second method (optional): a recursive algorithm.**
   We again want to generate all the possible lists of 0's and 1's of a given size. We adopt the following procedure: if we know how to find all the lists of size $p-1$, then to obtain all the lists of size p, we just have to add one 0 at the beginning of each list of size $p-1$, then to start again by adding a 1 at the beginning of each list of size $p-1$.
   For example, there are 4 lists of length 2: $[0,\ 0], [0,\ 1], [1,\ 0], [1,\ 1]$. I get the 8 lists of length 3:
     - 4 lists by adding 0 at the front: $[0,\ 0,\ 0], [0,\ 0,\ 1], [0,\ 1,\ 0], [0,\ 1,\ 1]$,
     - 4 lists by adding 1 at the front: $[1,\ 0,\ 0], [1,\ 0,\ 1], [1,\ 1,\ 0], [1,\ 1,\ 1]$.
   This gives the following algorithm, which is known as a recursive algorithm (because the function calls itself).

   > **Algorithm.**
   > Use: `every_binary_number(p)`
   > Input: an integer p > 0
   > Output: the list of all possible lists of 0's and 1's of length p
   >   - If p = 1 return the list `[ [0], [1] ]`.
   >   - If p ⩾ 2, then:
   >   – get all lists of size p-1 by the call `every_binary_number(p-1)`
   >   – for each item in this list, build two new items:
   >   – on the one hand add 0 at the beginning of this element;
   >   – on the other hand add 1 at the beginning of this element;
   >   – then add these two items to the list of lists of size p.
   >   - Return the list of all the lists with a size p.

3. **De Morgan's laws.**
   De Morgan's laws state that for booleans (true/false) or bits (1/0), we always have these properties:
   $$\text{NOT}(b_1 \text{ OR } b_2) = \text{NOT}(b_1) \text{ AND NOT}(b_2) \quad \text{NOT}(b_1 \text{ AND } b_2) = \text{NOT}(b_1) \text{ OR NOT}(b_2)$$
   Experimentally check that these equations are still true for any list ℓ_1 and ℓ_2 of exactly 8 bits.

PART IV
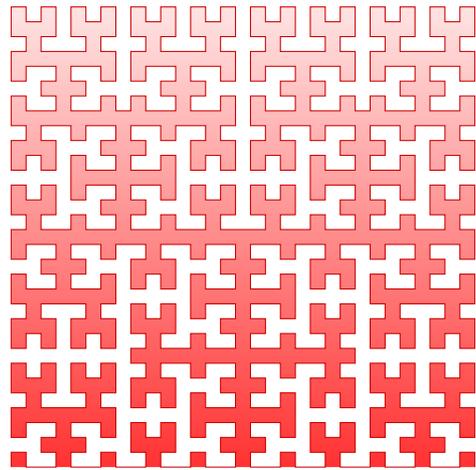
PROJECTS

Probabilities – Parrondo's paradox

You will program two simple games. When you play these games, you are more likely to lose than to win. However, when you play both games at the same time, you have a better chance of winning than losing! It's a paradoxical situation.

Lesson 1 (Random – Winning – Expected value).

A player plays a game of chance: he throws a coin; depending on the result he wins or loses money. We cannot predict how much the player will win or lose for sure, but we will introduce a value that estimates how much the player can expect to win on average if he plays many times.

- At the start the player's total winnings are zero: $g = 0$.

- Each time he draws, he throws a coin. If he wins, he gets one dollar, if he loses he owes one dollar.

- In the games we study, the coin is not balanced (it is a little rigged). The player does not have as many chances to win as to lose.

- Draws are repeated a number of times N. After N draws, the player's winnings (which can be positive or negative) are totaled.

- The *expected value*, is the amount that the player can expect to win with each throw. The expected value is estimated by averaging the earnings of a large number of draws. In other words, we have the formula:
$$\text{expected value} \simeq \frac{\text{winnings afters } N \text{ throws}}{N} \quad \text{with large } N.$$
For our games, the expected value will be a real number between -1 and $+1$.

- Examples:
    - If the coin is well balanced (50 chances out of 100 to win), then for a large number of N draws, the player will win about as many times as he loses; his winnings will be close to 0 and so the expected value will be close to $\frac{0}{N} = 0$. On average he earns 0 dollar per draw.
    - If the coin is rigged and the player wins all the time, then after N draws, he has pocketed N dollars. The expected value is therefore $\frac{N}{N} = 1$.
    - If the coin is rigged so that the player loses all the time, then after N draws, his win is $-N$ dollars. The expected value is therefore $\frac{-N}{N} = -1$.
    - An expected value of -0.5 means that on average the player loses 0.5 dollar per draw. This is possible with an unbalanced coin that wins in only one out of four cases. (Check the calculation!) If the player plays 1000 times, we can estimate that he will lose 500 dollars ($-0.5 \times 1000 = -500$).

Activity 1 (Game A: first losing game).

Goal: first, model a simple game, where, on average, the player loses.

Game A. In this first game, we throw a slightly unbalanced coin: the player wins one dollar in 49 cases out of 100; he loses one dollar in 51 cases out of 100.
1. **One draw.** Write a `throw_game_A()` function that does not depend on any argument and that models a draw of the game A. For this purpose:
    * Pick a random number $0 \leqslant x < 1$ using the `random()` function of the `random` module.
    * Returns $+1$ if x is smaller than 0.49; and -1 otherwise.
2. **Gain.** Write a `gain_game_A(N)` function that models N draws from game A and returns the total winnings of these draws. Of course, the result depends on the actual runs, it can vary from one time to another.
3. **Expected value.** Write an `expected_value_game_A(N)` function that returns an estimate of the expected value of game A according to the formula:
$$\text{expected value} \simeq \frac{\text{winnings afters } N \text{ throws}}{N} \quad \text{with large } N.$$
4. **Conclusion.**
    (a) Estimate the expected value by making at least one million draws.
    (b) What does it mean that the expected value is negative?
    (c) Deduct from the expected value how much I can expect to win (or lose) by playing 1000 times in game A.

Activity 2 (Game B: second losing game).

Goal: model a second game that is a little more complicated where on average the player still loses.

Game B. The second game is a little more complicated. At the beginning, the player starts with a zero win: $g = 0$. Then, depending on this gain, he plays one of the following two subgames:
* **Subgame B1.** If the gain g is a multiple of 3, then he throws a very disadvantageous coin: the player wins a dollar in only 9 cases out of 100 (so he loses a dollar in 91 cases out of 100).
* **Subgame B2.** If the gain g is not a multiple of 3, then he throws an advantageous coin: the player wins a dollar in 74 cases out of 100 (so he loses a dollar in 26 cases out of 100).
1. **One draw.** Write a `throw_game_B(g)` function that depends on the amount already acquired and models a draw from game B. You can use the `g%3 == 0` test to find out if g is a multiple of 3.
2. **Gain.** Write a `gain_game_B(N)` function that models N draws playing game B (starting from a zero initial gain) and returns the total gain of these draws.
3. **Expected value.** Write an `expected_value_game_B(N)` function that returns an estimate of the expected value of game B.
4. **Conclusion.**
    (a) Estimate the expected value by making at least one million draws.
    (b) How much can I expect to win or lose by playing 1000 times in game B?

Activity 3 (Games A and B: a winning game!).
Goal: invent a new game where at each round you play either game A or game B; strangely enough, on average, this game makes you win! That's Parrondo's paradox.

Game AB. In this third game, each round you play either game A or game B (the choice is made at

random). At the beginning the player starts with a zero win: $g = 0$. At each step, he chooses at random (50% of luck each):

- to play game A once,

- or to play game B once; more precisely with subgame B1 or subgame B2 depending on the win already acquired g.

1. **One draw.** Write a `throw_game_AB(g)` function that depends on the amount already acquired and models a draw of the AB game.

2. **Gain.** Write a `gain_game_AB(N)` function that models N draws from the AB game (starting from a zero initial gain) and returns the total gain of these draws.

3. **Expected value.** Write an `expected_value_game_AB(N)` function that returns an estimate of the expected value of the AB game.

4. **Conclusion.**
   (a) Estimate the expected value by performing at least one million game turns.
   (b) What can we say this time about the sign of the expected value?
   (c) How much can I expect to win or lose by playing the AB game 1000 times? Surprising, isn't it?

Reference: "Parrondo's paradox", Hélène Davaux, La gazette des mathématiciens, July 2017.

Find and replace

Finding and replacing are two very frequent tasks. Knowing how to use them and how they work will help you be more effective.

Activity 1 (Find).

Goal: learn different ways to search with `Python`.

1. **The operator "in".**
   The easiest way to know if a substring is present in a string is to use the operator "`in`". For example, the expression:

   $$\texttt{"NOT" in "TO BE OR NOT TO BE"}$$

   is equal to "True" because the substring **NOT** is present in the sentence.
   Use the `in` operator to define a `find_in(string,substring)` function that returns "True" or "False", depending on whether the substring is (or is not) present in the string.

2. **The method** `find()`.
   `string.find(substring)` returns the position at which the substring was found.
   Test this on the previous example. What does the function return if the substring is not found?

3. **The method** `index()`.
   The `index()` method has the same utility. `string.index(substring)` returns the position at which the substring was found.
   Test this on the previous example. What does the function return if the substring is not found?

4. **Your function** `find()`.
   Write your own `myfind(string,substring)` function which returns the starting position of the substring if it is found (and returns `None` if it is not).
   You are not allowed to use the already mentioned `Python` functions, you only have the right to test if two characters are equal.

Activity 2 (Replace).

Goal: replace portions of text with others.

1. The `replace()` method is used in the form:

   $$\texttt{string.replace(substring,new\_substring)}$$

   Each time the sequence `substring` is found in `string`, it is replaced by `new_substring`.
   Transform the sentence **TO BE OR NOT TO BE** into **TO BE AND NOT TO BE**, then into **TO HAVE AND NOT TO HAVE**.

2. Write your own `myreplace()` function which you will call in the following form:

   $$\texttt{myreplace(string,substring,new\_substring)}$$

and which only replaces the first occurrence of the `substring` found. For example, `myreplace("ABBA","B","XY")` returns `"AXYBA"`.

Hint. You can use your `myfind()` function from the previous activity to find the starting position of the sequence to replace.

3. Improve your function to build a `replace_all()` function which now replaces all occurrences encountered.

Lesson 1 (Regular expressions *regex*).

The *regular expressions* allow you to search for substrings with greater freedom: for example, you can allow a wildcard character or several possible choices for a character. There are many other possibilities, but we are only studying these two.

1. We allow ourselves a joker letter symbolized by a point "**.**". For example, if we look for the expression "**P.R**" then:

   - **PORK**, **EMPIRE**, **PURE**, **REPORT** contain this group (for example the point plays the role of **O** in the word **PORK**),

   - but the words **CAR**, **POOR**, **RAP**, **PRICE** do not.

2. We are still looking for groups of letters, we now allow ourselves several options. For example "**[CT]**" means "**C** or **T**". Thus the letter group "**[CT]O**" corresponds to the letter group "**CO**" or "**TO**". This group is therefore contained in **TOTEM**, **COST**, **ACTOR** but not in **BLOCK** nor in **VOTE**. Similarly "**[ABC]**" would mean "**A** or **B** or **C**".

We will use regular expressions through a command:

$$\text{python\_regex\_find(string,exp)}$$

whose function is defined below.

```
from re import *

def python_regex_find(string,exp):
    pattern = search(exp,string)
    if pattern:
        return pattern.group(), pattern.start(), pattern.end()
    else:
        return None
```

Program and test it. It returns: (1) the found substring, (2) the start position and (3) the end position.

```
python: re.search() - python_regex_find()
```

Use: `search(exp,string)`
          or `python_regex_find(string,exp)`
Input: a string `string` and a regular expression `exp`
Output: the result of the search (the substring found, its start position, its end position)

Example with `string = "TO BE OR NOT TO BE"`
- with `exp = "N.T"`, then `python_regex_find(string, exp)` returns (`'NOT'`, 9, 12).
- with `exp = "B..O"`, the function returns (`'BE O'`, 3, 7) (the space counts as a character).
- with `exp = "[NM]O"`, the function returns (`'NO'`, 9, 11).
- with `exp = "[BC]..O[RS]"`, the function returns (`'BE OR'`, 3, 8).

Activity 3 (Regular expressions *regex*)**.**

　　Goal: program a search for simple regular expressions.

1. Program your `regex_find_wildcard(string,exp)` function which is looking for a substring that can contain one or more wildcards ".". The function must return: (1) the found substring, (2) the start position and (3) the end position (as in the `python_regex_find()` function above).
2. Program your `regex_find_choice(string,exp)` function which is looking for a substring that can contain one or more choices contained in tags "[]". The function must return again: (1) the found substring, (2) the start position and (3) the end position.
   Hint. You can start by writing an `all_choices(exp)` function that generates all possibilities from exp. For example, if `exp = "[AB]X[CD]Y"` then `all_choices(exp)` returns the list formed of: `"AXCY"`, `"BXCY"`, `"AXDY"` and `"BXDY"`.

Lesson 2 (Replace 0 and 1 and start again!)**.**
We consider a "sentence" composed of only two possible characters **0** and **1**. In this sentence we will search for a pattern (a substring) and replace it with another one.

Example.

Apply the transformation $01 \rightarrow 10$ to the sentence 10110.

We read the sentence from left to right, we find the first pattern 01 starting at the second character, we replace it with 10:

$$1(01)10 \quad \longmapsto \quad 1(10)10$$

We can start again from the beginning of the sentence obtained, with the same transformation $01 \rightarrow 10$:

$$11(01)0 \quad \longmapsto \quad 11(10)0$$

The pattern 01 no longer appears in the sentence 11100 so the transformation $01 \rightarrow 10$ now leaves this sentence unchanged.

Let's summarize: here is the effect of the iterated transformation $01 \rightarrow 10$ in the sentence 10110:

$$10110 \quad \longmapsto \quad 11010 \quad \longmapsto \quad 11100$$

Example.

Apply the transformation $001 \rightarrow 1100$ to the sentence 0011.

A first time:

$$(001)1 \quad \longmapsto \quad (1100)1$$

A second time:

$$11(001) \quad \longmapsto \quad 11(1100)$$

And then the transformation no longer modifies the sentence.

Example.

Let's see one last example with the transformation $01 \rightarrow 1100$ for the starting sentence 0001:

$$0001 \quad \longmapsto \quad 001100 \quad \longmapsto \quad 01100100 \quad \longmapsto \quad 1100100100 \quad \longmapsto \quad \cdots$$

We can iterate the transformation, to obtain longer and longer sentences.

Activity 4 (Replacement iterations)**.**

> *Goal: study some transformations and their iterations.*

Here we will consider only transformations of the type $0^a 1^b \rightarrow 1^c 0^d$, i.e. a pattern with first 0's then 1's is replaced by a pattern with first 1's then 0's.

1. **One iteration.**

   Using your `myreplace()` function from the first activity, check the above examples. Make sure you replace only one pattern at each step (the leftmost one).

   Example: the transformation $01 \rightarrow 10$ applied to the sentence 101, is calculated by `myreplace("101","01","10")` and returns `"110"`.

2. **Multiple iterations.**

   Program an `iterations(sentence,pattern,new_pattern)` function that, from a sentence, iterates the transformation. Once the sentence is stabilized, the function returns the number of iterations performed and the resulting sentence. If the number of iterations does not seem to stop (for example when it exceeds 1000) then returns `None`.

   Example. For the transformation $0011 \rightarrow 1100$ and the sentence 00001101, the sentences obtained

are:

$$000011011 \xmapsto{\ 1\ } 001100011 \xmapsto{\ 2\ } 110000011 \xmapsto{\ 3\ } 110001100 \xmapsto{\ 4\ } 110110000 \longmapsto \quad \cdots$$

For this example, the call to the `iterations()` function returns 4 (the number of transformations before stabilization) and `"110110000"` (the stabilized sentence).

3. **The most iterations possible.**

   Program a `max_iterations(p,pattern,new_pattern)` function which, among all the sentences of length p, is looking for one of those that takes the longest to stabilize. This function returns:

   - the maximum number of iterations,

   - a sentence that achieves this maximum,

   - and the corresponding stabilized sentence.

   Example: for the transformation $01 \rightarrow 100$, among all the sentences of length $p = 4$, the maximum number of possible iterations is 7. An example of such a sentence is 0111, which will stabilize (after 7 iterations) in 11100000000. So the `max_iteration(4,"01","100")` command returns:

   $$7, \ '0111', \ '11100000000'$$

   Hint. To generate all sentences with a length of p formed of 0 and 1, you can consult the "Binary II" chapter (activity 3).

4. **Categories of transformations.**

   - **Linear transformation.** Experimentally check that the transformation $0011 \rightarrow 110$ is *linear*, i.e. for all sentences with a length of p, there will be at most about p iterations before stabilization. For example, for $p = 10$, what is the maximum number of iterations?

   - **Quadratic transformation.** Experimentally check that the transformation $01 \rightarrow 10$ is *quadratic*, i.e. for all sentences with a length of p, there will be at most about p^2 iterations before stabilization. For example, for $p = 10$, what is the maximum number of iterations?

   - **Exponential transformation.** Experimentally check that the transformation $01 \rightarrow 110$ is *exponential*, i.e. for all sentences with a length of p, there will be a finite number of iterations, but that this number can be very large (much larger than p^2) before stabilization. For example, for $p = 10$, what is the maximum number of iterations?

   - **Transformation without end.** Experimentally verify that for the transformation $01 \rightarrow 1100$, there are sentences that will never stabilize.

Polish calculator – Stacks

You're going to program your own calculator! For that you will discover a new notation for formulas and also discover what a "stack" is in computer science.

Lesson 1 (Stack)**.**

A *stack* is a sequence of data with three basic operations:

- *push*: you add an element to the top of the stack,
- *pop*: the value of the element at the top of the stack is read and this element is removed from the stack,
- and finally, we can test if the stack is empty.

A stack of integers   **A stack of strings**   **An empty stack**
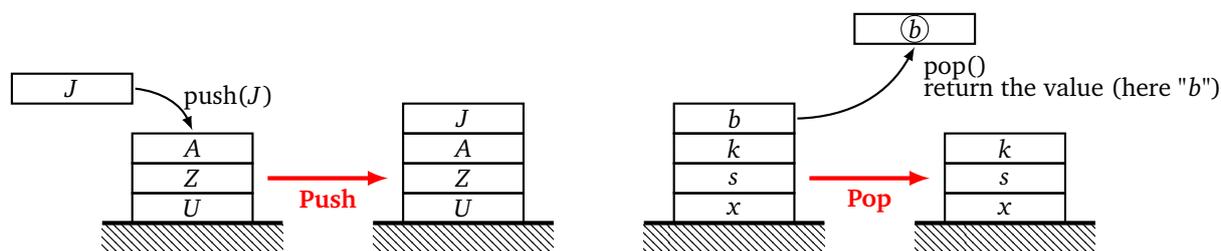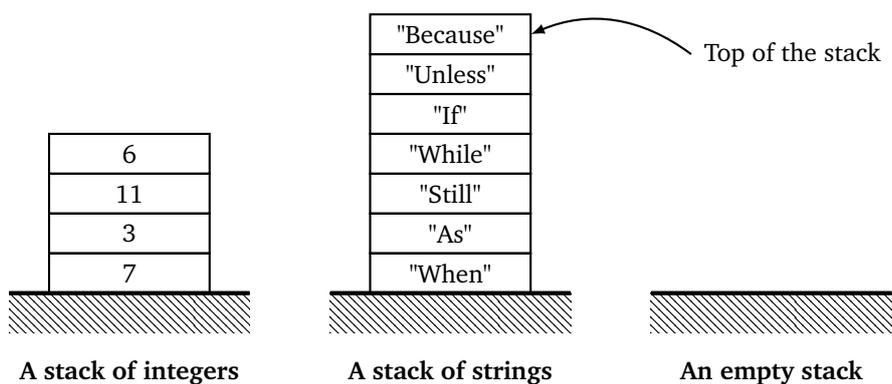
Remarks.

- **Analogy.** You can make the connection with a stack of plates. You can put plates on a stack one by one. You can remove the plates one by one, starting of course with the top one. In addition, it must be considered that on each plate is drawn one piece of data (a number, a character, a string . . .).

- **Last in, first out.** In a queue, the first one to wait is the first one to be served and comes out. Here it's the opposite! A stack works on the principle of "last in, first out".

- In a list, you can directly access any element; in a stack, you can only directly access the element at the top of the stack. To access the other elements, you have to pop several times.

- The advantage of a stack is that it is a very simple data structure that corresponds well to what happens in a computer's memory.

Lesson 2 (Global variable).

A *global variable* is a variable that is defined for the entire program. It is generally not recommended to use such variables but it may be useful in some cases. Let us look at an example.

The global variable, here the gravitational constant, is declared at the beginning of the program as a classic variable:

$$\texttt{gravitation = 9.81}$$

The content of the variable `gravitation` is now available everywhere. On the other hand, if you want to change the value of this variable in a function, you must specify to `Python` that you are aware of modifying a global variable.

For example, for calculations on the Moon, it is necessary to change the gravitational constant, which is much lower there.

```python
def on_the_moon():
    global gravitation   # Yes, I really want to modify this variable!
    gravitation = 1.625  # New value for the entire program
    ...
```

Activity 1 (Stack basic operations).

Goal: define the three (very simple) commands to use the stacks.

In this chapter, a stack will be modeled by a list. The item at the end of the list is the top of the stack.

| 7 |
| 11 |
| 2 |
| 4 |
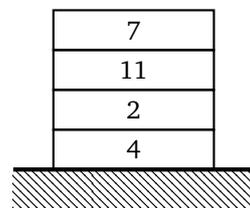
A stack

`stack = [4,2,11,7]`

Model as a list

The stack will be stored in a global variable `stack`. It is necessary to start each function that modifies the stack with the command:

$$\texttt{global stack}$$

1. Write a `push_to_stack()` function that adds an element to the top of the stack.

---

push_to_stack()

Use: `push_to_stack(item)`
Input: an integer, a string...
Output: nothing
Action: the stack contains an additional element

Example: if at the beginning `stack = [5,1,3]` then, after the instruction `push_to_stack(8)`, the stack is `[5,1,3,8]` and if you continue with the instruction `push_to_stack(6)`, the stack is now `[5,1,3,8,6]`.

---

2. Write a `pop_from_stack()` function, without parameters, that removes the element at the top of the stack and returns its value.

> ### pop_from_stack()
>
> Use: `pop_from_stack()`
> Input: nothing
> Output: the element at the top of the stack
> Action: the stack contains one less element
>
> Example: if initially `stack = [13,4,9]` then the instruction `pop_from_stack()` returns the value 9 and the stack is now `[13,4]`; if you execute a new instruction `pop_from_stack()`, it returns this time the value 4 and the stack is now `[13]`.

3. Write an `is_stack_empty()` function, without parameter, that tests if the stack is empty or not.

> ### is_stack_empty()
>
> Use: `is_stack_empty()`
> Input: nothing
> Output: true or false
> Action: does nothing on the stack
>
> Example:
> - if `stack = [13,4,9]` then the instruction `is_stack_empty()` returns `False`,
> - if `stack = []` then the instruction `is_stack_empty()` returns `True`.

Activity 2 (Operations on a stack).

Goal: handle the stack using only the three functions `push_to_stack()`, `pop_from_stack()` *and* `is_stack_empty()`.

In this exercise, we work with a stack of integers. The questions are independent.

1. (a) Starting from an empty stack, arrive at a stack `[5,7,2,4]`.
   (b) Then execute the instructions `pop_from_stack()`, `push_to_stack(8)`, `push_to_stack(1)`, `push_to_stack(1)`, `push_to_stack(3)`. What is the stack now? What does the instruction `pop_from_stack()` now return?
2. Start from a stack. Write an `is_in_stack(item)` function that tests if the stack contains a given element.
3. Start from a stack. Write a function that calculates the sum of the elements of the stack.
4. Start from a stack. Write a function that returns the second last element of the stack (the last element is the one at the very bottom; if this second last element does not exist, the function returns `None`).

Lesson 3 (String manipulation).

1. The function `split()` is a `Python` method that separates a string into pieces. If no separator is specified, the separator is the space character.

> **python: split()**
>
> Use: `string.split(separator)`
> Input: a string `string` and possibly a separator `separator`
> Output: a list of strings
>
> Example:
> - `"To be or not to be.".split()` returns `['To', 'be', 'or', 'not', 'to', 'be.']`
> - `"12.5;17.5;18".split(";")` returns `['12.5', '17.5', '18']`

2. The function `join()` is a `Python` method that gathers a list of strings into a single string. This is the opposite of `split()`.

> **python: join()**
>
> Use: `separator.join(mylist)`
> Input: a list of strings `mylist` and a separator `separator`
> Output: a string
>
> Example:
> - `"".join(["To", "be", "or", "not", "to", "be."])` returns `'Tobeornottobe.'` Spaces are missing.
> - `" ".join(["To", "be", "or", "not", "to", "be."])` returns `'To be or not to be.'` It's better when the separator is a space.
> - `"--".join(["To", "be", "or", "not", "to", "be."])` returns `'To--be--or--not--to--be.'`

3. The `isdigit()` function is a `Python` method that tests if a string contains only numbers. This allows you to test if a string corresponds to a positive integer. Here are some examples: `"1776".isdigit()` returns `True`; `"Hello".isdigit()` returns `False`.

   Remember that you can convert a string into an integer by the `int(string)` command. The following small program tests if a string can be converted into a positive integer:

```python
mystring = "1776"                    # A string
if mystring.isdigit():
    myinteger = int(mystring)        # myinteger is an integer
else:                                # Problem
    print("I don't know how to convert this string to an integer!")
```

Activity 3 (Sorting station).

Goal: solve a sorting problem by modeling a storage area by a stack.

A train has blue wagons with a number and red wagons with a letter.

Unsorted train.

The stationmaster wants to separate the wagons: first all the blues and then all the reds (the order of the blue wagons does not matter, neither does the order of the red wagons).

Sorted train.

For this purpose, there is an output station and a waiting area: a wagon can either be sent directly to the output station or temporarily stored in the waiting area.

Here are the instructions from the stationmaster.

- **Phase 1.** For each wagon in the train:
  - if it is a blue wagon, send it directly to the output station;
  - if it is a red wagon, send it to the waiting area.
- **Phase 2.** Then, move the (red) wagons one by one from the waiting area to the output station by attaching them to the blue ones.

Phase 2

Here is how we will model the train and its waiting area.

- The train is a string of characters made up of a series of numbers (blue wagons) and letters (red wagons) separated by spaces. For example `train = "G 6 Z J 14"`.
- The list of wagons is obtained by calling `train.split()`.
- We test if a wagon is blue by checking if it is marked with a number, using the `wagon.isdigit()` function.
- The train reconstructed by the sorted wagons is also a string of characters. At first, it is the empty string.
- The waiting area will be the stack. At the beginning the stack is empty. We're only going to add the red wagons. At the end, the stack is drained towards the tail end of the reconstituted train.

Following the station manager's instructions and using stack operations, write a `sort_wagons()` function that separates the blue and red wagons from a train.

---

sort_wagons()

Use: `sort_wagons(train)`
Input: a string with blue wagons (numbers) and red wagons (letters)
Output: blue wagons first and red wagons second.
Action: use a stack

Example:
- `sort_wagons("A 4 C 12")` returns `"4 12 C A"`
- `sort_wagons("K 8 P 17 L B R 3 10 2 N")` returns `"8 17 3 10 2 N R B L P K"`

Lesson 4 (Polish notation).

Writing in Polish notation (of its full name, reverse Polish notation) is another way to write an algebraic expression. Its advantage is that this notation does not use brackets and is easier to handle for a computer. Its disadvantage is that we're not used to it.

Here is the classic way to write an algebraic expression (left) and its Polish notation (right). In any case, the result will be 13!

$$\text{Classic:} \quad 7 + 6 \qquad\qquad \text{Polish:} \quad 7 \ 6 \ +$$

Other examples:

- Classic: $(10+5) \times 3$; Polish: 10 5 + 3 ×
- Classic: $10 + 2 \times 3$; Polish: 10 2 3 × +
- Classic: $(2+8) \times (6+11)$; Polish: 2 8 + 6 11 + ×

Let's see how to calculate the value of an expression in Polish notation.
- We read the expression from left to right:

$$2 \ 8 \ + \ 6 \ 11 \ + \ \times \longrightarrow$$

- When you meet a first operator $(+, \times, \ldots)$ you calculate the operation *with the two members just before this operator*:

$$\underbrace{2 \ 8 \ +}_{2+8} \ 6 \ 11 \ + \ \times$$

- This operation is replaced by the result:

$$\underbrace{10}_{\substack{\text{result of} \\ 2+8}} \ 6 \ 11 \ + \ \times$$

- We continue reading the expression (we are looking for the first operator and the two terms just before):

$$10 \ \underbrace{6 \ 11 \ +}_{6+11=17} \ \times \qquad \text{becomes} \qquad 10 \ 17 \ \times \qquad \text{that is equal to} \qquad 170$$

- At the end there is only one value left, it's the result! (Here 170.)

Other examples:
- 8 2 ÷ 3 × 7 +

$$\underbrace{8 \ 2 \ \div}_{8 \div 2 = 4} \ 3 \ \times \ 7 \ + \qquad \text{becomes} \qquad \underbrace{4 \ 3 \ \times}_{4 \times 3 = 12} \ 7 \ + \qquad \text{becomes} \qquad 12 \ 7 \ + \qquad \text{that is equal to} \qquad 19$$

- 11 9 4 3 + − ×

$$11 \ 9 \ \underbrace{4 \ 3 \ +}_{4+3=7} \ - \ \times \quad \text{becomes} \quad 11 \ \underbrace{9 \ 7 \ -}_{9-7=2} \ \times \quad \text{becomes} \quad 11 \ 2 \ \times \quad \text{that is equal to} \quad 22$$

Exercise. Compute the value of the expressions:
- 13 5 + 3 ×
- 3 5 7 × +
- 3 5 7 + ×
- 15 5 ÷ 4 12 + ×

Activity 4 (Polish calculator).

Goal: program a mini-calculator that evaluates expressions in Polish notations.

1. Write a function `operation()` that calculates the sum or product of two numbers.

> ### operation()
>
> Use: `operation(a,b,op)`
> Input: two numbers a and b, one operation character "+" or "*"
> Output: the result of the operation `a + b` or `a * b`
>
> Example:
> - `operation(2,4,"+")` returns 6
> - `operation(2,4,"*")` returns 8

2. Program a Polish calculator, according to the following algorithm:

   Algorithm.
   - – Input: an expression in Polish notation (a string).
     - – Output: the value of this expression.
     - – Example: "2 3 + 4 *" (the calculation $(2+3) \times 4$) returns 20.
   - Start with an empty stack.
   - For each element of the expression (read from left to right):
     - – if the element is a number, then add this number to the stack,
     - – if the element is an operation character, then:
       - – pop the stack once to get a number b,
       - – pop a second time to get a number a,
       - – calculate $a + b$ or $a \times b$ depending on the operation,
       - – push this result to the stack.
   - At the end, the stack contains only one element, it is the result of the calculation.

> ### polish_calculator()
>
> Use: `polish_calculator(expression)`
> Input: an expression in Polish notation (a string)
> Output: the result of the calculation
> Action: uses a stack
>
> Example:
> - `polish_calculator("2 3 4 + +")` returns 9
> - `polish_calculator("2 3 + 5 *")` returns 25

Bonus. Change your code to support subtraction and division!

Activity 5 (Expression with balanced brackets).

Goal: determine if the parentheses in an expression are placed appropriately.

Here are some examples of well and bad balanced bracketed expressions:
- $2 + (3 + b) \times (5 + (a - 4))$ has well balanced parentheses;
- $(a + 8) \times 3) + 4$ is incorrectly bracketed: there is a closing bracket ")" that does not have an opening bracket;

- $(b + 8/5)) + (4$ is incorrectly bracketed: there are as many opening parentheses "(" as closing parentheses ")" but they are poorly positioned.

1. Here is the algorithm that decides if the parentheses of an expression are well placed. The stack acts as an intermediate storage area for opening parenthesis "(". Each time we find a closing parenthesis ")" in the expression we delete an opening parenthesis from the stack.

   Algorithm.
   Input: any expression (a string).
   Output: "True" if the parentheses are well balanced, "False" otherwise.
   - Start with an empty stack.
   - For each character of the expression read from left to right:
     - if the character is neither "(", nor ")" then do nothing!
     - if the character is an opening parenthesis "(" then add this character to the stack;
     - if the character is a closing parenthesis ")":
       - test if the stack is empty, if it is empty then return "False" (the program ends there, the expression is incorrectly parenthesized), if the stack is not empty continue,
       - pop the stack once, it gives a "(".
   - If at the end, the stack is empty then return the value "True", otherwise return "False".

   ┌───┐
   │   **are_parentheses_balanced()**   │
   ├───┤
   │ Use: `are_parentheses_balanced(expression)`   │
   │ Input: an expression (string)   │
   │ Output: true or false depending on whether the parentheses are correctly │
   │ placed or not   │
   │ Action: uses a stack   │
   │   │
   │ Example: │
   │   - `are_parentheses_balanced("(2+3)*(4+(8/2))")` returns   │
   │   `True`   │
   │   - `are_parentheses_balanced("(x+y)*((7+z)")` returns `False` │
   └───┘

2. Enhance this function to test an expression with parentheses and square brackets. Here is a correct expression: $[(a + b) * (a − b)]$, here are two incorrect expressions: $[a + b)$, $(a + b] * [a − b)$. Here is the algorithm to program an `are_brackets_balanced()` function.

Algorithm.

Input: an expression (a string).

Output: "True" if the parentheses and square brackets are well balanced, "False" otherwise.

- Start with an empty stack.
- For each character of the expression read from left to right:
  - if the character is neither "(", nor ")", nor "[", nor "]" then do nothing;
  - if the character is an opening parenthesis "(" or an opening square bracket "[", then add this character to the stack;
  - if the character is a closing parenthesis ")" or a closing square bracket "]", then:
    - test if the stack is empty, if it is empty then return "False" (the program ends there, the expression is not correct), if the stack is not empty continue,
    - pop the stack once, you get a "(" or a "[",
    - if the popped (opening) character does not match the character read in the expression, then return "False". The program ends there, the expression is not consistent. To match means that the character "(" corresponds to ")" and "[" corresponds to "]".
- If at the end, the stack is empty then return the value "True", otherwise return "False".

This time the stack can contain opening parentheses "(" or opening square brackets "[". Each time you find a closing parenthesis ")" in the expression, the top of the stack must be an opening parenthesis "(". Each time you find a closing square bracket "]" in the expression, the top of the stack must be a opening square bracket "[".

Activity 6 (Conversion to Polish notation)**.**

Goal: transform a classic algebraic expression with parentheses into a Polish notation expression. This algorithm is a much improved version of the previous activity. We will not give any justification.

You are used to writing "(13+5)×7" and you have seen that the computer can easily calculate "13 5 + 7 ×". All that remains is to switch from a classical algebraic expression (with parentheses) to Polish notation (without parentheses)!

Here is the algorithm for expressions with only additions and multiplications.

Algorithm.

Input: a classic expression

Output: an expression in Polish notation

- Start with an empty stack.
- Start with an empty string, `polish`, which at the end will contain the result.
- For each character of the expression (read from left to right):
  - if the character is a number, then add this number to the output string `polish`;
  - if the character is an opening parenthesis `"("`, then add this character to the stack;
  - if the character is the multiplication operator `"*"`, then add this character to the stack;
  - if the character is the addition operator `"+"`, then:

    while the the stack is not empty:

    > pop an element from the stack,
    > if this element is the multiplication operator `"*"`, then:
    >> add this element `"*"` to the output string `polish`
    > otherwise:
    >> push this element `"*"` to the stack (we put it back on the stack after removing it)
    >> end the "while" loop immediately (with `break`)

    finally, add the `"+"` addition operator to the stack.
  - if the character is a closing parenthesis `")"`, then:

    while the stack is not empty:

    > pop an element from the stack,
    > if this element is an opening parenthesis `"("`, then:
    >> end the "while" loop immediately (with `break`)
    > otherwise:
    >> add this item to the output string `polish`

- If at the end, the stack is not empty, then add each element of the stack to the output string `polish`.

`polish_notation()`
Use: `polish_notation(expression)` Input: a classic expression (with elements separated by spaces) Output: the expression in Polish notation Action: use the stack Example: • `polish_notation("2 + 3")` returns `"2 3 +"` • `polish_notation("4 * ( 2 + 3 )")` returns `"4 2 3 + *"` • `polish_notation("( 2 + 3 ) * ( 4 + 8 )")` returns `"2 3 + 4 8 + *"`

In this algorithm, each element between two spaces of an expression is miscalled "character". Example: the characters of `"( 17 + 10 ) * 3"` are (, 17, +, 10,), * and 3.

> You see that addition is more complicated to process than the multiplication. This is due to the fact that multiplication takes priority over addition. For example, $2+3\times 5$ means $2+(3\times 5)$ and not $(2+3)\times 5$. If you want to take into account subtraction and division, you have to be careful about non-commutativity ($a-b$ is not equal to $b-a$ and $a \div b$ is not equal to $b \div a$).

End this chapter by checking that everything works correctly with different expressions. For example:

- Define an expression `exp = "( 17 * ( 2 + 3 ) ) + ( 4 + ( 8 * 5 ) )"`

- Ask `Python` to calculate this expression: `eval(exp)`. `Python` returns 129.

- Convert the expression to Polish notation: `polish_notation(exp)` returns
$$\texttt{"17 2 3 + * 4 8 5 * + +"}$$

- With your calculator calculate the result: `polish_calculator("17 2 3 + * 4 8 5 * + +")` returns 129. We get the same result!

Text viewer – Markdown

You will program a simple word processor that displays paragraphs cleanly and highlights words in bold and italics.

Lesson 1 (Text with tkinter).
Here's how to display text with `Python` and the graphics window module `tkinter`.

<div align="center">

Text with Python!

</div>

The code is:

```python
from tkinter import *
from tkinter.font import Font
# tkinter window
root = Tk()
canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(fill="both", expand=True)
# Font
myfont = Font(family="Times", size=30)
# Some text
canvas.create_text(100,100, text="Text with Python!",
anchor=SW, font=myfont, fill="blue")
# Launch the window
root.mainloop()
```

Some explanations:

- `root` and `canvas` are the variables that define a graphic window (here of width 800 and height 600 pixels). This window is launched by the last command: `root.mainloop()`.

- We remind you that for the graphic coordinates, the y-axis is directed downwards and the origin is the top-left corner. To define a rectangle, simply specify the coordinates (x_1, y_1) and (x_2, y_2) from two opposite vertices (see figure below).

- The text is displayed by the `canvas.create_text()` command. It is necessary to specify the coordinates (x, y) of the point from which you want to display the text.

- The `text` option allows you to pass the string to display.

- The `anchor` option allows you to specify the text anchor point, `anchor=SW` means that the text box is anchored to the Southwest point (*SW*) (see figure below).

- The `fill` option allows you to specify the text color.
- The option `font` allows you to define the font (i.e. the style and size of the characters). Here are some examples of fonts, it's up to you to test them:
    - `Font(family="Times", size=20)`
    - `Font(family="Courier", size=16, weight="bold")` in **bold**
    - `Font(family="Helvetica", size=16, slant="italic")` in *italics*
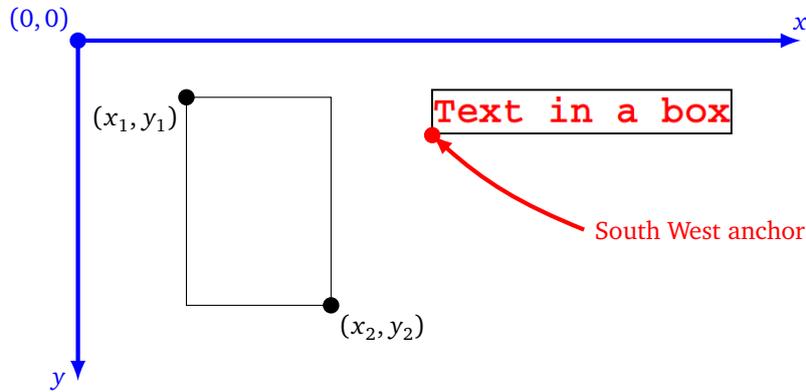
Activity 1 (Display a text with tkinter)**.**

Goal: display text with the graphics module `tkinter`*.*

1. (a) Define a `tkinter` window of size 800×600 for example.
    (b) Draw a gray rectangle (which will be our background box) with a size of `back_width` × `back_height` (for example 700×500).
    (c) Define several types of fonts: `title_font`, `subtitle_font`, `bold_font`, `italic_font`, `text_font`.
    (d) Display texts with different fonts.
2. Write a `text_box(word,font)` function that draws a rectangle around a text. To do this, use the `canvas.bbox(myobject)` method which returns the x_1, y_1, x_2, y_2 coordinates of the desired rectangle. (Here `myobject = canvas.create_text(...)`).
3. Write a `length_word(word,font)` function that calculates the length of a word in pixels (this is the width of the rectangle from the previous question).
4. Write a `font_choice(mode,in_bold,in_italics)` function that returns the name of an adapted font (among those defined in the first question) according to a mode (among `"title"`, `"subtitle"`, `"text"`) and according to booleans `in_bold`, `in_italics`.
    For example, `font_choice("text",True,False)` returns the font `bold_font`.

Lesson 2 (Markdown)**.**
The *Markdown* is a simple markup language that allows you to write your own easy to read text file and possibly convert it to another format (html, pdf...).

Here is an example of a text file with the *Markdown* syntax with its graphic rendering just below.

```
# On the Origins of Species

## by Charles Darwin

When on board H.M.S. * Beagle * as naturalist, I was much struck with
certain facts in the distribution of the inhabitants of South America,
and in the geological relations of the present to the past inhabitants
of that continent. These facts seemed to me to throw some light on the
** origin of species ** that mystery of mysteries, as it has been called
by one of our greatest philosophers.

## Chapters

+ Variation under domestication.
+ Variation under nature.
+ Struggle for existence.
+ Natural selection.
+ ...
```

On the Origins of Species

by Charles Darwin

When on board H.M.S. *Beagle* as naturalist, I was much struck with certain facts in the distribution of the inhabitants of South America, and in the geological relations of the present to the past inhabitants of that continent. These facts seemed to me to throw some light on the **origin of species** that mystery of mysteries, as it has been called by one of our greatest philosophers.

Chapters

• Variation under domestication.
• Variation under nature.
• Struggle for existence.
• Natural selection.
• ...

The syntax is simple, with a clear and clean text file. Here are some elements of this syntax:
- a **bold text** is obtained by surrounding the text with two asterisks **;
- a *text in italics* is obtained by surrounding the text with one asterisk *;
- the line of a title begins with one hashtag #;
- the line of a subtitle begins with two hashtags ##;
- for the elements of a list, each line starts with a special symbol, for us it will be the "plus" symbol +.
- There is also syntax to display links, tables, code...

In the following we will use the simplified syntax as described above.

Activity 2 (View *Markdown*).

Goal: display text with our simplified Markdown *syntax.*

1. Write a function `print_line_v1(par,posy)` that displays *one by one* the words of a paragraph `par` (at the line of ordinate `posy`).
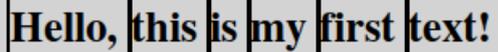
Hints.

- These words are obtained one by one with the command `par.split()`.
- The displayed line starts at the very left, it overflows to the right if it is too long.
- After each word we place a space and then the next word.
- In the picture above the words are framed.

2. Improve your function in `print_line_v2(par,posy)` to take into account titles, subtitles and lists.
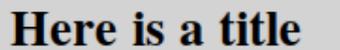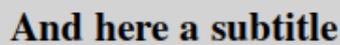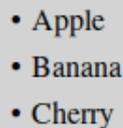
Hints.

- To know which mode to use when displaying the line, simply test the first characters of the line. The line of a title begins with #, that of a subtitle with ##, that of a list with +.
- For lists, you can get the bullet point character "•" by the unicode character `u'\u2022'`. You can also indent each item of the list for more readability.
- Use the `font_choice()` function from the first activity.
- In the image above, each line is produced by a call to the function. For example `print_line_v2("## And here a subtitle",100)`.

3. Further improve your function in `print_line_v3(par,posy)` to take into account the words in bold and italics in the text.
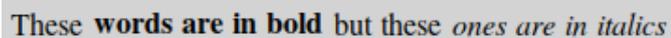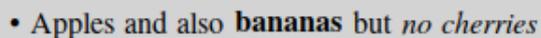
Hints.

- Words in bold are surrounded by the tag **, words in italics by the tag *. In our simplified syntax, tags are separated from words by spaces, for example: `"Words ** in bold ** and in * italics *"`
- Define a boolean variable `in_bold` that is false at the beginning; each time you encounter the tag ** reverse the value of `in_bold` ("True" becomes "False", "False" becomes "True", you can use the `not` operator).
- Still use the `font_choice()` function from the first activity.
- In the image above, each line is produced by a call to the function. For example `print_line_v3("+ Apples and also ** bananas ** but * no cherries *",100)`

4. Further improve your function in `print_paragraph(par,posy)` which manages the display of a paragraph (i.e. a string that can be very long) on several lines.

> Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam nec dui ac sem molestie viverra quis sit amet felis. Donec felis mi, tempus in laoreet non, pellentesque non sem. Praesent pretium mi at odio congue eleifend. Integer magna neque, feugiat a commodo eget, malesuada in velit. Donec ac orci quis eros molestie lacinia. Sed nisi mi, pretium et tellus eget, dignissim venenatis felis. Mauris sit amet ex in metus ornare cursus non nec sapien.Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam nec dui ac sem molestie viverra quis sit amet felis. Donec felis mi, tempus in laoreet non, pellentesque non sem. Praesent pretium mi at odio congue eleifend. Integer magna neque, feugiat a commodo eget, malesuada in velit. Donec ac orci quis eros molestie lacinia. Sed nisi mi, pretium et tellus eget, dignissim venenatis felis. Mauris sit amet ex in metus ornare cursus non nec sapien.

Hints.
- As soon as you place a word that exceeds the length of the line (see those that come out of the frame in the image above), then the next word is placed on the next line.
- The function will therefore modify the variable `posy` at each line break. At the end, the function returns the new value of `posy`, which will be useful for displaying the next paragraph.

5. End with a `print_file(filename)` function that displays the paragraphs of a text file with our simplified *Markdown* syntax.

Activity 3 (Justification).

Goal: understand how it is possible to "justify" a text, i.e. to ensure that the words are well aligned on the left and right sides of the page. To model the problem we work with a series of integers that represent the lengths of our words.

In this activity:
- `list_lengths` is a list of integers (for example a list of 50 integers between 5 and 15) which represent the lengths of the words;
- we set a constant `line_length` which represents the length of a line. For our examples, this length is equal to 100.

In previous activities, we moved to the next line after a word had passed the end of the line. We represent this by the following figure:
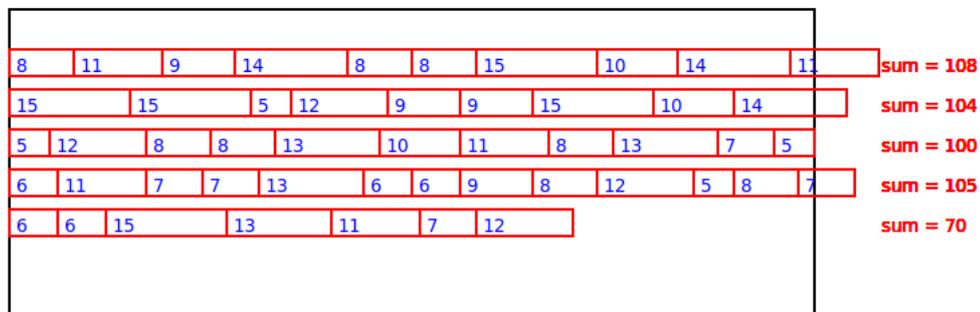
You're going to try to place the words more nicely!
The drawings are based on the example:

```
list_lengths = [8, 11, 9, 14, 8, 8, 15, 10, 14, 11, 15, 15, 5, 12, 9, 9, 15, 10, 14, 5, 12, 8, 8, 13, 10,
        11, 8, 13, 7, 5, 6, 11, 7, 7, 13, 6, 6, 9, 8, 12, 5, 8, 7, 6, 6, 15, 13, 11, 7, 12]
```

which was obtained by random integers:

```
from random import randint
list_lengths = [randint(5,15) for i in range(50)]
```

1. Write a `justification_simple()` function that calculates the indices at which to make the text alignment corresponding to the figure below, i.e. an alignment on the left (without spaces) and without exceeding the total length of the line (here of length 100).
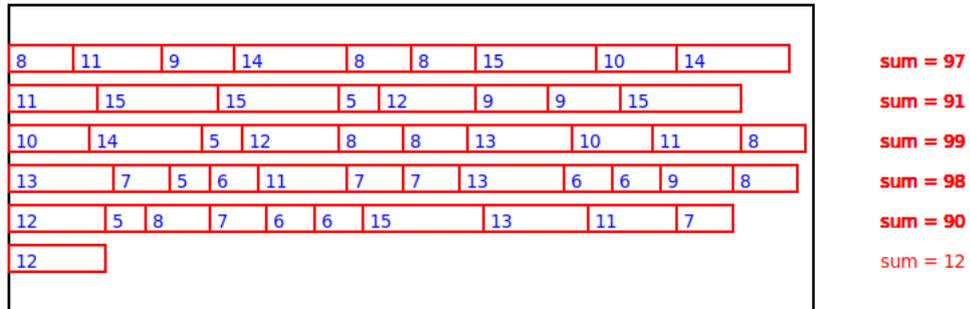
justification_simple()

Use: `justification_simple(list_lengths)`
Input: a sequence of lengths (a list of integers)
Output: the list of indices where to make a cut

Example: `justification_simple(list_lengths)` where `list_lengths` is the example given above, returns the list `[0, 9, 17, 27, 39, 49, 50]`. That is to say that:
- the first line corresponds to indices 0 to 8 (given by `range(0,9)`),
- the second line corresponds to indices 9 to 16 (given by `range(9,17)`),
- the third line corresponds to indices 17 to 26 (given by `range(17,27)`),
- ...
- the last line corresponds to the index 49 (given by `range(49,50)`).

2. Modify your work into a `justification_spaces()` function that adds a space (with `space_length = 1`) between two words of the same line (but not at the beginning of the line, nor at the end of the line). This corresponds to the following drawing:
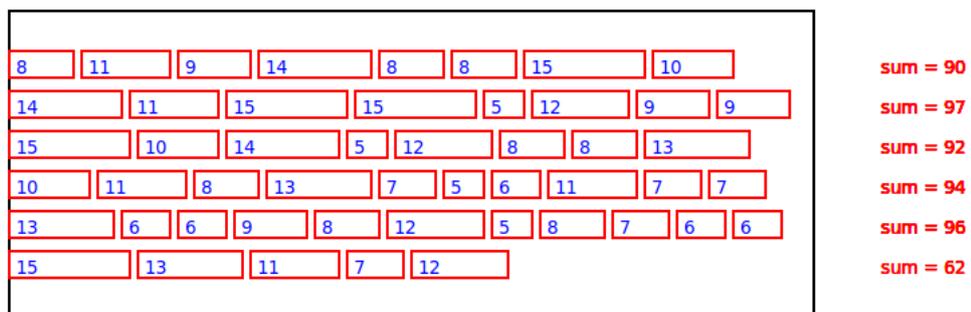
For our example, the justifications returned are `[0, 8, 16, 24, 34, 45, 50]`.

3. In order to be able to justify the text, you allow spaces to be larger than the initial length of 1. On each line, the spaces between the words are all the same length (greater than or equal to 1) so that the last word is aligned on the right. From one line to another, the length of the spaces can change.

Write a `compute_space_lengths()` function that returns the length that the spaces of each line must have in order for the text to be justified. For our example, we obtain the list `[2.43, 1.43, 2.14, 1.67, 1.40, 1.00]`, i.e. for the first line the spaces must be 2.43 long, for the second line 1.43,…

To find the right formula, simply take the results of the `justification_spaces()` function and then, for each line, count the total length of words it contains, as well as the value it lacks to arrive at a total of 100.

You now have everything you need to visualize text written with the Markdown *syntax and justify it (see Darwin's text displayed text in the lesson). It's still a lot of work! You can also improve the support of the* Markdown *syntax: code, numbered lists, sublists, bold and italic words at the same time…*

L-systems

L-systems offer a very simple way to code complex phenomena. From an initial word and a number of replacement operation, we arrive at complicated words. When you "draw" these words, you get beautiful fractal figures. The "L" comes from the botanist A. Lindenmayer who invented L-systems to model plants.
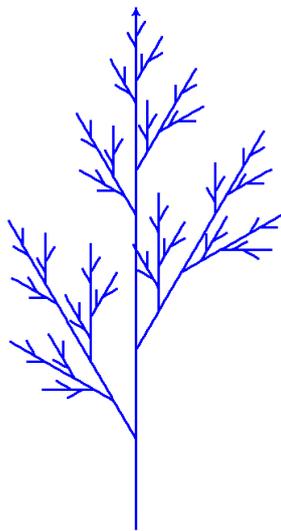
Lesson 1 (L-system)**.**

An *L-system* is the data of an initial word and replacement rules. Here is an example with a starting word and only one rule:

$$\textbf{BlArB} \qquad \textbf{A} \rightarrow \textbf{ABA}$$

The *k-iteration* of the L-system is obtained by applying the substitution to the starting word k times. Using our example:

- First iteration. The starting word is **BlArB**, the rule is **A** → **ABA**: we replace the **A** by **ABA**. We get the word **BlABArB**.

- Second iteration. We start from the word obtained **BlABArB**, we replace the two **A** by **ABA**: we get the word **BlABABABArB**.

- The third iteration is **BlABABABABABABABArB**, etc.

When there are two (or more) rules, they must be applied at the same time. Here is an example of a two-rule L-system:

$$\textbf{A} \qquad \textbf{A} \rightarrow \textbf{BlA} \qquad \textbf{B} \rightarrow \textbf{BB}$$

With our example:

- First iteration. The starting word is **A**, we apply the first rule **A** → **BlA** (the second rule does not

apply, because there is no **B** yet): we get the word **BlA**.

- Second iteration. We start from the word obtained **BlA**, we replace the **A** by **BlA** and at the same time the **B** by **BB**: we get the word **BBlBlA**.

- The third iteration is **BBBBlBBlBlA**, etc.

Lesson 2 (Optional argument for a function)**.**

I want to program a function that draws a line of a given length, with the possibility to change the thickness of the line and the color.

One method would be to define a function by:

```
def draw(length, width, color):
```

I would then call it like this:

```
draw(100, 5, "blue"):
```

But since my features will, most of the time, have a thickness of 5 and a blue color, I lose time and legibility by giving this information each time.

With `Python` it is possible to create optional arguments. There is a way to use optional arguments by giving the function default values:

```
def draw(length, width=5, color="blue"):
```

- The command `draw(100)` draws my line, and as I only specified the length, the arguments `width` and `color` get the default values (5 and blue).

- The command `draw(100, width=10)` draws my line with a new thickness (the color is the default one).

- The command `draw(100, color="red")` draws my line with a new color (the thickness is the default one).

- The command `draw(100, width=10, color="red")` draws my line with a new thickness and a new color.

- We can also use:
    - `draw(100, 10, "red")`: no need to specify the names of the arguments if you maintain the order.
    - `draw(color="red", width=10, length=100)`: if you name the arguments, then you can pass them in any order.

Activity 1 (Draw a word)**.**

Goal: make a drawing from a "word". Each character corresponds to a turtle instruction.

You are given a word (for example **AlArAArArA**) in which each letter (read from left to right) corresponds to an instruction for `Python`'s turtle.

- **A** or **B**: advance by a fixed distance (by tracing),

- **l**: turn left, without moving forward, by a fixed angle (most often 90 degrees),

- **r**: turns right by a fixed angle.

The other characters do not do anything. (More commands will be added later on.)

Program a `draw_lsystem(word,angle=90,scale=1)` function which displays the drawing corresponding to the letters of a string `word`. By default the angle is 90 degrees, and the distance you move forward is 100×`scale`.

For example: `draw_lsystem("AlArAArArA")` displays this:
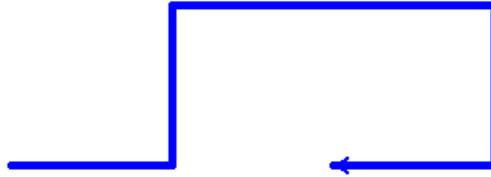
Activity 2 (Only one rule – Koch's snowflake)**.**

Goal: draw the Koch snowflake from a word obtained by iterations.
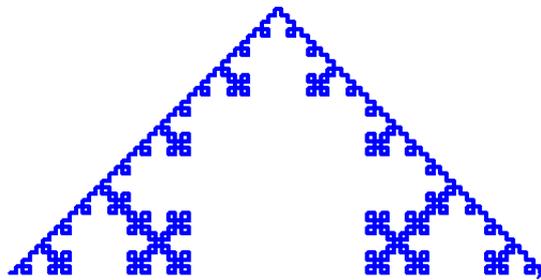
1. Program a `replace_1(word,letter,pattern)` function that replaces a letter with a pattern in a word.
   For example with `word = "ArAAl"`, `replace_1(word,"A","Al")` returns the word `AlrAlAll`: each letter **A** has been replaced by the pattern **Al**.
2. Program an `iterate_lsystem_1(start,rule,k)` function which calculates the *k*-iteration of the L-system associated with the initial word `start` according to the rule `rule` which contains the pair formed by the letter and its replacement pattern. For example, with:
   - `start = "A"`
   - `rule = ("A","AlArArAlA")` i.e. **A → AlArArAlA**
   - for `k = 0`, the function returns the starting word `A`,
   - for `k = 1`, the function returns `AlArArAlA`,
   - for `k = 2`, the function returns:

     `AlArArAlAlAlArArAlArAlArArAlArAlArArAlAlAlArArAlA`
   - for `k = 3`, the function returns: `AlArArAlAlAl...` a word of 249 letters.
3. Trace the first images of the Koch's snowflake given as above by:

   start: **A**   rule: **A → AlArArAlA**

   Here are the images for *k* = 1 up to *k* = 5. For *k* = 1, the word is **AlArArAlA**, you can draw yourself and confirm the trace of the first image.
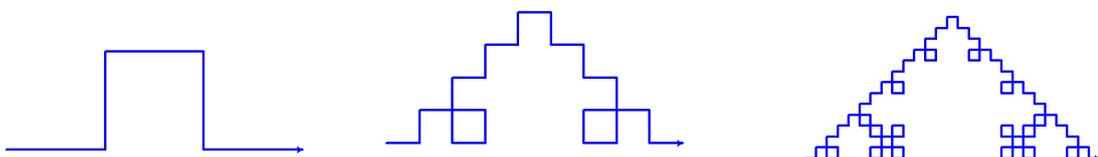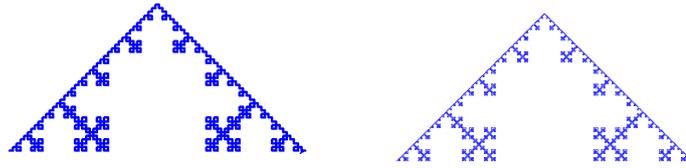
4. Trace other fractal figures from the following L-systems. For all these examples the starting word is `"ArArArA"` (a square) and the rule is to be chosen among:

   - (`"A"`,`"ArAlAlAArArAlA"`)
   - (`"A"`,`"AlAArAArArAlAlAArArAlAlAAlAArA"`)
   - (`"A"`,`"AArArArArAA"`)
   - (`"A"`,`"AArArrArA"`)
   - (`"A"`,`"AArArArArArAlA"`)
   - (`"A"`,`"AArAlArArAA"`)
   - (`"A"`,`"ArAArrArA"`)
   - (`"A"`,`"ArAlArArA"`)
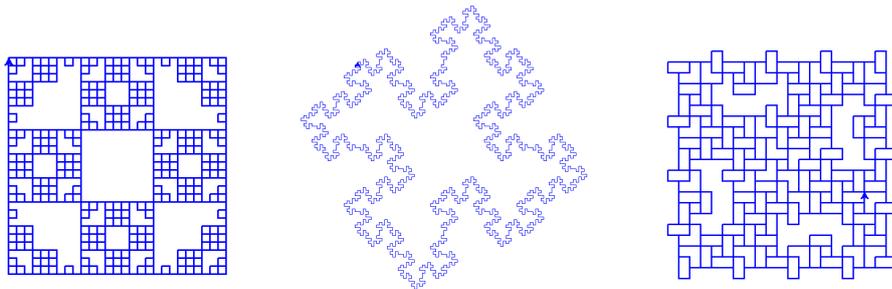
Invent and trace your own L-systems!

Activity 3 (Two rules – Sierpinski triangle).

Goal: draw more complicated L-systems by allowing two replacement rules instead of one.
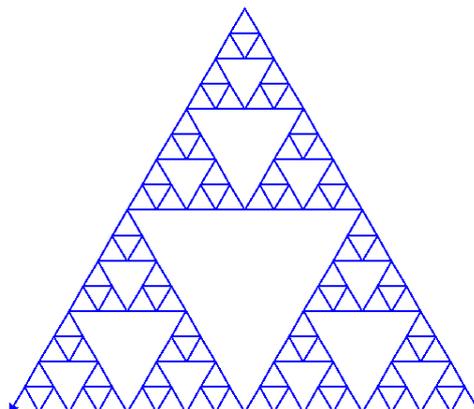
1. Program a `replace_2(word,letter1,pattern1,letter2,pattern2)` function that replaces the first letter with a pattern and the second letter with another pattern.

For example when `word` = `"ArBlA"`, `replace_2(word,"A","ABl","B","Br")` returns the word `ABlrBrlABl`: each letter **A** has been replaced by the pattern **ABl** and at the same time each letter **B** has been replaced by **Br**.

Warning! You should not get `ABrlrBrlABrl`. If this is the case, it is because you used the `replace_1()` function first to replace the A, then a second time to replace the B (but after the first replacement, new B appeared). A new function must be programmed to avoid this.

2. Program an `iterate_lsystems_2(start,rule1,rule2,k)` function which calculates the *k*-iteration of the L-system associated with the initial word `start`, according to the rules `rule1` and `rule2`. For example, with:
   - `start = "ArBrB"`
   - `rule1 = ("A","ArBlAlBrA")` i.e. **A → ArBlAlBrA**
   - `rule2 = ("B","BB")` i.e. **B → BB**
   - for `k = 0`, the function returns the starting word `ArBrB`,
   - for `k = 1`, the function returns `ArBlAlBrArBBrBB`,
   - for `k = 2`, the function returns:
     $$ArBlAlBrArBBlArBlAlBrAlBBrArBlAlBrArBBBBrBBBB$$

3. Trace the first pictures of the Sierpinski triangle given as above by:
   $$\text{start: } \mathbf{ArBrB} \qquad \text{rules: } \mathbf{A \to ArBlAlBrA} \qquad \mathbf{B \to BB}$$
   The angle is −120 degrees. Here are the images for $k = 0$ up to $k = 4$.
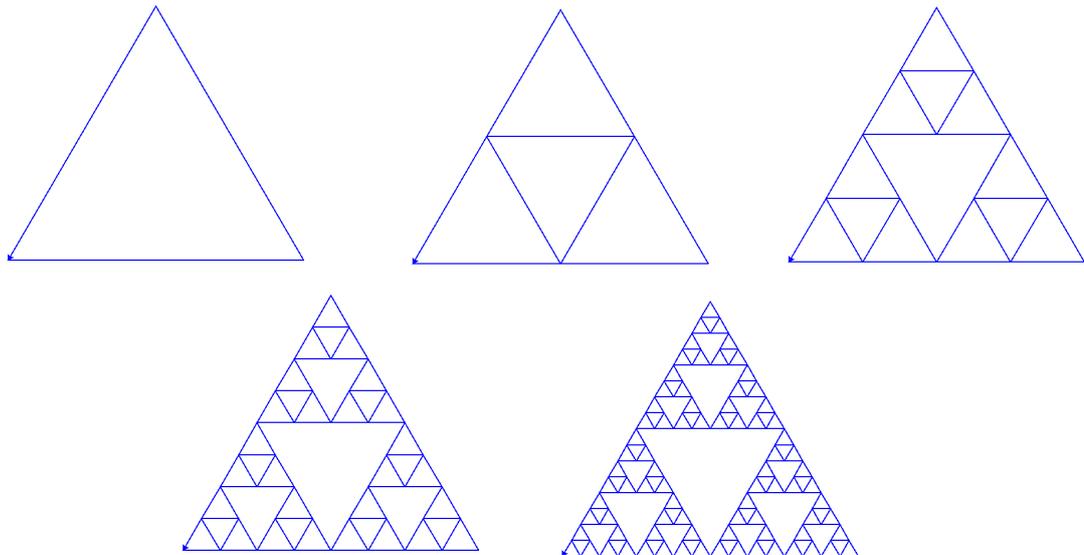
4. Trace other fractal figures from the following L-systems.
   - The dragon curve:
     $$\text{start=}"AX" \text{ rule1=}("X","XlYAl") \text{ rule2=}("Y","rAXrY")$$
     The letters X and Y do not correspond to an action.
   - A variant of the Sierpinski triangle, where `angle = 60`:
     $$\text{start=}"A" \text{ rule1=}("A","BrArB") \text{ rule2=}("B","AlBlA")$$
   - The Gosper curve, where `angle = 60`:
     $$\text{start=}"A"$$
     $$\text{rule1=}("A","AlBllBrArrAArBl")$$
     $$\text{rule2=}("B","rAlBBllBlArrArB")$$
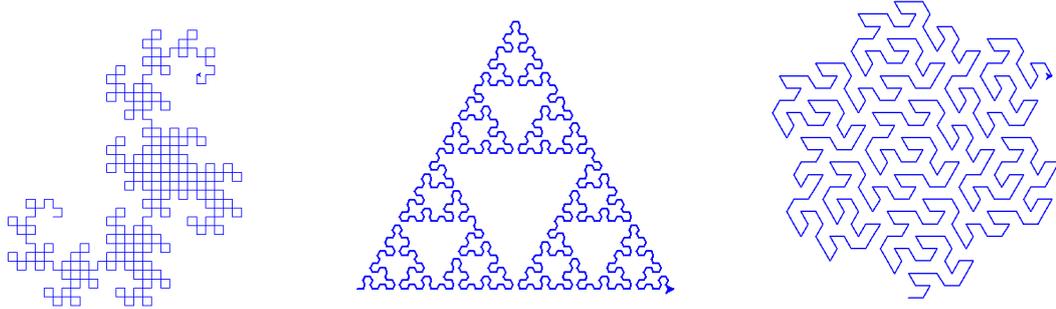
Invent and trace your own L-systems with two rules!

Lesson 3 (Stacks).

A *stack* is a temporary storage area. Details are in the "Polish calculator – Stacks" chapter. Here are just a few reminders.
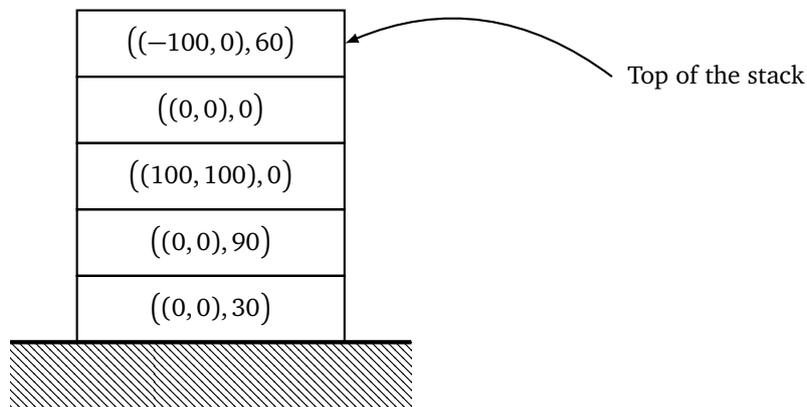
A stack

* A stack is like a stack of plates; elements are placed one by one to the top of the stack; the elements are removed one by one, always from the top. It follows the "last in, first out" principle.
* We model a stack using a list.
* At the beginning the stack is empty: `stack = []`.
* **Push.** We add the items to the end of the list: `stack.append(element)` or `stack = stack + [element]`.
* **Pop.** An item is removed by using the `pop()` command:
$$\text{element = stack.pop()}$$
which returns the last item in the stack and removes it from the list.
* On the drawing and in the next activity, the elements of the stack are of type $\big((x,y),\theta\big)$ that will store a state of the turtle: (x,y) is the position and θ is its direction.

Activity 4 (L-system, stack and turtle).

Goal: improve our drawings by allowing us to move forward without tracing and also by using a kind of flashback, to trace plants.
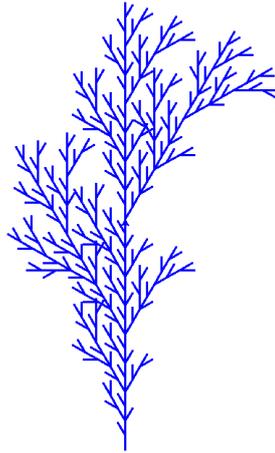
1. **Forward without tracing.**

   Increase the possibilities by allowing the turtle to move forward without drawing a line, when the instruction is the letter **a** (in lowercase). (It is sufficient to modify the `trace_lsystems()` function.)

   Then trace the following L-system:
   - `start = "ArArArA"`
   - `rule1 = ("A","AlarAAlAlAAlAalAAralAArArAArAarAAA")`
   - `rule2 = ("a","aaaaaa")`
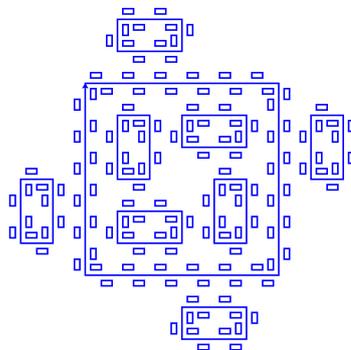
2. **Return back.**

   We now allow square brackets in our words. For example **AlA[lAAA]A[rAA]A**. When you encounter a opening bracket "[", you store the position of the turtle, then the commands in brackets are executed as usual, when you reach the closing bracket "]" you go back to the stored position.

   Let us work through the example: **AlA [lAAA] A [rAA] A**

   AlA [lAAA] A [rAA] A
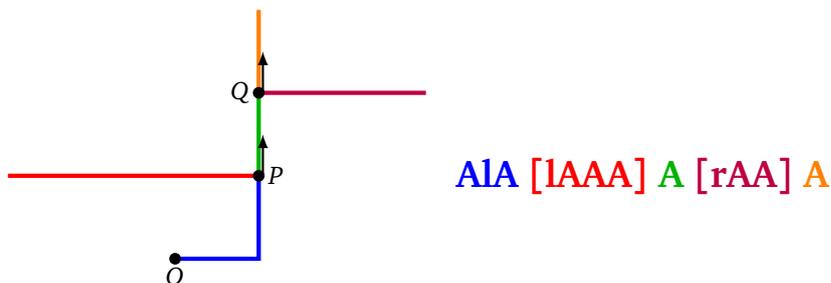
   - **AlA**: we start from the point O, we move forward, we turn, we move forward.

- **[lAAA]**: we store the current position (the point *P*) and the direction; we turn, we advance three times (we trace the red segment); at the end we return the turtle to the position *P* (without tracing and with the same direction as before).

- **A**: from *P* we advance (green segment).

- **[rAA]**: we store the position *Q* and the direction, we turn and we trace the purple segment. We come back to *Q* with the old state.

- **A**: from *Q* we trace the last segment.

Here is how to draw a word containing brackets using a stack:

- At the beginning the stack is empty.

- We read the characters of the word one by one. The actions are the same as before.

- If the character is the opening bracket "[" then add the current position and direction of the turtle $\big((x, y), \theta\big)$ to the stack. You get $\big((x, y), \theta\big)$ by (`position()`, `heading()`).

- If the character is the closing bracket "]" then pop (i.e. read the top element of the stack and remove it). Set the position of the turtle and the angle to the read values. Use `goto()` and `setheading()`.

3. Trace the following L-systems, where the starting word and rule (or rules) are given. The angle is to be chosen between 20 and 30 degrees.

- `"A"`   `("A","A[lA]A[rA][A]")`

- `"A"`   `("A","A[lA]A[rA]A")`

- `"A"`   `("A","AAr[rAlAlA]l[lArArA]")`

- `"X"`   `("X","A[lX][X]A[lX]rAX")`   `("A","AA")`

- `"X"`   `("X","A[lX]A[rX]AX")`   `("A","AA")`

- `"X"`   `("X","Ar[[X]lX]lA[lAX]rX")`   `("A","AA")`
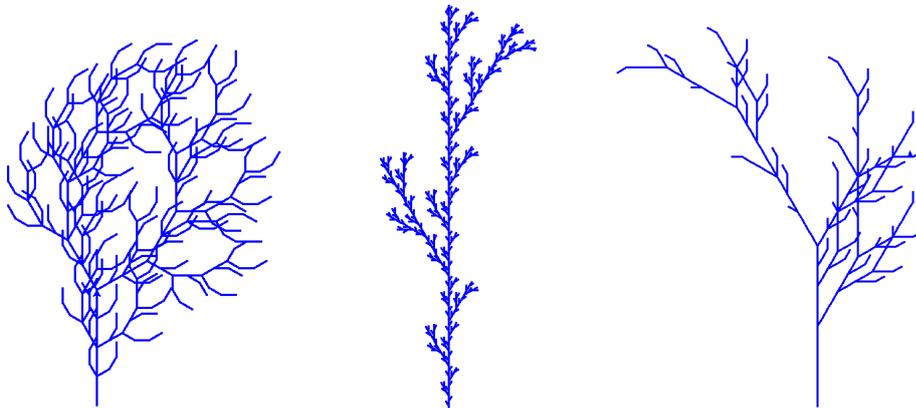
Invent your own plant!

Dynamic images

We will distort images. By repeating these distortions, the images become blurred. But by a miracle after a certain number of repetitions the original image reappears!

Lesson 1 (Photo booth transformation)**.**

We start from an $n \times n$ array, with even n, each element of the table represents a pixel. The rows are indexed from $i = 0$ to $i = n - 1$, the columns from $j = 0$ to $j = n - 1$. From this image we calculate a new image by moving each pixel according to a transformation, called the *photo booth transformation*.

We cut the original image into small 2×2 squares. Each small square is therefore composed of four pixels. Each of these pixels is sent to four different locations in the new image: the pixel at the top left remains in an area at the top left, the pixel at the top right of the small square is sent to an area at the top right of the new image,...
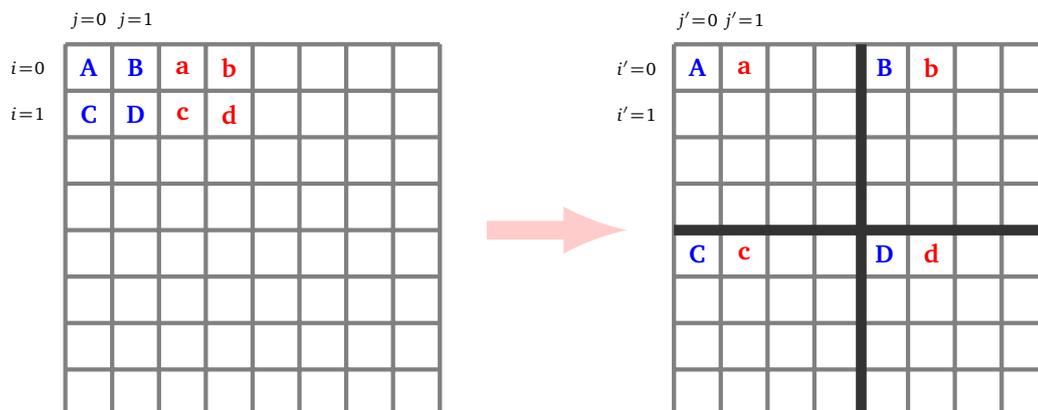
For example, the pixel in position $(1,1)$ (symbolized by the letter **D**) is sent to position $(4,4)$.

Let's explain this principle through formulas. For each pair (i, j), we calculate its image (i', j') using the photo booth transformation according to the following formulas:

- If i and j are even: $(i', j') = (i//2, j//2)$.
- If i is even and j is odd: $(i', j') = (i//2, (n + j)//2)$.
- If i is odd and j is even: $(i', j') = ((n + i)//2, j//2)$.
- If i and j are odd: $(i', j') = ((n + i)//2, (n + j)//2)$.

Here is an example of a 4×4 array before (left) and after (right) the photo booth transformation.

1	2	3	4		1	3	2	4
5	6	7	8		9	11	10	12
9	10	11	12		5	7	6	8
13	14	15	16		13	15	14	16

Here is a 256 × 256 image and its first transformation:
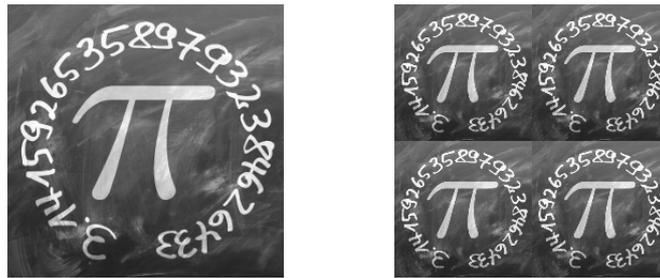
Here is what happens if you repeat the photo booth transformation several times:
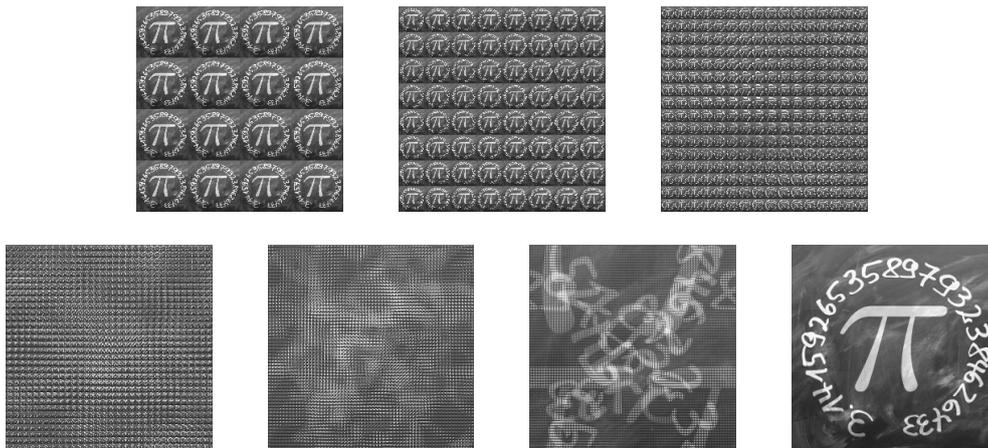
The image becomes more and more blurred, but after some number transformation, we fall back to the original image!

Activity 1 (Photo booth transformation).

Goal: program the photo booth transformation that decomposes an image into sub-pictures. When this transformation is iterated, the image gradually disintegrates, then suddenly re-formes.

1. Program a `transformation(i,j,n)` function that uses the photo booth transformation formula and returns the new coordinates (i', j') of the pixel (i, j), in the image.
   For example, `transformation(1,1,8)` returns `(4,4)`.
2. Program a `photo_booth(array)` function that returns the table calculated by completing a transformation.
   For example, the array on the left is transformed into the array on the right.

1	2	3	4		1	3	2	4
5	6	7	8		9	11	10	12
9	10	11	12		5	7	6	8
13	14	15	16		13	15	14	16

   Hints. You can initialize a new table with the command:
   ```
   new_array = [[0 for j in range(n)] for i in range(n)]
   ```
   Then fill it with commands of the type:
   ```
   new_array[ii][jj] = array[i][j]
   ```
3. Program a `photo_booth_iterate(array,k)` function that returns the table calculated after *k* iterations of the photo booth transformation.

4. *To be finished after completing activity 2.*
   Program a `photo_booth_images(image_name,kmax)` function that calculates the images corresponding to the transformation, for all iterations from $k = 1$ to $k = k_{max}$.

5. Experiment for different values of the size n, to see after how many iterations we return to the original image.

Here is the starting image of size 256×256 and the images obtained by iterations of the photo booth transformation for $k = 1$ up to $k = 8$. After 8 iterations we return to the initial image again.
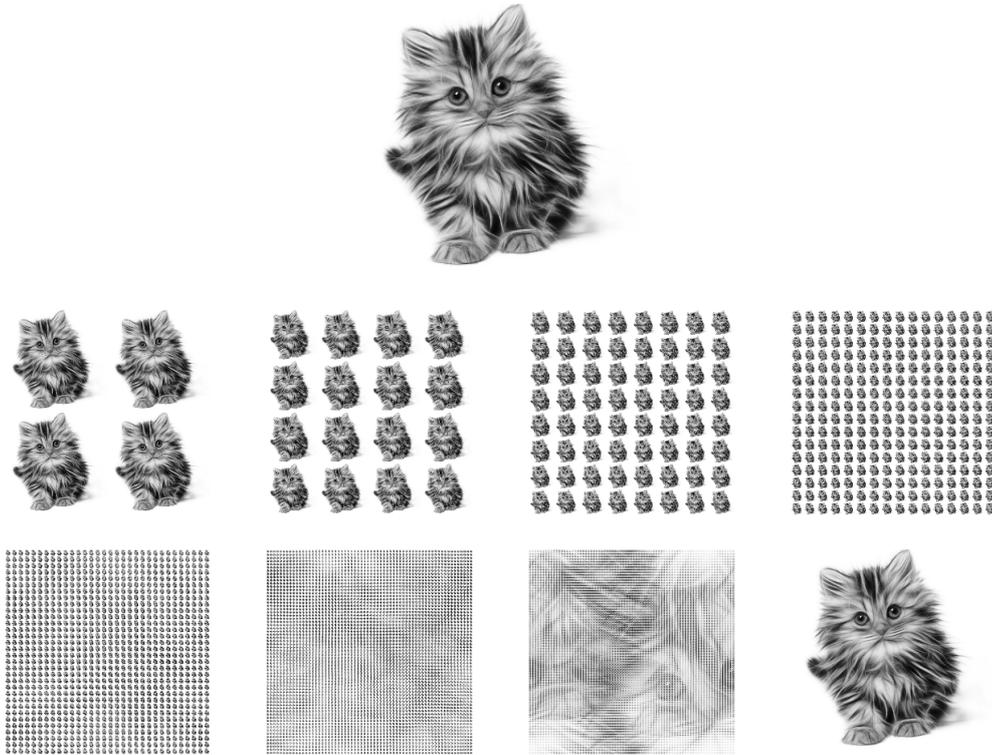
Activity 2 (Conversion array/image).

Goal: switch from an array to an image file and vice versa. The image is displayed in the "pgm" format which has been manipulated in the "Files" chapter.

1. **Array to image.**
   Program an `array_to_image(array,image_name)` function that writes an image from a grayscale table to a file in "pgm" format.

```
P2
5 5
255
128 192 128 192 128
224   0 228   0 224
228 228 228 228 228
224  64  64  64 224
192 192 192 192 192
```
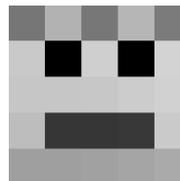

For example, with `array = [ [128,192,128,192,128], [224,...] ]`, the `array_to_image(array,"test")` command writes a `test.pgm` file (on the left) to be displayed as the image on the right.

2. **Image to array.**

   Program an `image_to_array(image_name)` function which takes an image file in "pgm" format and returns an array of gray levels.

Lesson 2 (Baker's transformation).

We start from an $n \times n$ array, with even n, where each element represents a pixel. We will apply two elementary transformations each time:

- **Stretching.** The principle is as follows: the first two lines (each with a length of n) produce a single line with a length of $2n$. We mix the values of each line by alternating an upper element and a lower element.
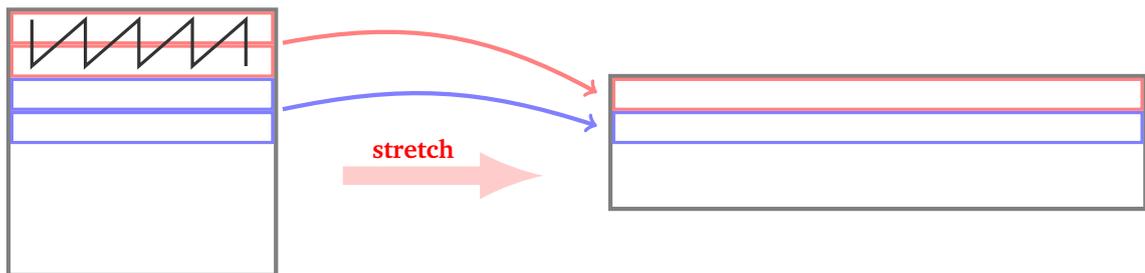
Here is how two lines mix into one:
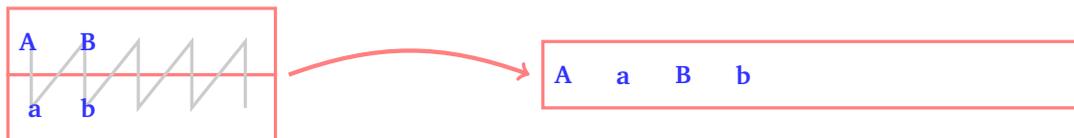
Formulas. An element at position (i, j) of the target array, corresponds to an element $(2i, j//2)$ (if j is even) or $(2i + 1, j//2)$ (if j is odd) of the source array, with here $0 \leqslant i < \frac{n}{2}$ and $0 \leqslant j < 2n$.

Example. Here is a 4×4 array, and the stretched 2×8 array. The rows 0 and 1 on the left give the row 0 on the right. The rows 2 and 3 on the left give the row 1 on the right.

$$
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
13 & 14 & 15 & 16
\end{array}
\qquad
\begin{array}{cccccccc}
1 & 5 & 2 & 6 & 3 & 7 & 4 & 8 \\
9 & 13 & 10 & 14 & 11 & 15 & 12 & 16
\end{array}
$$

- **Fold.** The principle is as follows: the right part of a stretched array is turned upside down, then added under the left part. Starting from a $\frac{n}{2} \times 2n$ array you get an $n \times n$ array.
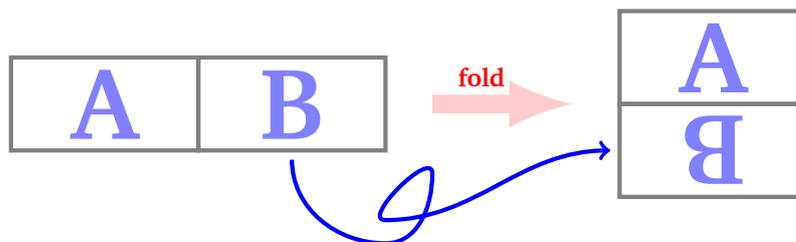
Formulas. For $0 \leqslant i < \frac{n}{2}$ and $0 \leqslant j < n$ the elements in position (i, j) of the array are kept in place. For $\frac{n}{2} \leqslant i < n$ and $0 \leqslant j < n$ an element of the array (i, j) corresponds to an element $\left(\frac{n}{2} - i - 1, 2n - 1 - j\right)$ of the source array.

Example. From the stretched 2×8 array on the left, we obtain a folded 4×4 array on the right.

$$
\begin{array}{cccccccc}
1 & 5 & 2 & 6 & 3 & 7 & 4 & 8 \\
9 & 13 & 10 & 14 & 11 & 15 & 12 & 16 \\
\end{array}
\qquad
\begin{array}{cccc}
1 & 5 & 2 & 6 \\
9 & 13 & 10 & 14 \\
16 & 12 & 15 & 11 \\
8 & 4 & 7 & 3 \\
\end{array}
$$

The *baker transformation* is the succession of stretching and folding, starting and ending with an $n \times n$ array.

Let's see an example of several baker transformations. On the left is the initial 128×128 image, then the result of $k = 1, 2, 3$ iterations.
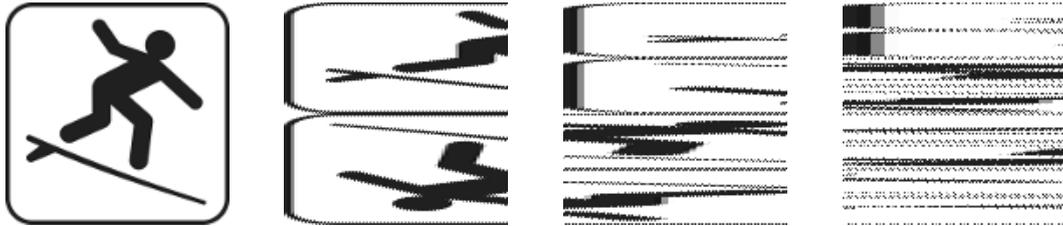
Here are the images for $k = 12, 13, 14, 15$ iterations:

Activity 3 (Baker's transformation).

Goal: program a new transformation that stretches and folds an image. Once again, the image becomes more and more distorted but, after a certain number of iterations, we return to the original image again.

1. Program a `baker_stretch(array)` function that returns the new array obtained by "stretching" the input table.
2. Program a `baker_fold(array)` function that returns the table obtained by "folding" the input table.
3. Program a `baker_iterate(array,k)` function that returns the table calculated after k iterations of baker's transformation.

   For example, the original 4×4 table is on the left, its image after a transformation ($k = 1$) and its image after a second transformation ($k = 2$) are also shown.

$$
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
13 & 14 & 15 & 16 \\
\end{array}
\qquad
\begin{array}{cccc}
1 & 5 & 2 & 6 \\
9 & 13 & 10 & 14 \\
16 & 12 & 15 & 11 \\
8 & 4 & 7 & 3 \\
\end{array}
\qquad
\begin{array}{cccc}
1 & 9 & 5 & 13 \\
16 & 8 & 12 & 4 \\
3 & 11 & 7 & 15 \\
14 & 6 & 10 & 2 \\
\end{array}
$$

4. Program a `baker_images(image_name,kmax)` function that calculates the images corresponding to baker's transformation, with iterations ranging from $k = 1$ to $k = k_{\max}$.
5. Experiment with different values of the size n to see after how many iterations we get back to the original image.

Caution! It sometimes takes many iterations to get back to the original image. For example when $n = 4$, we return to the starting image after $k = 5$ iterations; when $n = 256$ it takes $k = 17$. Conjecture a return value in the case where n is a power of 2. However, for $n = 10$, you need $k = 56\,920$ iterations!

Here is an example with an image of size 256×256, first the initial image, then one transformation ($k = 1$) and a second iteration ($k = 2$).
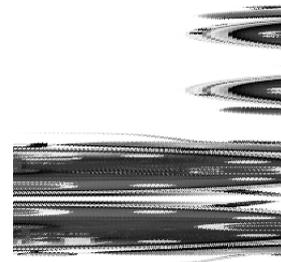
$k = 3, 4, 5$:
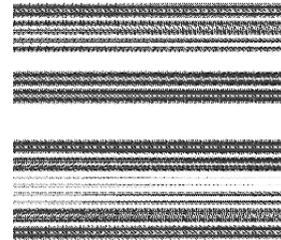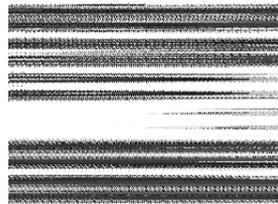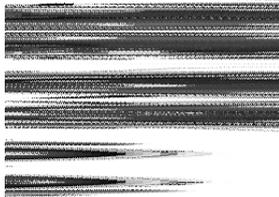
$k = 15, 16, 17$:
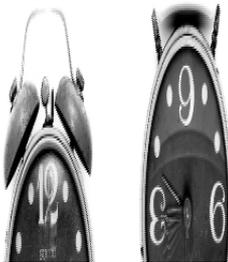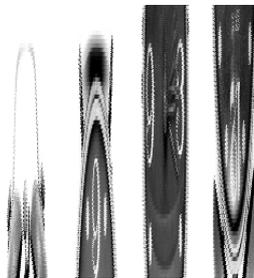
For $k = 17$ you get back to the original image!

This chapter is based on the article "Blurred images, recovered images" by Jean-Paul Delahaye and Philippe Mathieu (Pour la Science, 1997).

Game of life

The game of life *is a simple model of the evolution of a population of cells that split and die over time. The "game" consists of finding initial configurations that give interesting evolution: some groups of cells disappear, others stabilize, some move...*
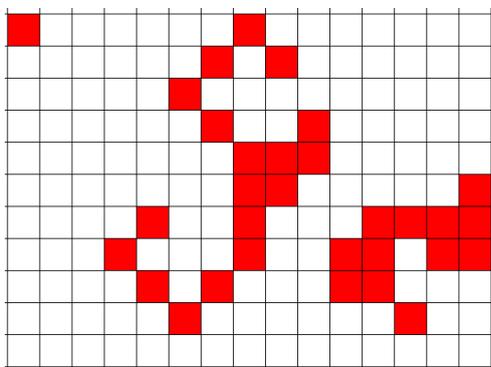
Lesson 1 (Rules of the game of life)**.**

The *game of life* takes place on a grid. Each square can contain one cell. Starting from an initial configuration, each day cells will be born and others will die depending on the number of its neighbors. Here are the rules:

- For an empty square on day j and having exactly 3 neighboring cells: a cell is born on day $j + 1$.

Has 3 neighbors   Birth

Initial position (day j)   **Number of neighbors**   **Evolution (day $j + 1$)**

Birth
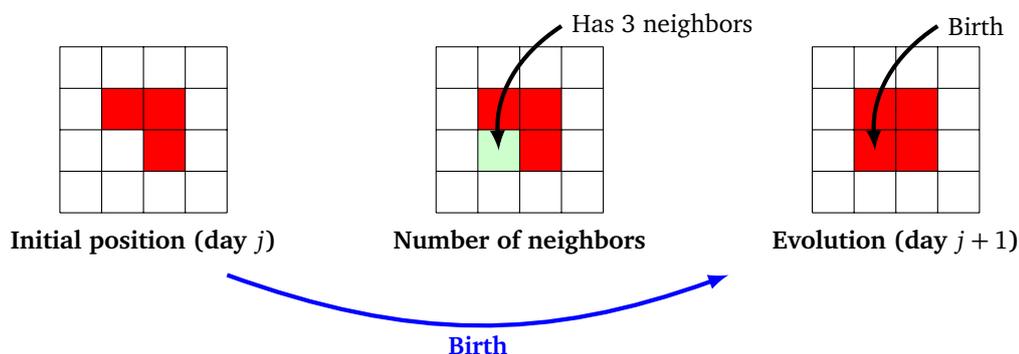
- If a square containing a cell at day j, has either 2 or 3 neighboring cells: then the cell continues to live. In other cases the cell dies (with 0 or 1 neighbors, it dies of isolation, with more than 4 neighbors, it dies of overpopulation).

Initial position (day j)   **Numbers of neighbors**   **Evolution (day $j+1$)**

Three deaths (and two births)
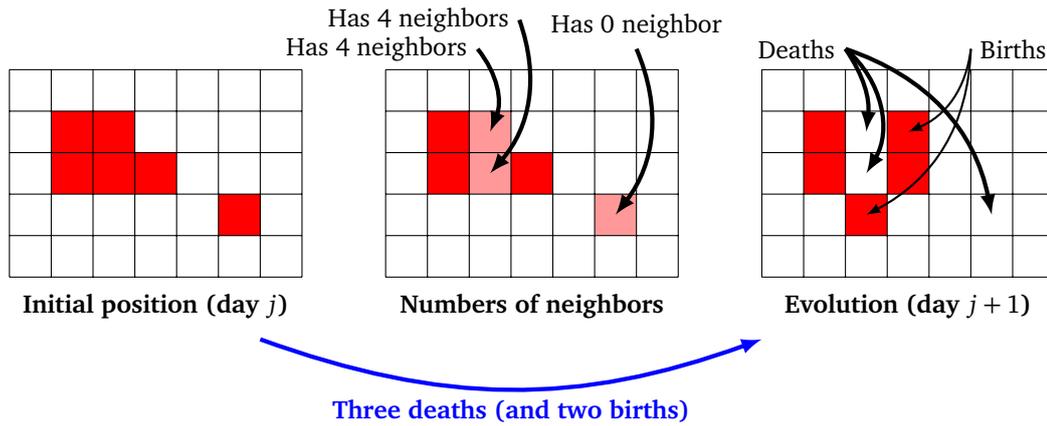
Here is a simple example, the "blinker".

Initial position (day j)   **Numbers of neighbors**   **Evolution (day $j+1$)**
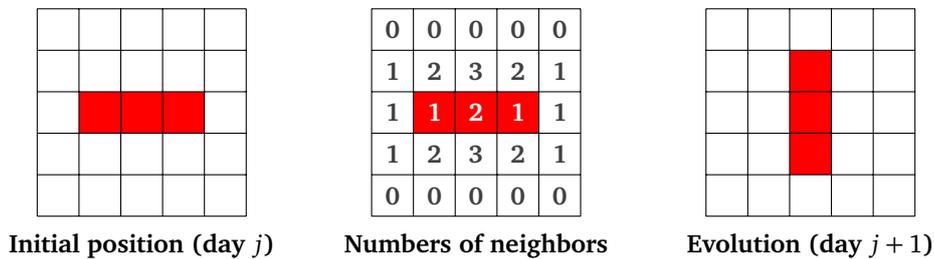
Here is another example (with the number of neighbors written in each case), the first evolution and the evolution of the evolution.

Day 0   **Day 1**   **Day 2**
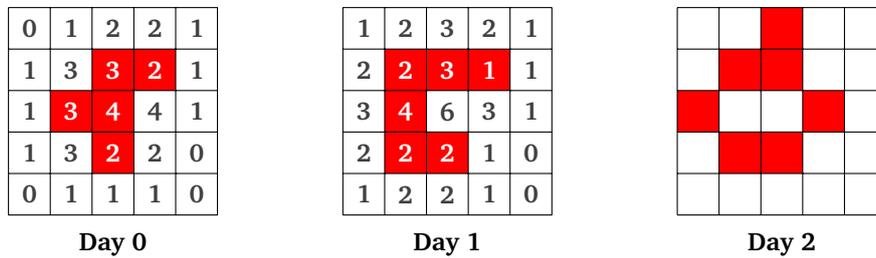
Activity 1 (Display).

Goal: define and display arrays.

We model the living space of the cells by a double entry table, containing integers, 1 to indicate the presence of a cell, 0 otherwise. Here is an example of the "toad" configuration and its array:

Cells   **Array**
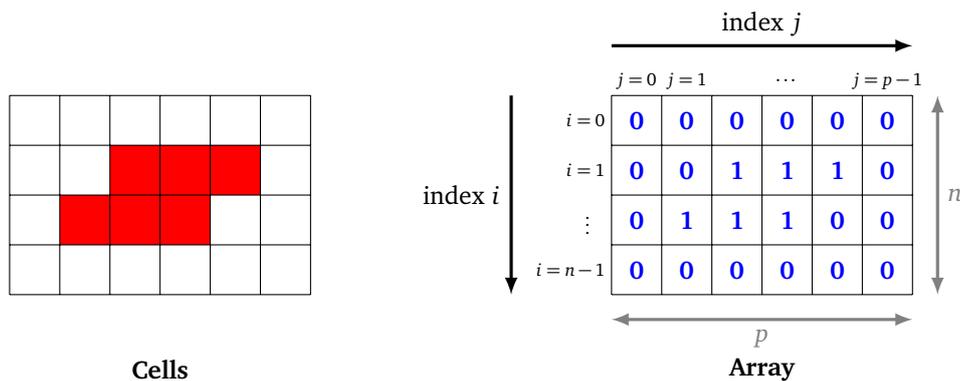
1.
   - Initialize two variables: n (the height of the table) and p (the width) (for example at 5 and 8).
   - Define a two-dimensional table filled with zeros by the command:
     ```
     array = [[0 for j in range(p)] for i in range(n)]
     ```
   - By using instructions of the type `array[i][j] = 1`, fill in the table to define the configurations of the blinker, the toad …

2. Program the display of a given array on the screen. For example, the blinker is displayed as follows:

```
00000000
00000000
00111000
00000000
00000000
```

Hint. By default the `print()` function starts a new line on each call (it effectively adds `"\n"` which is the end of line character). It can be specified not to do this by the option `print("My text",end="")`.

Activity 2 (Graphic display).

Goal: create a graphical display of a cell configuration.
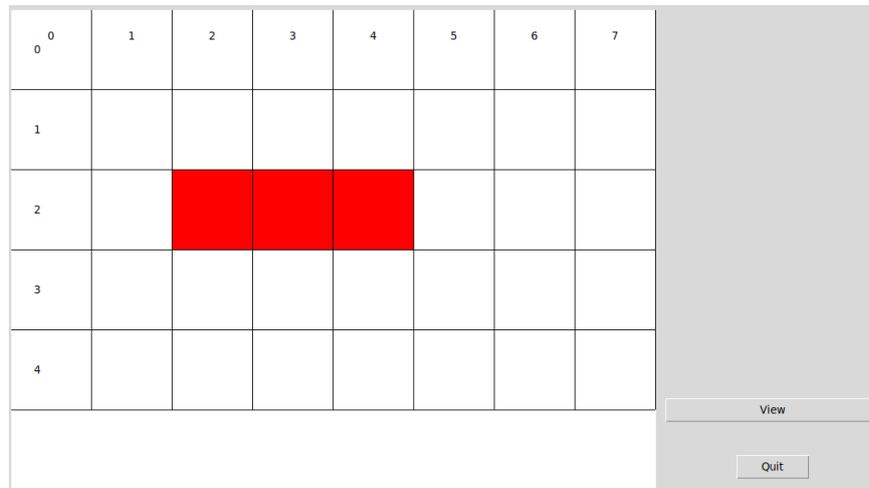
1. Program the creation of a `tkinter` window (see the "Statistics – Data visualization" chapter). Write a `draw_grid()` function, without parameters, that displays a blank grid of coordinates. You will need a constant `scale` (for example `scale = 50`) to transform the indices i and j into graphic coordinates x and y, depending on the relationship: $x = s \times j$ and $y = s \times i$ (s being the scale).
   Bonus. You can also mark the values of the indices i and j at the top and the left to make it easier to read the grid.

2. Build a `draw_array(array)` function that graphically displays the cells from a table.
   Bonus. Add a "View" button and a "Quit" button (see the "Statistics – Data visualization" chapter).

Activity 3 (Evolution).

Goal: calculate the evolution of a configuration from one day to the next.

1. Program a `number_neighbors(i,j,array)` function that calculates the number of living neighboring cells to the cell (i, j).
   Hints.
   - There is a maximum of 8 possible neighbors. The easiest way is to test them one by one!
   - For counting, it is necessary to be very careful with the cells that are placed near an edge (and have less than 8 possible neighbors).

2. Program an `evolution(array)` function that receives a table as an input and returns a new array corresponding to the situation on the next day, according to the rules of the game of life described at the beginning. For example, if the table on the left corresponds to the input, then the output corresponds to the table on the right:

00000000		00000000
00000000		00010000
00111000	evolves in	00010000
00000000		00010000
00000000		00000000

Hints. To define a new table, use the command:

```
new_array = [[0 for j in range(p)] for i in range(n)]
```

and then modify the table as desired.

Activity 4 (Iterations).

Goal: complete the graphic program so that the user can define configurations and make them evolve with a simple click.

1. Improve the graphics window to make the user's life easier:
   - An "evolve" button which displays the next evolution with each click.
   - Buttons to display predefined configurations (in the screenshot below is the configuration "pentadecathlon").
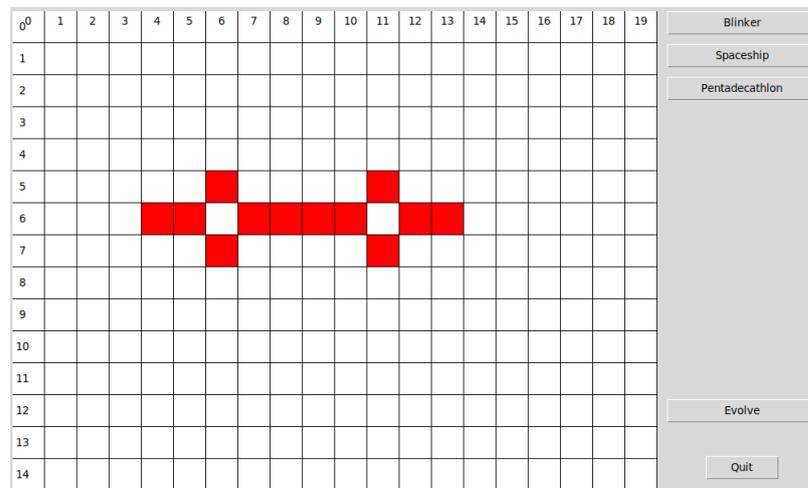
2. Perfect your program so that the user can draw the configuration he wants with mouse clicks. Clicking on an extinguished cell turns it on, clicking on an living cell turns it off. You can break this task down into three functions:

- on_off(i,j), which switches the cell (i, j).
- xy_to_ij(x,y) that converts graphic coordinates (x, y) into integers coordinates (i, j) (use the scale variable and the integer division).
- action_click_mouse(event) to retrieve the coordinates (x, y) with a mouse click (see course below) and switch the clicked box.

Lesson 2 (Mouse click).

Here is a small program that displays a graphic window. Each time the user clicks (with the left mouse button) the program displays a small square (on the window) and displays "Click at $x = \ldots$, $y = \ldots$" (on the console).

```python
from tkinter import *

# Window
root = Tk()
canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(side=LEFT, padx=5, pady=5)

# Catch mouse click
def action_mouse_click(event):
    canvas.focus_set()
    x = event.x
    y = event.y
    canvas.create_rectangle(x,y,x+10,y+10,fill="red")
    print("Click at x =",x,", y =",y)
    return

# Association click/action
canvas.bind("<Button-1>", action_mouse_click)

# Launch
root.mainloop()
```

Here are some explanations:

- The creation of the window is usual. The program ends with the launch of the `mainloop()` command.

- The first key point is to associate a mouse click to an action, that's what this line does:

  `canvas.bind("<Button-1>", action_mouse_click)`

  Each time the left mouse button is clicked, `Python` executes the function `action_mouse_click`. (Note that there are no brackets for the call to the function.)

- Second key point: the `action_mouse_click` function retrieves the click coordinates and then does two things here: it displays a small rectangle at the click location and prints the (x, y) coordinates to the terminal window.

- The coordinates x and y are expressed in pixels; $(0, 0)$ refers to the upper left corner of the window (the area delimited by `canvas`).

Here are some ideas to go further:
- Make sure that the grid automatically adapts to the screen width, i.e. calculate `scale` based on `n` and `p`.
- Add an intermediate step before evolving: color a cell that will be born in green and a cell that will die in black. For this purpose the elements of `array` can take values other than 0 and 1.

Here are some interesting configurations.

Square   **Spaceship**   **Pulsar**
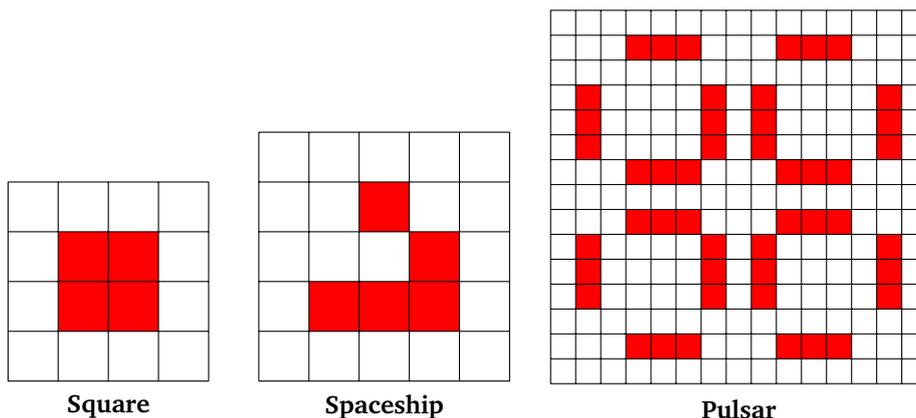
You will find many more on the Internet but above all have fun discovering new ones!

In particular, find configurations:
- which remain fixed over time;
- which evolve, then become fixed;
- which are periodic (the same configurations come back in loops) with a period of 2, 3 or more;
- which travel;
- which propel cells;
- whose population is increasing indefinitely!

Ramsey graphs and combinatorics

You will see that a very simple problem, which concerns the relationships between only six people, will require a lot of calculations to be solved.
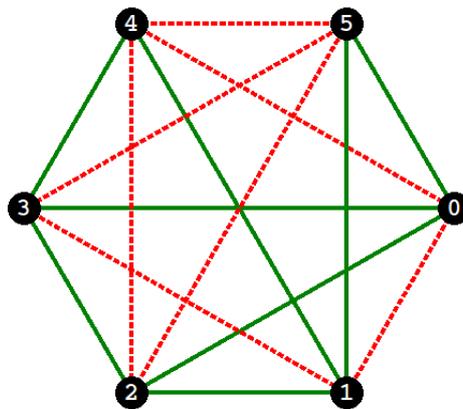
Lesson 1 (Ramsey's problem).

Proposition. In a group of 6 people, there is always 3 friends (the three know each other) or 3 strangers (all three are strangers to each other).

The purpose of this chapter is to have the computer demonstrate this statement. For that, we will model the problem by graphs, then we will check the statement for each of the 32 768 possible graphs.

We consider n people. For two of them, either they know each other (they are friends) or they do not know each other (they are strangers to each other). We represent this using a graph:

- a person is represented by a vertex (numbered from 0 to $n-1$);
- if two people are friends, the corresponding vertices are connected by a green solid edge;
- otherwise (they are strangers), the corresponding vertices are connected by a red dotted edge.

The graph below means that 0 is friend with 2; 1 is friend with 3. The other pairs don't know each other.
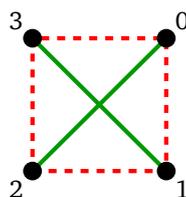
A graph checks the Ramsey problem, if there are among its vertices, either 3 friends, or either 3 strangers.

Three friends
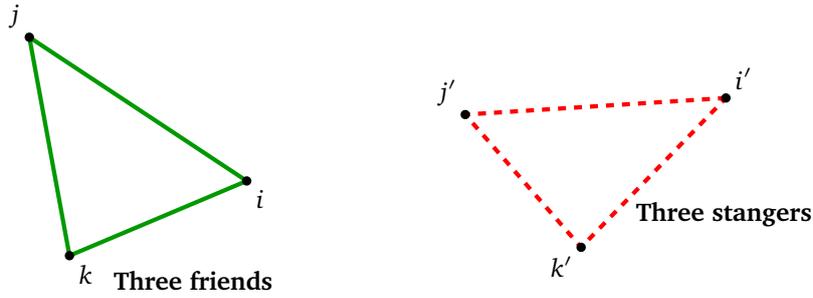
Three stangers

Here is an example of 5 peoples that verify the statement: there is group of 3 strangers (the 0, 2 and 4 vertices), even if there is no group of three friends.

A graph with $n = 5$ that satifies Ramsey's statement
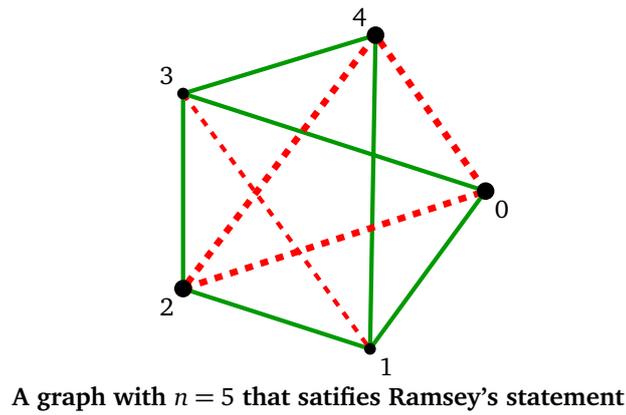
Lesson 2 (Model).

We model a graph using an array, containing 0's and 1's.

Let be a graph with n vertices, numbered from 0 to $n-1$. The *array of the graph* is a table of size $n \times n$ in which we place a 1 in position (i, j) if the vertices i and j are connected by an edge, otherwise we place a 0.

First example below: the vertices 0 and 2 are friends (because they are connected by a green edge) so the table contains a 1 in position $(0, 2)$ and also in $(2, 0)$. Similarly 1 and 3 are friends, so the table contains a 1 in positions $(1, 3)$ and $(3, 1)$. The rest of the table contains 0.

index j

	$j=0$	$j=1$	$j=2$	$j=3$
$i=0$	0	0	1	0
$i=1$	0	0	0	1
$i=2$	1	0	0	0
$i=3$	0	1	0	0

index i

n

n
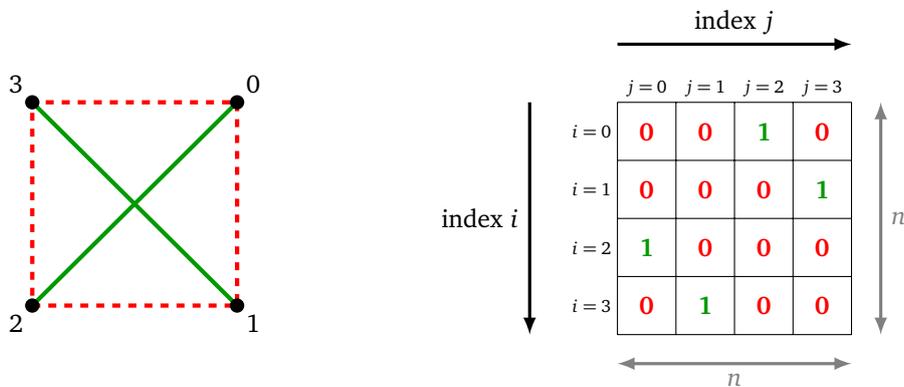
Here is a more complicated graph and its array:
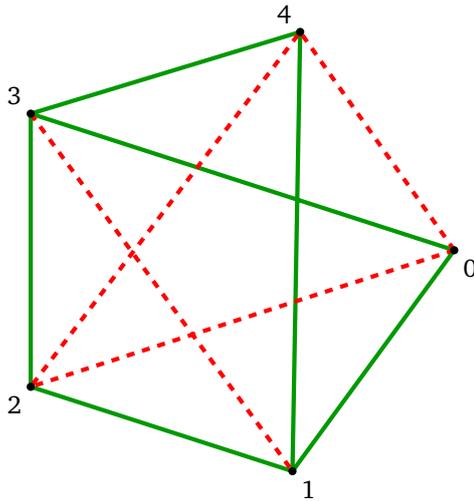
	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$
$i=0$	0	1	0	1	0
$i=1$	1	0	1	0	1
$i=2$	0	1	0	1	0
$i=3$	1	0	1	0	1
$i=4$	0	1	0	1	0

Activity 1 (Build graphs).

Goal: define graphs and test if three given vertices are friends.
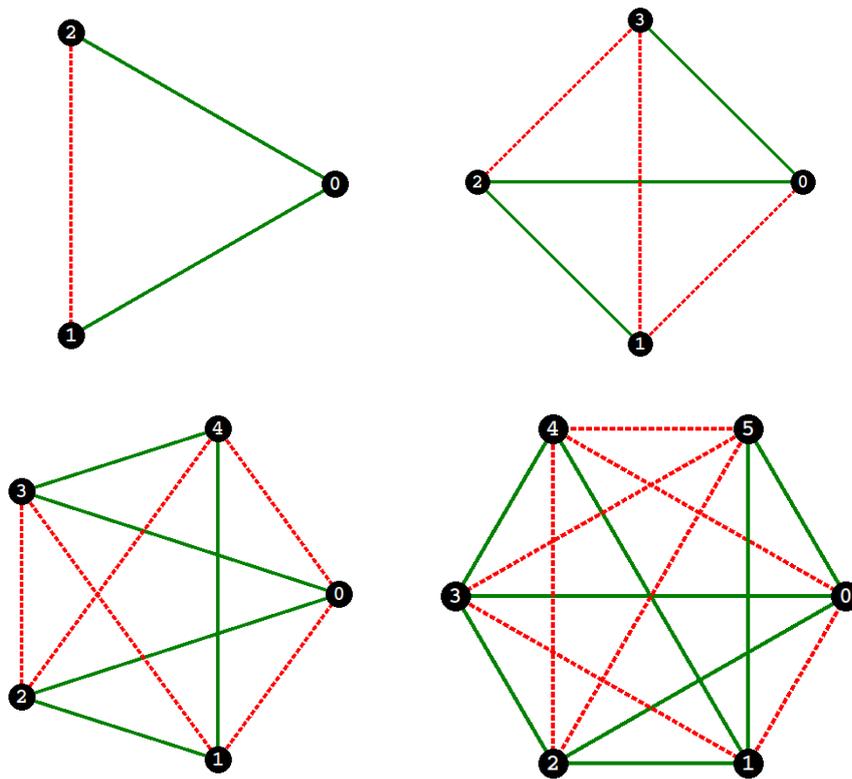
1. Define the graph array for the four examples above. You can start by initializing the array with

   ```
   array = [[0 for j in range(n)] for i in range(n)]
   ```

   Then add commands:

   ```
   array[i][j] = 1   and   array[j][i] = 1
   ```

   Don't forget that if vertex i is connected to the vertex j by an edge, then you have to put a 1 in position (i, j) but also in position (j, i).

2. Define a `print_graph(array)` function that allows you to display the table of a graph on the screen. Thus the third example above (with $n = 5$) should be displayed as follows:

```
0 0 1 1 0
0 0 1 0 1
1 1 0 0 0
1 0 0 0 1
0 1 0 1 0
```

3. We set the three vertices i, j, k of a graph. Write a `have_3_fix_friends(array,i,j,k)` function that tests if the vertices i, j, k are three friends (the function returns `True` or `False`). Do the same thing for a `have_3_fix_strangers(array,i,j,k)` function to find out if these vertices are strangers.

   Find on the fourth example (without computer), three friend or foreign vertices and check your answer using the functions you have just defined on these vertices.

Activity 2 (Draw nice graphs).

   Goal: draw a graph! Optional activity.
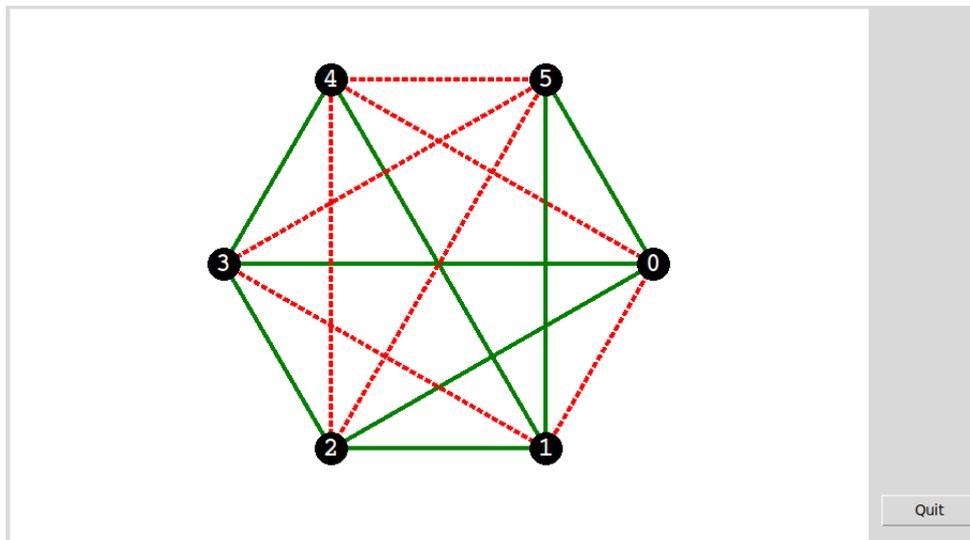
Program the graphical display of a graph by a `display_graph(array)` function.

Hints. This activity is not necessary for the next steps, it just helps to visualize the graphs. It is necessary to use the `tkinter` module and the `create_line()`, `create_oval()` and possibly `create_text()` functions.

The most delicate point is to obtain the coordinates of the vertices. You will need the sine and cosine functions (available in the `math` module). The coordinates (x_i, y_i) of the vertex number i of a graph with n elements can be calculated by using the formulas:

$$x_i = r \cos\left(\frac{2i\pi}{n}\right) \quad \text{and} \quad y_i = r \sin\left(\frac{2i\pi}{n}\right).$$

These vertices are located on the circle of radius r, centered at $(0, 0)$. You will have to choose a large enough r (for example $r = 200$) and shift the circle to display it on the screen.
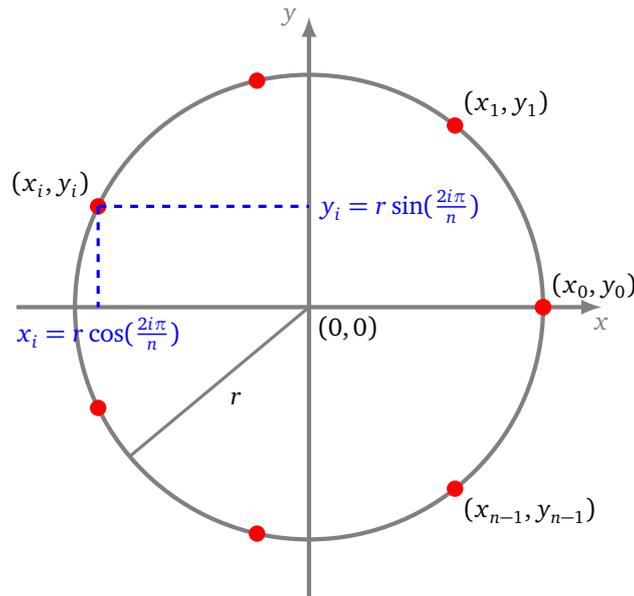
Activity 3 (Binary notation with zero-padding).

Goal: convert an integer to binary notation with possible leading zeros.

Program an `integer_to_binary(p,n)` function that displays an integer p in binary notation using n bits. The result is a list of 0's and 1's.

Example.
- The binary notation of $p = 37$ is 1.0.0.1.0.1. If you want its binary notation using $n = 8$ bits you have to add two 0's in front of it: 0.0.1.0.0.1.0.1.
- So the result of the command `integer_to_binary(37,8)` must be `[0, 0, 1, 0, 0, 1, 0, 1]`.
- The command `integer_to_binary(37,10)` returns 37 in binary notation using 10 bits: `[0, 0, 0, 0, 1, 0, 0, 1, 0, 1]`.

Hints.
- You can use the `bin(p)` command.
- The `list(mystring)` command returns the list of characters making up `mystring`.
- Attention! We want a list of integers 0 or 1, not of characters `'0'` or `'1'`. The command `int('0')` returns 0 and `int('1')` returns 1.
- `mylist = mylist + [element]` adds an item at the end of the list, while `mylist = [element] + mylist` adds the item at the beginning of the list.

Lesson 3 (Subsets).
Let $E_n = \{0, 1, 2, \ldots, n-1\}$ be the set of all integers from 0 to $n-1$. The set E_n therefore contains n elements. For example $E_3 = \{0, 1, 2\}$, $E_4 = \{0, 1, 2, 3\}$...

Subsets.
What are the subsets of E_n? For example there are 8 subsets of E_3, these are:
- the subset $\{0\}$ composed of the single element 0;

- the subset {1} composed of the single element 1;
- the subset {2} composed of the single element 2;
- the subset {0, 1} composed of the element 0 and the element 1;
- the subset {0, 2};
- the subset {1, 2};
- the subset {0, 1, 2} composed of all elements;
- and the empty set ∅, which contains no elements!

Proposition. The set E_n contains 2^n subsets.

For example $E_4 = \{0, 1, 2, 3\}$ has $2^4 = 16$ possible subsets. Have fun finding them all! For E_6 there are $2^6 = 64$ possible subsets.

Subsets of fixed cardinal.
We are only looking for subsets with a fixed number k of elements.
Examples:
- For $n = 3$ and $k = 2$, the subsets having two elements and contained in $E_3 = \{0, 1, 2\}$ are the three pairs: $\{0, 1\}, \{0, 2\}, \{1, 2\}$.
- For $n = 5$ and $k = 3$, the subsets having three elements and contained in $E_5 = \{0, 1, 2, 3, 4\}$ are the 10 triplets: $\{0, 1, 2\}, \{0, 1, 3\}, \{0, 2, 3\}, \{1, 2, 3\}, \{0, 1, 4\}, \{0, 2, 4\}, \{1, 2, 4\}, \{0, 3, 4\}, \{1, 3, 4\}, \{2, 3, 4\}$.

Activity 4 (Subsets).

Goal: generate all subsets in order to test all triplets of vertices. For this we will use binary notation.

Here is how we associate a subset of $E_n = \{0, 1, \ldots, n-1\}$ to each integer p verifying $0 \leqslant p < 2^n$.
Let's start with the example $n = 6$ and $p = 26$:
- the binary notation of $p = 26$ using $n = 6$ bits is $[0,1,1,0,1,0]$;
- there are 1's at indices 1, 2 and 4 (starting at rank 0 on the left);
- the associated subset is then $\{1, 2, 4\}$.
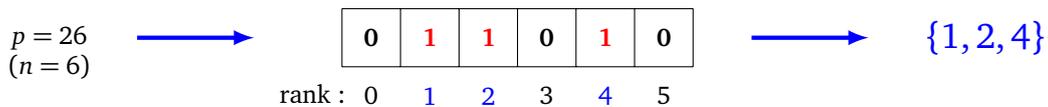
$p = 26$ ($n = 6$)	0 1 1 0 1 0 (rank: 0 1 2 3 4 5)	$\{1, 2, 4\}$
Initial integer p	**Binary notation of p using n bits**	**Subset associated to p**

Other examples.
- With $n = 8$ and $p = 57$ whose binary notation using 8 bits is $[0,0,1,1,1,0,0,1]$, the associated subset corresponds to the indices $2, 3, 4, 7$, so the associated subset is $\{2, 3, 4, 7\}$.
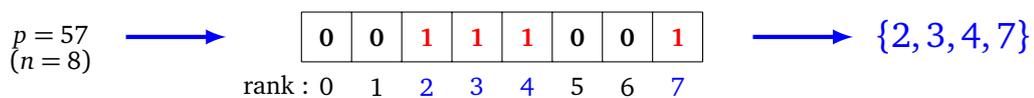
$p = 57$ ($n = 8$)	0 0 1 1 1 0 0 1 (rank: 0 1 2 3 4 5 6 7)	$\{2, 3, 4, 7\}$
Initial integer p	**Binary notation of p using n bits**	**Subset associated to p**

- When $p = 0$, the binary notation is only formed of 0's, the associated subset is the empty set.

- When $p = 2^n - 1$, the binary notation is full of 1's, the associated subset is the whole set $E_n = \{0, 1, \ldots, n-1\}$ itself.

We model a set as a list of elements. For example:
- For us the set E_4 is the list `[0,1,2,3]`.

- `[1,3]` is an example of a subset of E_4.

- The empty set is represented by the empty list `[]`.

1. Program a `subsets(n)` function which returns the list of all the possible subsets of $E_n = \{0, 1, 2, \ldots, n-1\}$. For example, when $n = 3$, `subsets(n)` returns the list (which itself contains lists):

    `[ [], [2], [1], [1, 2], [0], [0, 2], [0, 1], [0, 1, 2] ]`

    That is to say the 8 subsets (starting with the empty set):

    $$\varnothing \quad \{2\} \quad \{1\} \quad \{1,2\} \quad \{0\} \quad \{0,2\} \quad \{0,1\} \quad \{0,1,2\}.$$

    Hint. To test your program, check that the returned list contains 2^n subsets.

2. Derive from it a `fix_subsets(n,k)` function that returns only the subsets of E_n that have exactly k elements.

    For example, for $n = 3$ and $k = 2$, `fix_subsets(n,k)` returns the list of pairs:

    `[ [0, 1], [0, 2], [1, 2] ]`

    Test your program:
    - For $n = 4$ and $k = 3$, the list returned by `fix_subsets(n,k)` contains 4 triplets.

    - For $n = 5$ and $k = 3$, there are 10 possible triplets.

    - For $n = 10$ and $k = 4$, there are 210 possible subsets!

From now on we will mostly use subsets with 3 elements. In particular, for $n = 6$, there are 20 triplets included in $\{0, 1, 2, 3, 4, 5\}$:

```
[[3, 4, 5], [2, 4, 5], [2, 3, 5], [2, 3, 4], [1, 4, 5],
[1, 3, 5], [1, 3, 4], [1, 2, 5], [1, 2, 4], [1, 2, 3],
[0, 4, 5], [0, 3, 5], [0, 3, 4], [0, 2, 5], [0, 2, 4],
[0, 2, 3], [0, 1, 5], [0, 1, 4], [0, 1, 3], [0, 1, 2]]
```

Activity 5 (Ramsey's theorem for $n = 6$).

Goal: check that all graphs with 6 vertices contain three friends or three strangers.

1. Program a `have_3(array)` function that tests if a graph contains 3 friends or 3 strangers. You must therefore call the functions `have_3_fix_friends(array,i,j,k)` and `have_3_fix_strangers(array,i,j,k)` from the first activity on all possible triplets of vertices (i, j, k).

    For the four examples from the first activity, only the fourth (with 6 summits) passes the test.

2. Program an `all_graphs(n)` function that computes all the possible graph arrays with n vertices. There are $N = \frac{(n-1)n}{2}$ possible arrays. You can generate them by using a method similar to the one for subsets:
    - for each integer p that satisfies $0 \leqslant p < 2^N$,

    - calculate the binary notation of p using N bits,

    - fill the array in element by element, with the 0's and 1's of the binary notation.

    Hints. To write a function to fill an array from a given binary notation of p named `binary_notation` (that is to say a list of 0's and 1's), you can use a double loop like:

```
for j in range(0,n):
    for i in range(j+1,n):
        b = binary_notation.pop()
        array[i][j] = b
        array[j][i] = b
```

Here is the principle of the loop that fills in the part above the diagonal in the grid (and also the part below it by symmetry). This loop takes the last bit of the list and places it in the first empty box above the diagonal; then the second last bit is placed in the second empty box...; the first bit of the list fills the last empty square.

Binary notation of p with N bits

Filling of an array
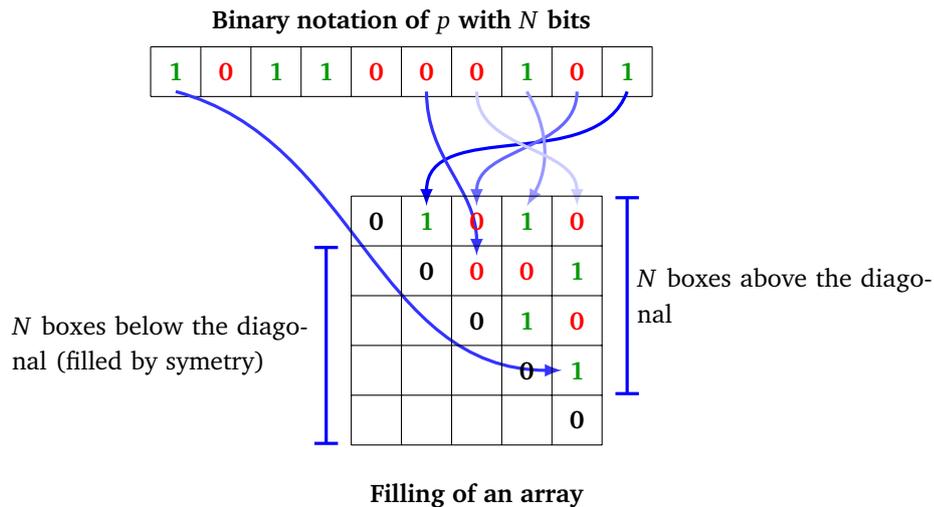
3. Convert the previous function into a `test_all_graphs(n)` function which tests the conjecture "there are three friends or three strangers" for all graphs with n vertices. You must find that:
   - for $n = 4$ and $n = 5$ the conjecture is false. Give a graph having 4 vertices (then 5 vertices) that has neither 3 friends, nor 3 strangers;
   - for $n = 6$ let the computer check that, for each of the $N = 2^{\frac{5 \times 6}{2}} = 32\,768$ graphs with 6 vertices, either it has 3 friends or it has 3 strangers.

Activity 6 (To go further).

   Goal: improve your program and prove other conjectures. Optional activity.

1. Improve your program so that it checks the conjecture for $n = 6$ in less than a second.
   Ideas.
   - The list of triplets must only be generated once at the beginning of the program (and not for each new graph).
   - It is not necessary to generate a list of all possible graphs, then test them in a second step. It is better to generate one and then test it before moving on to the next.
   - As soon as you find 3 friends (or 3 strangers) it's a win! Immediately stop the loop, even if it means using the `break` instruction, and move on to the next graph.
   - You can only test the graphs that correspond to p between 0 and $2^N/2$ (because for the rest of the p's it is like exchanging the green segments for red and vice versa).

With these tips here are the calculation times you can expect:

Number of vertices	Number of graphs	Approximate calculation time
$n = 6$	32 768	< 1 second
$n = 7$	2 097 152	< 1 minute
$n = 8$	268 435 456	< 1 hour
$n = 9$	68 719 476 476 736	< 10 days

2. Here is a more difficult task. It is a question of finding out at what size n, a graph will always contain either 4 friends or 3 strangers. Being 4 friends means every pair is connected by a green segment, as below:
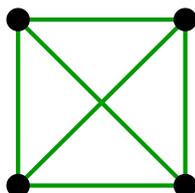
   (a) Find graphs with $n = 6$ (then $n = 7$) vertices that do not satisfy this statement.
   (b) By searching a little with your computer find graphs with 8 vertices that do not satisfy this statement.
   (c) Prove that any graph that has 9 vertices contains 4 friends or 3 strangers!
       Hints. It is necessary to test all the graphs corresponding to the integer p between 0 and $2^N = 2^{\frac{8 \times 9}{2}} = 68\,719\,476\,736$. The total calculation time is about 20 days! You can split the calculations between several computers: one computer does the calculations for $0 \leqslant p \leqslant 1\,000\,000$, a second computer for $1\,000\,001 \leqslant p \leqslant 2\,000\,000,\dots$

3. • There are sets of arguments that prove with pencil and paper that for $n = 6$ there are always 3 friends or 3 strangers. Look for this kind of reasoning! With a little more effort, we can also prove that $n = 9$ answers the 4 friends/3 strangers problem.

   • We know how to prove that we need $n = 18$ to always have 4 friends or 4 strangers.

   • However, no one in the world knows what the smallest n is for the 5 friends/5 strangers problem!

Bitcoin

The bitcoin is a dematerialized and decentralized currency. It is based on two computer principles: public key cryptography and proof of work. To understand this second principle, you will create a simple model of bitcoin.

Activity 1 (Proof of work).

Goal: understand what "proof of work" means by studying a simple model. This activity is independent from the rest of the chapter. The idea is to find a problem that is difficult to solve but easy to check. Like sudokus for example: it only takes ten seconds to check that a grid is filled correctly, but it takes more than ten minutes to solve it.

The mathematical problem is as follows: you are given a prime number p and an integer y; you must find an integer x such that:

$$x^2 = y \pmod{p}$$

In other words, x is a square root of y modulo p. Be careful, there is not always a solution for x.

Examples.

- For $p = 13$ and $y = 10$, a solution is $x = 6$: because $x^2 = 6^2 = 36$ and $36 = 2 \times 13 + 10$ so $x^2 = 10$ (mod 13).

- The solution is not necessarily unique. For example, check that $x = 7$ is also a solution.

- There is not always a solution, for example for $p = 13$ and $y = 5$, a solution does not exist.

- Exercise: for $p = 13$, by hand find two solutions, x, to the problem $x^2 = 9$ (mod 13); also find a solution, x, to the problem $x^2 = 3$ (mod 13).

The prime number p is fixed for the rest of the activity. For easy examples we will use $p = 101$, for medium examples $p = 15\,486\,869$ and for difficult examples $p = 2\,276\,856\,017$. The larger the integer p is, the more difficult it is to obtain proof of work.

1. **Verification (easy).** Write a `verification(x,y)` function that returns `True` if x is the solution to the problem $x^2 = y$ (mod p) and `False` otherwise.

   Check that $x = 6\,543\,210$ is a solution when $y = 8\,371\,779$ and $p = 15\,486\,869$. Determine the calculation time required for this verification. (See the explanations after this activity.)

2. **Search for a solution (difficult).** To find a solution x, there is really no other choice for us than to test all x starting with $x = 0$, $x = 1$... Program a `square_root(y)` function that returns the first solution x to the problem for a given y (or `None` if there is no solution).

   - For $p = 101$ and $y = 17$, find x such that $x^2 = y$ (mod p).

   - For $p = 15\,486\,869$ and $y = 8\,371\,779$, find the solution x of the first question. How long did the search take?

   - Find a solution for $p = 15\,486\,869$ and $y = 13\,017\,204$.

Conclusion: we found a problem that was difficult to solve, but for which it is easy to check that a

given solution is suitable. For higher values of p, the search for a solution, x, can be much too long and may not succeed. We'll see how we can adjust the difficulty of the problem.

3. Instead of looking for an exact solution to our problem $x^2 = y \pmod{p}$, which is equivalent to $x^2 - y \pmod{p} = 0$, we can look for an approximate solution, i.e. one that checks:

$$x^2 - y \pmod{p} \leqslant m.$$

For example if $m = 5$, then you can have (modulo p): $x^2 - y = 0$, $x^2 - y = 1$, $x^2 - y = 2,\dots$ or $x^2 - y = 5$. Of course there are now many possible solutions, x.

Program an `approximate_square_root(y,margin)` function that finds an approximate solution to our problem $x^2 = y \pmod{p}$.

How long does it take to find one solution to the problem when $p = 15\,486\,869$, $y = 8\,371\,779$ and $m = 20$? Choose a large prime number p and find a margin of error m so that finding a solution to the approximate problem requires about 30 to 60 seconds of calculations (for any y).

Here are some examples of prime numbers you can use for your tests:

$$101 \quad 1097 \quad 10651 \quad 100109 \quad 1300573 \quad 15486869$$

$$179426321 \quad 2276856017 \quad 32416187567$$

Lesson 1 (Stopwatch).

The `time` module allows you to measure the execution time and also to know the date and time (see also the `timeit` module explained in the "Arithmetic – While loop – I" chapter). Here is a small script to measure the calculation time of an instruction.

```
import time

start_chrono = time.time()

time.sleep(2)

end_chrono = time.time()

total_chrono = end_chrono - start_chrono
print("Execution time (in seconds):", total_chrono)
```

Explanations.

- The module is named `time`.

- The function that returns the time is also named `time()`, so we call it using `time.time()`!

- The `time()` function returns a number that we are not interested in. What interests us is the difference between two values of this function.

- `start_chrono = time.time()` is like starting a stopwatch.

- `end_chrono = time.time()` is like stopping it.

- `total_chrono = end_chrono - start_chrono` is the total time taken by our script in seconds.

- The `time.sleep(duration)` function pauses the program for a certain amount of seconds.

Lesson 2 (Bitcoin and blockchain).

Bitcoin is a dematerialized currency. Transactions are recorded in a ledger called *blockchain*. We will build a (very simplified) model of such an account book.

Imagine a group of friends who want to share the group's expenses in the simplest possible way. At the beginning everyone has 1000 bitcoins and everyone's expenses and income are recorded as they go along. At the end of the holidays, they will regularize the situation.

The list of expenses/revenues is noted in the account book, for example:

- "Amir spent 100 bitcoins"

- "Barbara received 45 bitcoins"

- etc.

Just look through the account book to see how much everyone has received or spent since the beginning.

To prevent someone from falsifying the account book, after each transaction a certification based on a "proof of work" is added to the book. Here is what is written in the ledger:

1. We start with any initial proof of work. For us it will be [0,0,0,0,0,0].
2. The first transaction is written (for example "Amir -100").
3. A proof of work is calculated and written in the book, which will serve as a certificate. It's a list (for example [56,42,10,98,2,34]) obtained after many calculations taking into account the previous transaction and the previous proof of work.
4. For each new transaction (for example "Barbara +45"), someone calculates a proof of work for the last transaction associated with the previous proof. We write the transaction, then the proof of work.
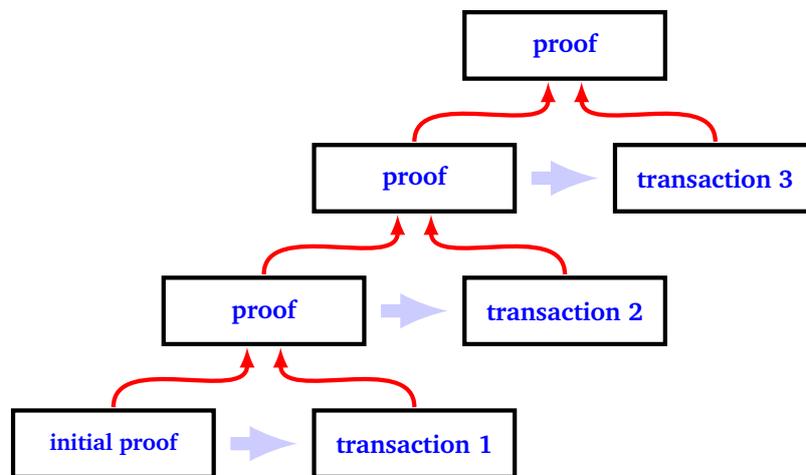
The proof of work that is calculated depends on the previous transaction but also on the previous proof of work, which itself depends on the previous data... Thus, each new proof of work actually depends on everything that has been written since the beginning (even if the calculation explicitly refers only to the last two entries).

Someone who would like to fake a past transaction should recalculate all the proofs of work that come next. This is not possible for a single person: there are several proofs of work to calculate and each proof requires a lot of calculations.

Activity 2 (Tools for lists).

Goal: build useful functions to manipulate lists of integers for the following activities.

Our lists are lists of integers between 0 and 99.

1. Program an `addition(mylist1,mylist2)` function that adds two lists of the same length, element by element, and applies modulo 100 to each term. For example, `addition([1,2,3,4,5,6],[1,1,1,98,98,98])` returns `[2,3,4,2,3,4]`.

2. A list `mylist` is smaller than the `max_list` list if each element of `mylist` is less than or equal to it's corresponding element of `max_list`. We will look for lists that start with zeros (or zeros and then rather small numbers).

   For example, the list `[0,0,1,2,3,4]` is smaller than the list `[0,0,5]`. This is not the case for the list `[0,10,0,1,1]`. Another example: being smaller than the list `[0,0,0]` means starting with three zeros. Being smaller than the list `[0,0,1]` means starting with `[0,0,0]` or `[0,0,1]`.

   Program an `is_smaller(mylist,max_list)` function that returns `True` when `mylist` is smaller than `max_list`.

3. We will need to transform a sentence into a list of numbers. In addition, we will split our lists into blocks of size N (with $N = 6$), we will add zeros at the beginning of the list so that its length is a multiple of N.

   Write a `sentence_to_list(sentence)` function that converts a string into a list of integers between 0 and 99 and if necessary adds leading zeros so that the list has the correct size.

   The formula to use to convert a character to an integer strictly less than 100 is:
   $$\text{ord(c) \% 100}$$
   For example: if `sentence = "Be happy!"` then the function returns:
   $$[0, 0, 0, 66, 1, 32, 4, 97, 12, 12, 21, 33]$$
   The character `"e"` has ASCII/unicode code 101 so, modulo 100, the number is 1. Note that the function adds three 0's at the beginning of the list to have a length that is a multiple of $N = 6$.

Activity 3 (Hash function).

> *Goal: create a hash function. From a long message we calculate a short fingerprint. It's hard to find two different messages with the same fingerprint.*

In this activity, our message is a list of integers (between 0 and 99) of any length that is a multiple of $N = 6$. We transform it into a list of length $N = 6$: its fingerprint, named *hash*. Here are some examples of what our hash function will do:

- the list `[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]` has a hash:
  $$[10, 0, 58, 28, 0, 90]$$
- the slightly different list `[1, 1, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]` has a hash:
  $$[25, 14, 29, 1, 19, 6]$$

The idea is to mix the numbers per block of $N = 6$ integers, then combine this block with the next and start again, until you get a single block.

1. **One round.**

   For a block $[b_0, b_1, b_2, b_3, b_4, b_5]$ of size $N = 6$, *doing a round* consists in making the following operations:

   (a) We add some integers:
   $$[b'_0, b'_1, b'_2, b'_3, b'_4, b'_5] = [b_0, b_1 + b_0, b_2, b_3 + b_2, b_4, b_5 + b_4]$$

   (b) We multiply these integers by prime numbers (in order $7, 11, 13, 17, 19, 23$) and add 1:
   $$[b''_0, b''_1, b''_2, b''_3, b''_4, b''_5] = [7 \times b'_0 + 1, 11 \times b'_1 + 1, 13 \times b'_2 + 1, 17 \times b'_3 + 1, 19 \times b'_4 + 1, 23 \times b'_5 + 1]$$

(c) We make a circular permutation (the last one goes first):
$$[b_0''', b_1''', b_2''', b_3''', b_4''', b_5'''] = [b_5'', b_0'', b_1'', b_2'', b_3'', b_4'']$$

(d) We reduce each integer modulo 100 to get integers between 0 and 99.

b_0	b_1	b_2	b_3	b_4	b_5

(a) additions

(b) multiplications

$\times 7 + 1$ $\times 11 + 1$ $\times 13 + 1$ $\times 17 + 1$ $\times 19 + 1$ $\times 23 + 1$

(c) circular permutation
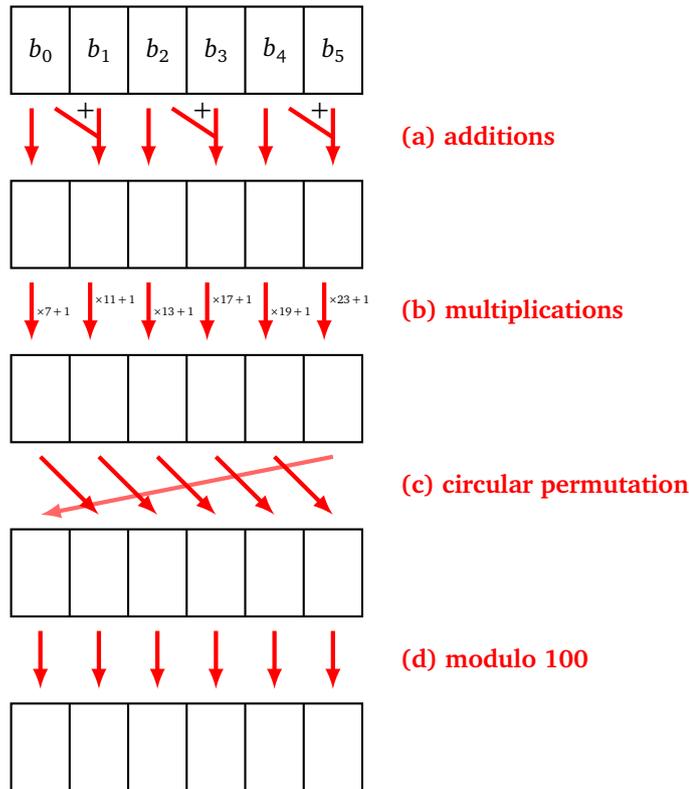
(d) modulo 100

Starting from the block $[0, 1, 2, 3, 4, 5]$, we have successively completed:

(a) additions: $[0, 1, 2, 5, 4, 9]$

(b) multiplications: $[7 \times 0 + 1, 11 \times 1 + 1, 13 \times 2 + 1, 17 \times 5 + 1, 19 \times 4 + 1, 23 \times 9 + 1] = [1, 12, 27, 86, 77, 208]$

(c) permutation: $[208, 1, 12, 27, 86, 77]$

(d) reduction modulo 100: $[8, 1, 12, 27, 86, 77]$

Program an `one_round(block)` function which returns the transformation of the block after these operations. Verify that the block $[1, 1, 2, 3, 4, 5]$ is transformed into $[8, 8, 23, 27, 86, 77]$.

2. **Ten rounds.**

To mix each block well, program a `ten_rounds(block)` function that iterates the previous operations ten times. After 10 rounds:

• the block $[0, 1, 2, 3, 4, 5]$ becomes $[98, 95, 86, 55, 66, 75]$,

• the block $[1, 1, 2, 3, 4, 5]$ becomes $[18, 74, 4, 42, 77, 42]$.

Two nearby blocks are transformed into two very different blocks!

3. **Hash of a list.**

Starting from a list whose length is a multiple of $N = 6$, it is split into blocks of length 6 and the hash of this list is calculated according to the following algorithm:

• The first block of the list is extracted, then mixed for 10 rounds.

• We add term to term (and modulo 100), the result of this mix to the second block.

• We start again from the new second block.

• When there is only one block left, we mix for 10 rounds, the result is the hash of the list.

Here is the diagram of a situation with three blocks: first there are three blocks (A, B, C); next there are only two blocks (B' and C) left; at the end there is only one block (C"): this is the hash!
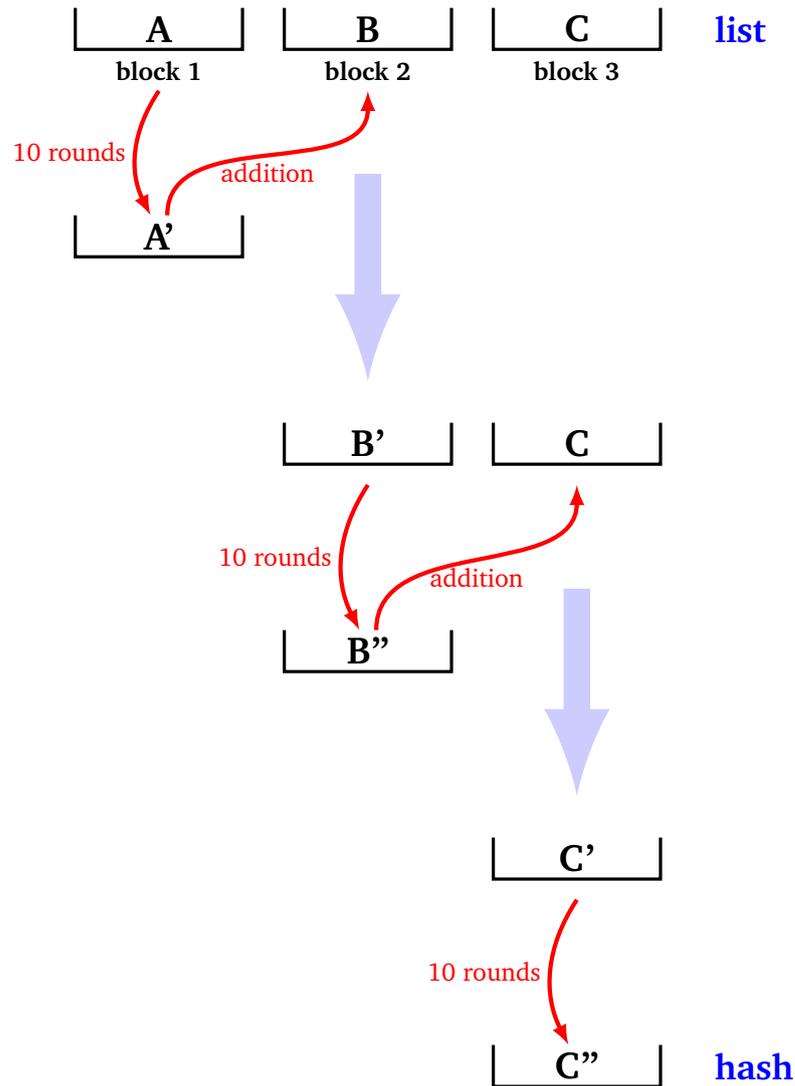
Example using the list $[0, 1, 2, 3, 4, 5, 1, 1, 1, 1, 1, 1, 10, 10, 10, 10, 10, 10]$.

- The first block is $[0, 1, 2, 3, 4, 5]$, its mix after 10 rounds is $[98, 95, 86, 55, 66, 75]$.
- This mix is added to the second block $[1, 1, 1, 1, 1, 1]$ (see the `addition()` function in activity 2).
- The remaining list is now $[99, 96, 87, 56, 67, 76, 10, 10, 10, 10, 10, 10]$.
- Let's do it again. The new first block is $[99, 96, 87, 56, 67, 76]$, its mix after 10 rounds is $[60, 82, 12, 94, 6, 80]$, it is added to the last block $[10, 10, 10, 10, 10, 10]$ to get (modulo 100) $[70, 92, 22, 4, 16, 90]$.
- A last mix is done for 10 rounds to obtain the hash: $[77, 91, 5, 91, 89, 99]$.

Program a `bithash(mylist)` function that returns the fingerprint of a list. Test it on the examples given at the beginning of the activity.

Activity 4 (Proof of work - Mining).

Goal: build a proof of work mechanism using our hash function.

We are going to build a complicated problem to solve, for which, if someone gives us the solution, then it is easy to check if it is correct.

Problem to solve. We are given a list, we need to find a block such that, when you add it to the list, it produces a hash starting with zeros. More precisely, given a list `mylist` and a maximum target of `max_list`, we have to find a block, `proof`, which, concatenated to the list and then hashed, is smaller than the list of `max_list`, i.e.:

$$\text{bithash(mylist + proof) smaller than max\_list}$$

The list is of any length (a multiple of $N = 6$), the proof is a block of length N, the objective is to find a list starting with 0's (see activity 2).

For example: let `mylist = [0,1,2,3,4,5]` and `max_list = [0,0,7]`. Which `proof` block can I concatenate to `mylist` to solve my problem?

- `proof = [12,3,24,72,47,77]` is appropriate because concatenated to our list it gives `[0,1,2,3,4,5,12,3,24,72,47,77]` and the hash of this whole list gives `[0,0,5,47,44,71]` which starts with `[0,0,5]` smaller than the target.

- `proof = [0,0,2,0,61,2]` is also suitable because after concatenation we have `[0,1,2,3,4,5,0,0,2,0,61,2]` whose hash gives `[0,0,3,12,58,92]`.

- `[97,49,93,87,89,47]` is not suitable, because after concatenation and hashing we get `[0,0,8,28,6,60]` which is greater than the desired objective.

1. **Verification (easy).**
   Program a `verification_proof_of_work(mylist,proof)` function that returns true if the proposed solution `proof` is suitable for `mylist`. Use the `is_smaller()` function from activity 2.

2. **Search for a solution (difficult).**
   Program a `proof_of_work(mylist)` function that looks for a `proof` block that is a solution to our problem for the given list.
   Hints.
   - The easiest method is to take a `proof` block of random numbers and start again until a solution is found.

   - You can also systematically test all blocks starting with `[0,0,0,0,0,0]`, then `[0,0,0,0,0,1]`... and stop at the first appropriate one.

   - You adjust the difficulty of the problem by changing the objective: easy with `max_list = [0,0,50]`, medium with `max_list = [0,0,5]`, difficult with `max_list = [0,0,0]`, very difficult with `max_list = [0,0,0,0]`.

   - As there are several solutions, you do not necessarily get the same solution for each search.

3. **Calculation time.**
   Compare the calculation time of a simple check against the time of searching for a solution. Choose the `max_list` objective so that the search for a proof of work requires about 30 to 60 seconds of calculations.

For the bitcoin, those who calculate proofs of work are called the *miners*. The first one to find a proof wins a reward. The difficulty of the problem is adjusted so that that the calculation time taken by the winner (among all the miners) to find a solution is about 10 minutes.

Activity 5 (Your bitcoins)**.**

Goal: create an account book (called blockchain *for bitcoin) that records all transactions, this ledger is public and certified. It is practically impossible to falsify a transaction that has already been registered.*
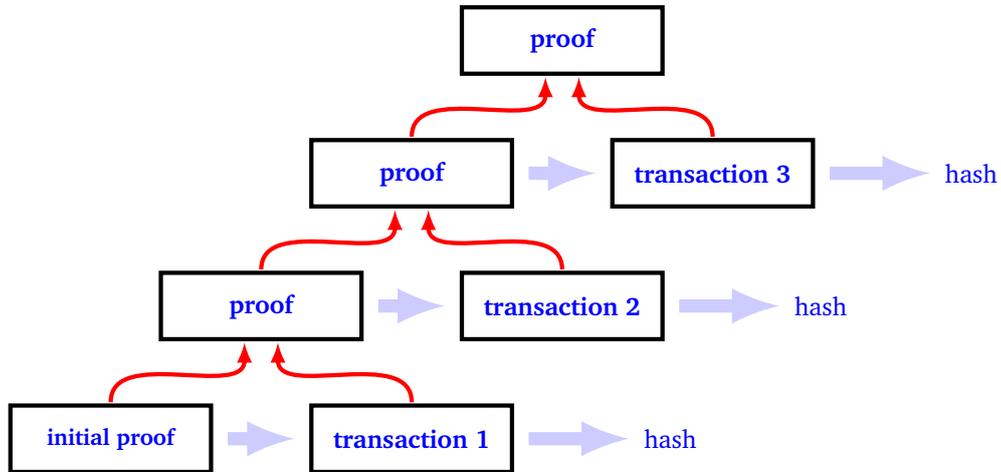
1. **Initialization and addition of a transaction.**
   (a) Initialize a global `blockchain` variable which is a list and contains at the beginning a zero proof: `blockchain = [ [0,0,0,0,0,0] ]`.
   (b) A *transaction* is a string including a name and the amount to add (or deduct) to your account. For example `"Abel +25"` or `"Barbara -45"`.
   Program an `add_transaction(transaction)` function that adds the string `transaction` to the list `blockchain`. For example after initialization `add_transaction("Camille +100")`, `blockchain` is `[ [0,0,0,0,0,0], "Camille +100" ]`. Careful, to be able to modify `blockchain` you must start the function with: `global blockchain`.
2. As soon as a transaction is added, a proof of work must be calculated and added to the account book. Program a `mining()` function, without parameters, that adds a proof of work to the book. Here's how to do it:
   - We take the last transaction `transaction`, we transform it into a list of integers by the `sentence_to_list()` function from activity 2.
   - We also need `prev_proof`, the previous proof of work just before this transaction.
   - We form the `mylist` list composed, first of elements of `prev_proof`, then of elements of the list of integers obtained by converting the string `transaction`.
   - A proof of work is calculated from this list.
   - This proof is added to the account book.
   For example, if the book ends with:
   $$[3,1,4,1,5,9], \text{"Abel +35"}$$
   then after calculation of the proof of work the book ends with, for example:
   $$[3,1,4,1,5,9], \text{"Abel +35"}, [32,17,37,73,52,90]$$
   It should be remembered that a proof of work is not unique and that it also depends on the `max_list` objective.
   Only one person at a time adds a proof of work. However, everyone has the opportunity to verify that the proposed proof is correct (and should do it). Write a `verification_blockchain()` function, without parameters, that checks that the last proof added to the `blockchain` is valid.
3. Write a blockchain that corresponds to the following data:
   - We take `max_list = [0,0,5]` and start with `blockchain = [ [0,0,0,0,0,0] ]`.

- "`Alfred -100`" (Alfred owes 100 bitcoins).
- Barnabe receives 150.
- Chloe wins 35 bitcoins.

Conclusion: let's imagine that Alfred wants to cheat, he wants to change the account book in order to receive 100 bitcoins instead of having a debit, so he has to change the transaction concerning him to "`Alfred +100`" but then he has to recalculate a new proof of work which is complicated, moreover he has to recalculate the proof of the Barnabe transaction and also that of the Chloe transaction!

Someone who wants to modify a transaction must modify all the following proofs of work. If each proof requires sufficient computing time this is impossible. For the bitcoin each proof requires a lot of calculations (too much for a single person) and a new proof is to be calculated every 10 minutes. It is therefore impossible for a person to modify a past transaction.

The other aspect of the bitcoin that we didn't address is to make sure that each person involved is who they say they are, so that no one can get someone else's money. This is made possible by private/public key cryptography (RSA system). Each account is identified by a public key (two very large integers), which guarantees anonymity. But above all, only the one who has the private key to the account (a large integer) can access his bitcoins.

Random blocks

You will program two methods to build figures that look like algae or corals. Each figure is made up of small randomly thrown blocks that stick together.

Lesson 1 (Fall of blocks).

Square blocks are dropped into a grid, similar to the game "Connect 4": after choosing a column, a block falls from top to bottom. The blocks are placed on the bottom of the grid, on the other blocks, or next to the other blocks. There is a big distinction from the game "Connect 4", here the blocks are "sticky", i.e. a block stays stuck as soon as it meets a neighbor on the left or the right.

Here is an example of how to throw blocks:
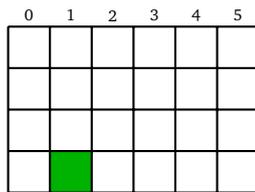
1. Block fallen from col. 1
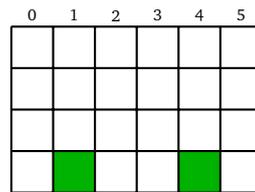
2. Block fallen from col. 4
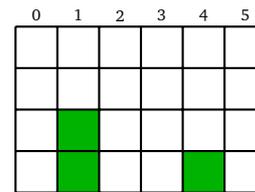
3. Block fallen from col. 1
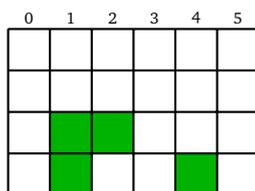
4. Block fallen from col. 2
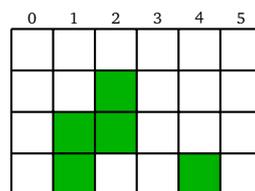
5. Block fallen from col. 2
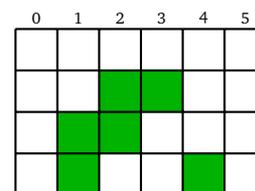
6. Block fallen from col. 3

For example, in step 4, the block thrown in the column 2 does not go down to the bottom but remains hung on its neighbor, so it is permanently suspended.

The random throwing of hundreds of blocks on a large grid produces pretty geometric shapes resembling algae.
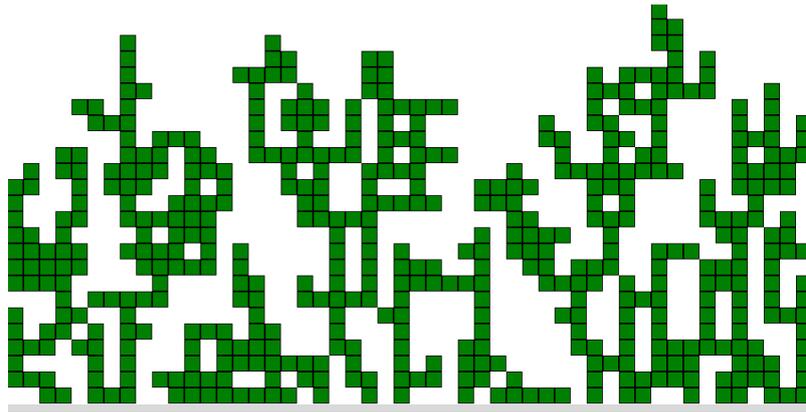
Activity 1 (Fall of blocks).

   Goal: program dropping blocks (without graphic display).

The workspace is modeled by an array of n rows and p columns. At the beginning the array only contains 0's; then the presence of a block is represented by 1.
Here is how to initialize the table:

```
array = [[0 for j in range(p)] for i in range(n)]
```

The table is modified by instructions of the type:

```
array[i][j] = 1
```

Here is an example of a table (left) that represents the graphical situation on the right (the block at the top right is falling).
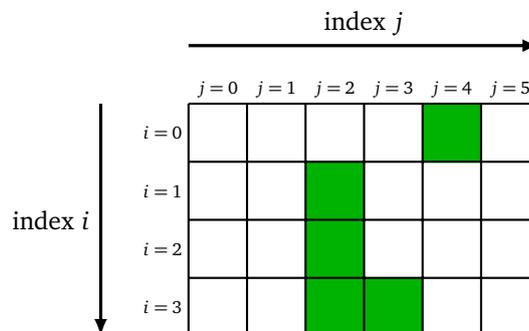
An array with 5 **blocks** ($n = 4$, $p = 6$)

1. Program a `can_fall(i,j)` function that determines if the block in position (i, j) can drop to the square below or not.
   Here are the cases in which the block *cannot* fall:
   - if the block is already on the last line,
   - if there is a block just below,
   - if there is a block just to the right or just to the left.
2. Program a `drop_one_block(j)` function that drops a block in the j column until it can no longer go down. This function modifies the array.
   For example, here is the table before (left) and after (right) dropping a block in the $j = 3$ column.

$$\begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{array} \qquad \begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{array}$$

Before   **After**
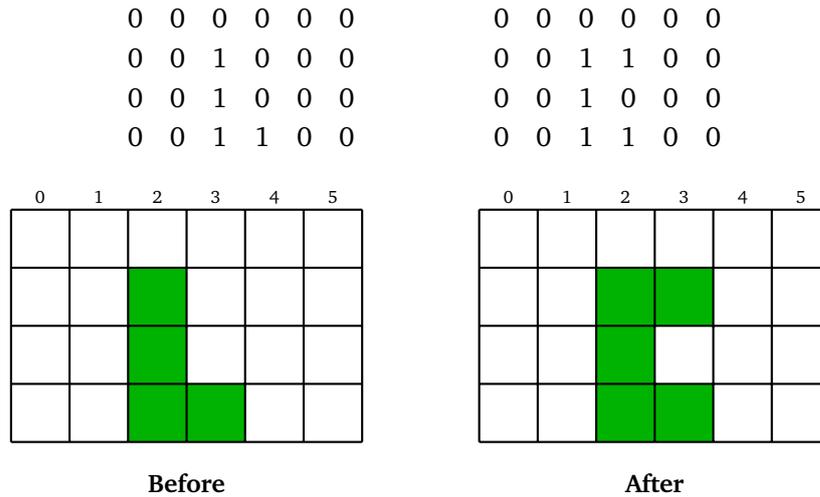
3. Program a `drop_blocks(k)` function that launches k blocks one by one, each time choosing a random column (i.e. an integer j, $0 \leqslant j < p$).

   Here is an example of a table obtained after throwing 10 blocks:

$$\begin{array}{cccccc} 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{array}$$

Throw of 10 **blocks**
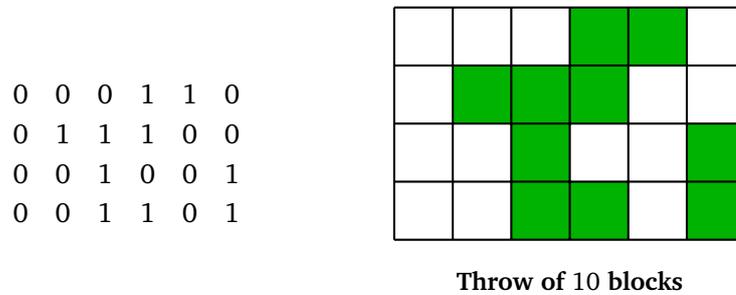
Activity 2 (Falling blocks (continued)).

   Goal: program a graphic display of the blocks.

Static display. Program a graphic display of blocks from an array.
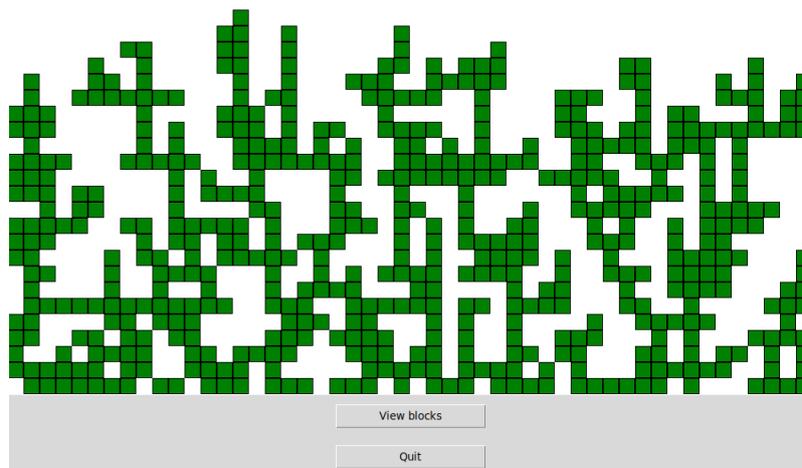
Hints.

• Use the module `tkinter`, see the "Statistics – Data visualization" chapter.

• You can add a button that launches a block (or several at once).

Dynamic display (optional and difficult). Program the display of falling blocks.
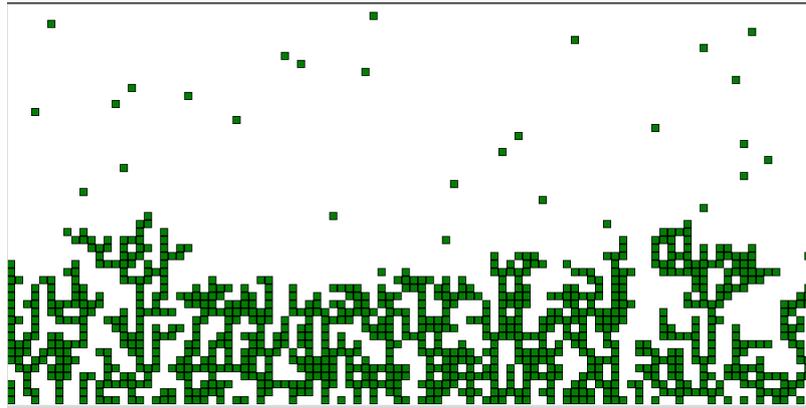
Hints.

- It's much more complicated to program, but very nice to see!
- For moving the blocks, use the program "Moves with `tkinter`" at the end of this chapter.
- How to make a "block rain"? On a regular basis (for example every tenth of a second) all existing blocks are dropped by one square and a new one appears on the top line.

Lesson 2 (Brownian trees)**.**

Here is a slightly different construction, much longer to calculate, but which also draws pretty figures called "brownian trees".
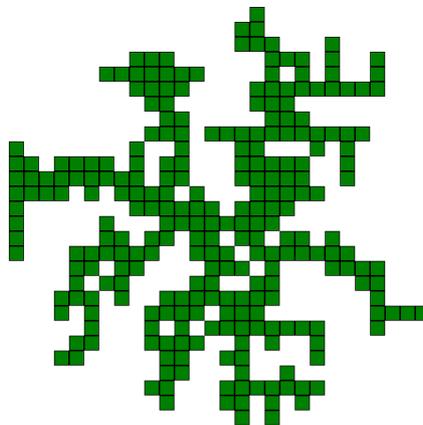
The principle is as follows:

- We start from a grid (this time we have to imagine that it is drawn flat on a table). In its center, we place a first fixed block, the *seed*.
- A random new block is created on the grid. At each step, this block moves at random to one of the eight adjacent squares, this called a *brownian movement*.
- As soon as this block touches another block from one side, it sticks to it and no longer moves.
- If the block leaves the grid, it disintegrates.
- Once the block has been glued or disintegrated, a new block is then restarted from a random point on the grid.
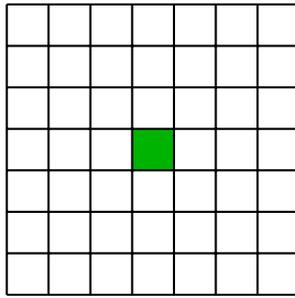
The seed
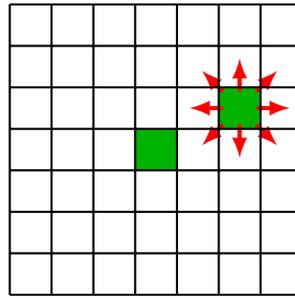
**A new block and its
8 possible moves**
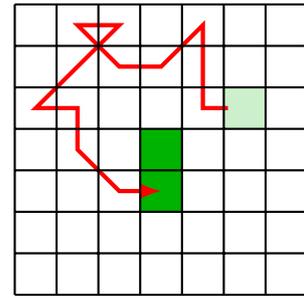
Random move of the block
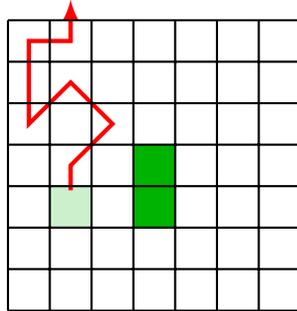
A block leaves the grid
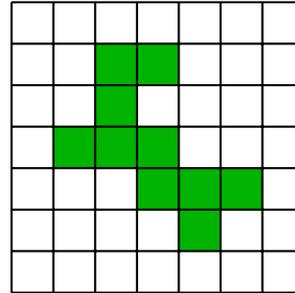
10 blocks

Gradually, we obtain a kind of tree that looks like coral. The calculations are very long because many blocks leave the grid or take a long time to fix (especially at the beginning). In addition, the blocks can only be thrown one by one.

Activity 3 (Brownian trees).

Goal: program the creation of a brownian tree.

Part 1.
1. Model the workspace again with an array of n rows and p columns containing 0's or 1's. Initialize all values to 0, except 1 in the center of the table.
2. Program an `is_inside(i,j)` function which determines if the position (i, j) is in the grid (otherwise the block is coming out).
3. Program a `can_move(i,j)` function that determines if the block in position (i, j) can move (the function returns `True`) or if it is stuck (the function returns `False`).
4. Program a `launch_one_block()` function, without parameters, that simulates the creation of a block and its random movement, until it sticks or leaves the grid.
   Hints.
     • The block is created at a random position (i, j) of the grid.
     • As long as the block is in the grid and free to move:
         – you choose a horizontal move by randomly drawing an integer from $\{-1, 0, +1\}$, the same for a vertical move;
         – you move the block according to the combination of these two movements.
     • Then modify the table.
5. End with a `launch_blocks(k)` function that launches k blocks.
Second part.

Program the graphic display using `tkinter`. You can add a button that launches 10 blocks at once.
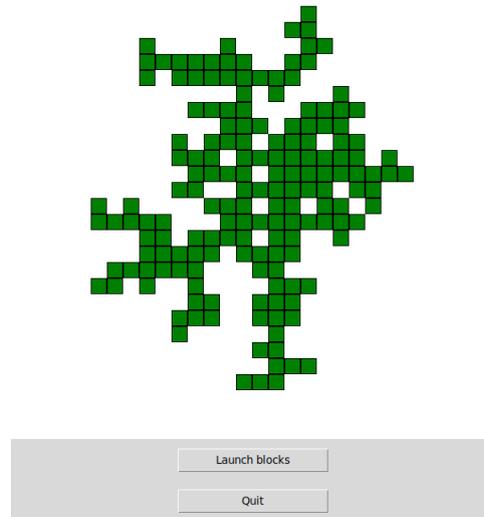
Lesson 3 (Moves with "tkinter").

Here is a program that moves a small square and bounces it off the edges of the window.
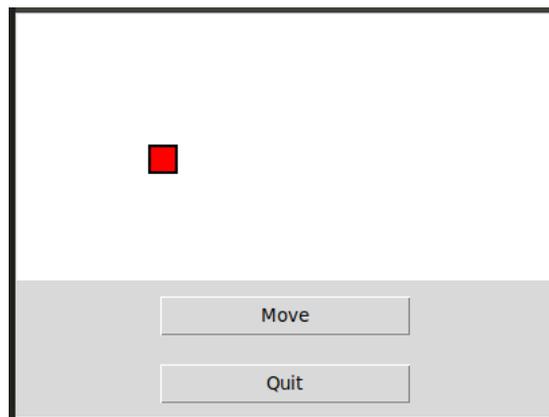
Here are the main points:

- A `rect` object is defined, it is a global variable, as well as its coordinates x0, y0.
- This object is moved by the `mymove()` function which shifts the rectangle by (`dx`,`dy`).
- The key point is that this function will be executed again after a short period of time. The command:

$$\texttt{canvas.after(50,mymove)}$$

  requests a new execution of the `mymove()` function after a short delay (here 50 milliseconds).
- The repetition of small shifts simulates movement.

```
from tkinter import *

the_width = 400
the_height = 200

root = Tk()
canvas = Canvas(root,width=the_width,height=the_height,background="white")
canvas.pack(fill="both", expand=True)
```

```python
# Coordinates and speed
x0, y0 = 100,100
dx = +5  # Horizontal speed
dy = +2  # Vertical speed

# The rectangle to move
rect = canvas.create_rectangle(x0,y0,x0+20,y0+20,width=2,fill="red")

# Main function
def mymove():
    global x0, y0, dx, dy

    x0 = x0 + dx  # New abscissa
    y0 = y0 + dy  # New ordinate

    canvas.coords(rect,x0,y0,x0+20,y0+20)  # Move

    if x0 < 0 or x0 > the_width:
        dx = -dx  # Change of horizontal direction
    if y0 < 0 or y0 > the_height:
        dy = -dy  # Change of vertical direction

    canvas.after(50,mymove)  # Call after 50 milliseconds

    return

# Function for the button
def action_move():
    mymove()
    return

# Buttons
button_move = Button(root,text="Move", width=20, command=action_move)
button_move.pack(pady=10)

button_quit = Button(root,text="Quit", width=20, command=root.quit)
button_quit.pack(side=BOTTOM, pady=10)

root.mainloop()
```

PART V

GUIDES

Python survival guide

1. Test and loops

1.1. If ... then ...

```
if  condition :
    instruction_1
    instruction_2
    ...
other instructions
```

keyword "if"

a condition

colon

indented block
will be executed only
if the condition is true

program continuation

1.2. If ... then ... else ...

```
if  condition :
    instruction
    instruction
    ...
else:
    instruction
    ...
other instructions
```

this block will be executed
if the condition is true

this block will be executed
if the condition is not true

1.3. Loop for

```
                        keywords "for" and "in"
                        a variable
                        list to browse, here 0, 1, 2, ..., n−1
                        colon
for  i  in  range(n) :
    instruction_1       indented block of instructions
    instruction_2       will be repeated n times
    ...                 for i = 0, then i = 1... up to i = n−1
other instructions
                        indentation
                        program continuation
```

1.4. Loop while

```
                        keyword "while"
                        a condition
                        colon
while  condition :
    instruction_1       indented block will
    instruction_2       be executed as long as
    ...                 the condition is true
other instructions
                        program continuation
```

1.5. Quit a loop

The command `break` immediately exits a "while" loop or a "for" loop.

2. Data type

- `int` Integer. Examples: 123 or -15.
- `float` Floating point (or decimal) number. Examples: 4.56, -0.001, 6.022e23 (for 6.022×10^{23}), 4e-3 (for $0.004 = 4 \times 10^{-3}$).
- `str` Character or string. Examples: 'Y', "k", 'Hello', "World!".
- `bool` Boolean. `True` or `False`.
- `list` List. Example: [1,2,3,4].

3. Define functions

3.1. Definition of a function

```
                                        keyword "def"
                                        well-chosen name
                                        parentheses
                                        colon
def  my_function () :
     instruction_1
     instruction_2            instructions
     ...
     return
                                        end of the function
```
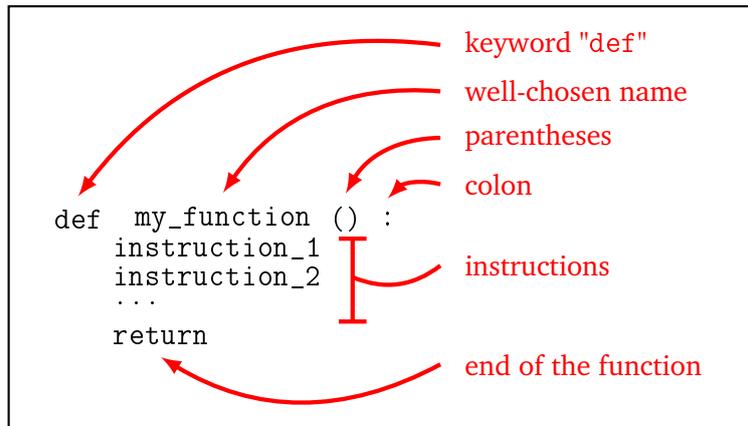
3.2. Function with parameter

Functions achieve their full potential with:

- an *input*, which defines variables that serve as *parameters*,

- an *output*, which is a result returned by the function (and which will often depend on the input parameters).

```
# Definition of the function
def  my_function (param) :
     instruction_1
     instruction_2
     ...                      a parameter
     return  result
                              returns the result


# Call of the function
x = 7
val =  my_function (x)

                              argument

                              call of the function

                              returned result
```
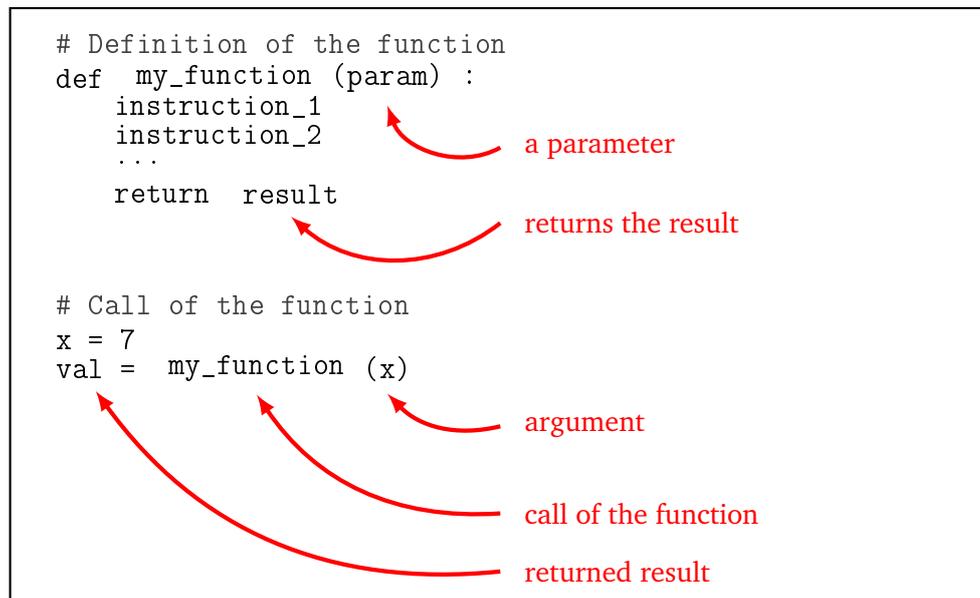
3.3. Function with several parameters

There can be several input parameters, there can be several output results.
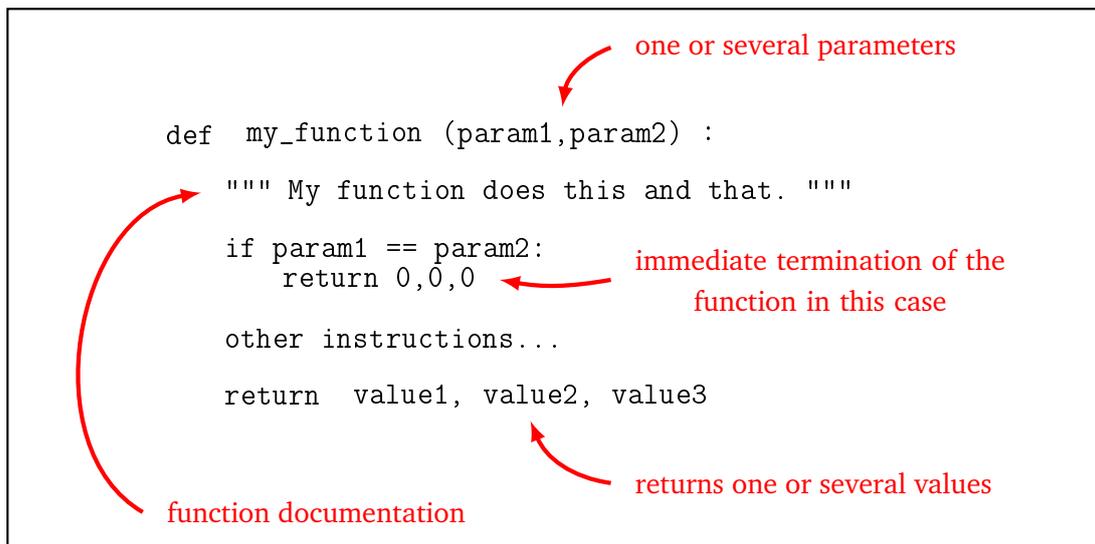
Here is an example of a function with two parameters and two outputs.

```python
def sum_product(a,b):
    """ Computes the sum and product of two numbers. """
    s = a + b
    p = a * b
    return s, p

# Call of the function
mysum, myprod = sum_product(6,7)   # Results
print("The sum is:",mysum)         # Display
print("The product is:",myprod)    # Display
```

- Very important! Do not confuse displaying and returning a value. Display (by the command `print()`) just displays something on the screen. Most functions do not display anything, but instead return one (or more) value. This is much more useful because this value can then be used elsewhere in the program.

- As soon as the program encounters the instruction `return`, the function stops and returns the result. There may be several instances of the `return` instruction in a function but only one will be executed. It is also possible not to put an instruction `return` if the function returns nothing.

- You can, of course, call other functions in the body of your function!

3.4. Comments and docstring

- **Comment.** Anything following the hash sign `#` is a comment and is ignored by `Python`. For example:

```python
# Main loop
while r != 0:   # While this number is not zero
    r = r - 1   # Decrease it by one
```

- **Docstring.** You can describe what a function does by starting it with a *docstring,* i.e. a description (in English) surrounded by three quotation marks. For example:

```python
def product(x,y):
    """ Compute the product of two numbers
    Input: two numbers x and y
    Output: the product of x and y """
```

```
        p = x * y
        return p
```

3.5. Local variable

Here is a very simple function that takes a number as an input and returns the number increased by one.

```
def my_function(x):
    x = x + 1
    return x
```

- Of course `my_function(3)` returns 4.
- If I define a variable by `y = 5` then `my_function(y)` returns 6. And the value of `y` has not changed, it is still equal to 5.
- Here is the delicate situation that you must understand:

```
x = 7
print(my_function(x))
print(x)
```

  - The variable `x` is initialized to 7.
  - The call of the function `my_function(x)` is therefore the same as `my_function(7)` and logically returns 8.
  - What is the value of `x` at the end? The variable `x` is unchanged and is still equal to 7! Even if in the meantime there has been an instruction `x = x + 1`. This instruction changed the `x` inside the function, but not the `x` outside the function.

- Variables defined within a function are called *local variables*. They do not exist outside the function.
- If there is a variable in a function that has the same name as a variable in the program (like the `x` in the example above), it is as if there were two distinct variables; the local variable only exists inside the function.

To understand the scope of the variables, you can color the global variables of a function in red, and the local variables with one color per function. The following small program defines two functions. The first adds one and the second calculates the double.
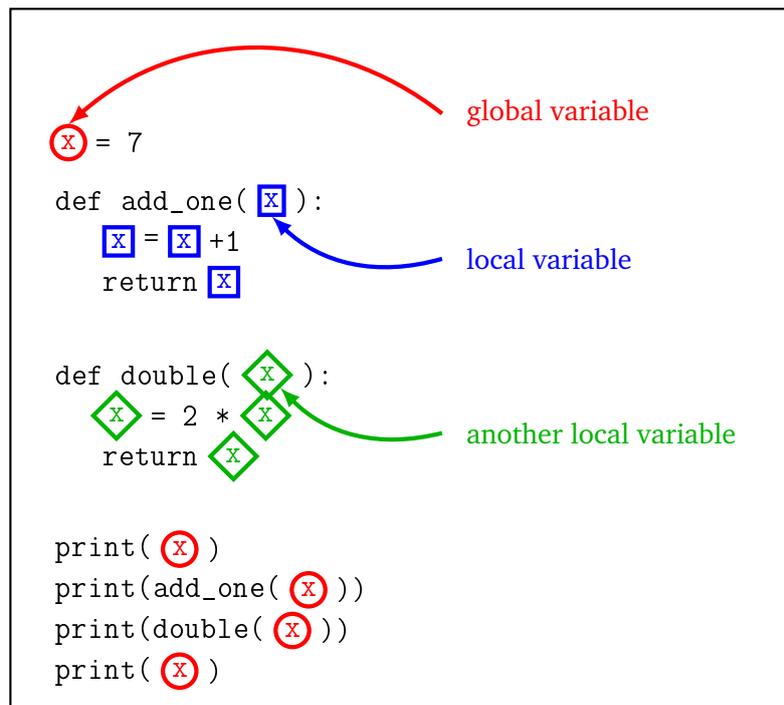
The program first displays the value of x, so 7, then it increases it by 1, so it displays 8, then it displays twice as much as x, so 14. The global variable x has never changed, so the last display of x is still 7.

3.6. Global variable

A *global variable* is a variable that is defined for the entire program. It is generally not recommended to use such variables but it may be useful in some cases. Let us look at an example.

The global variable, here the gravitational constant, is declared at the beginning of the program as a classic variable:

$$\text{gravitation = 9.81}$$

The content of the variable `gravitation` is now available everywhere. On the other hand, if you want to change the value of this variable in a function, you must specify to `Python` that you are aware of modifying a global variable.

For example, for calculations on the Moon, it is necessary to change the gravitational constant, which is much lower there.

```python
def on_the_moon():
    global gravitation   # Yes, I really want to modify this variable!
    gravitation = 1.625  # New value for the entire program
    ...
```

3.7. Optional arguments

It is possible to create optional arguments. Here is how to define a function (that would draw a line) by giving default values:

```python
def draw(length, width=5, color="blue"):
```

- The command `draw(100)` draws my line, and as I only specified the length, the arguments `width` and `color` get the default values (5 and blue).

- The command `draw(100, width=10)` draws my line with a new thickness (the color is the default one).

- The command `draw(100, color="red")` draws my line with a new color (the thickness is the default one).
- The command `draw(100, width=10, color="red")` draws my line with a new thickness and a new color.
- We can also use:
  - `draw(100, 10, "red")`: no need to specify the names of the arguments if you maintain the order.
  - `draw(color="red", width=10, length=100)`: if you name the arguments, then you can pass them in any order.

4. Modules

4.1. Use a module

- `from math import *`   Imports all functions from the `math` module. You are now able to use the sine function for example, by `sin(0)`. This is the simplest method and it is the one we use in this book.
- `import math`   Allows you to use the functions of the `math` module. You can then access the sine function with `math.sin(0)`. This is the officially recommended method to avoid conflicts between modules.

4.2. Main modules

- `math`   contains the main mathematical functions.
- `random`   simulates random draws.
- `turtle`   the `Python` turtle, it is some equivalent of *Scratch*.
- `matplotlib`   allows you to draw graphs and visualize data.
- `tkinter`   allows you to display graphics and windows.
- `time`   for date, time and duration.
- `timeit`   to measure the execution time of a function.

There are many other modules!

5. Errors

5.1. Indentation errors

```
a = 3
 b = 2
```

`Python` returns the error message *IndentationError: unexpected indent*. It also indicates the line number where the indentation error is located, it even points using the symbol "^" to the exact location of the error.

5.2. Syntax errors

- ```
  while x >= 0
      x  = x - 1
  ```

Python returns the error message *SyntaxError: invalid syntax* because the colon is missing after the condition. It should be `while x >= 0 :`

- `string = Hello world!` returns an error because the quotation marks to define the string are missing.

- `print("Hi there"` Python returns the error message *SyntaxError: unexpected EOF while parsing* because the expression is incorrectly parenthesized.

- `if val = 1:` Another syntax error, because you would have to write `if val == 1:`

5.3. Type errors

- **Integer**

```
n = 7.0
for i in range(n):
    print(i)
```

Python returns the error message *TypeError: 'float' object cannot be interpreted as an integer*. Indeed `7.0` is not an integer, but a floating point number.

- **Floating point number**

```
x = "9"
sqrt(x)
```

Python returns the error message *TypeError: a float is required,* because `"9"` is a string and not a number.

- **Wrong number of arguments**
  `gcd(12)` Python returns the error message *TypeError: gcd() takes exactly 2 arguments (1 given)* because the `gcd()` function of the `math` module requires two arguments, such as `gcd(12,18)`.

5.4. Name errors

- `if y != 0: y = y - 1` Python returns the message *NameError: name 'y' is not defined* if the variable y has not yet been defined.

- This error can also occur if upper/lower case letters are not scrupulously respected. `variable`, `Variable` and `VARIABLE` are three different variable names.

- `x = sqrt(2)` Python returns the message *NameError: name 'sqrt' is not defined,* you must import the `math` module to be able to use the `sqrt()` function.

- **Function not yet defined**

```
product(6,7)

def product(a,b):
    return a*b
```

Returns an error *NameError: name 'product' is not defined* because a function must be defined before it can be used.

5.5. Exercise

Fix the code! `Python` should display 7 5 9.

```
a == 7
if (a = 2) or (a >= 5)
    b = a - 2
     c = a + 2
else
b = a // 2
c = 2 * a
print(a b c)
```

5.6. Other problems

The program starts but stops along the way or doesn't do what you want? That's where the trouble starts, you have to debug the code! There are no general solutions but only a few tips:

- A clean, well structured, well commented code, with well chosen variable and function names, is easier to read.

- Test your algorithm with paper and pencil for easy cases.

- Do not hesitate to display the values of the variables, to see their evolution over time. For example `print(i,mylist[i])` in a loop.

- A better way to inspect the code is to view the values associated with the variables using the features *debug* of your favorite `Python` editor. It is also possible to make a step by step execution.

- Does the program work with some values and not others? Have you thought about extreme cases? Is *n* zero but was not allowed to? Is the list empty, while the program does not handle this case? etc.

Here are some examples.

- I want to display the squares of integers from 1 to 10. The following program does not return any errors but does not do what I want.

```
for i in range(10):
    print(i ** 2)
```

The loop iterates on integers from 0 to 9. You have to write `range(1,11)`.

- I want to display the last item in my list.

```
mylist = [1,2,3,4]
print(mylist[4])
```

`Python` returns the error message *IndexError: list index out of range* because the last element is the one of rank 3.

- I want to do a countdown. The next program never stops.

```
n = 10
while n != "0":
    n = n - 1
    print(n)
```

With a loop "while" you have to take great care to write the condition well and check that it ends up being wrong. Here, it is poorly formulated, it should be `while n!= 0:`

Main functions

1. Mathematics

Classical operations
- `a + b,  a - b,  a * b` classic operations
- `a / b` "real" division (returns a floating point number)
- `a // b` Euclidean division quotient (returns an integer)
- `a % b` remainder of the Euclidean division, called a modulo b
- `abs(x)` absolute value
- `x ** n` power x^n
- `4.56e12` for 4.56×10^{12}

"math" module
The use of other mathematical functions requires the `math` module which is called by the command:
$$\text{from math import *}$$
- `sqrt(x)` square root $\sqrt{x}$
- `cos(x),  sin(x),  tan(x)` trigonometric functions $\cos x$, $\sin x$, $\tan x$ in radians
- `pi` approximate value of $\pi = 3.14159265\ldots$
- `floor(x)` integer just below x
- `ceil(x)` integer just above x
- `gcd(a,b)` gcd of a and b

"random" module
The `random` module generates numbers in a pseudo-random way. It is called by the command:
$$\text{from random import *}$$
- `random()` on each call, returns a floating number x at random, satisfying $0 \leqslant x < 1$.
- `randint(a,b)` for each call, returns an integer n at random, satisfying $a \leqslant n \leqslant b$.
- `choice(mylist)` on each call, randomly draws an item from the list.
- `mylist.shuffle()` mixes the list (the list is modified).

Binary notation
- `bin(n)` returns the binary notation of the integer n as a string. Example: `bin(17)` returns `'0b10001'`.
- To write a number directly in binary notation, simply write the number starting with `0b` (without quotation marks). For example `0b11011` is equal to 27.

2. Booleans

A boolean is a data that takes either the value `True` or the value `False`.

Comparisons
The following comparison tests return a boolean.
- `a == b`   equality test
- `a < b`   strict lower test
- `a <= b`   large lower test
- `a > b`   or   `a >= b`   higher test
- `a != b`   non-equality test

Do not confuse "a = b" (assignment) and "a == b" (equality test).

Boolean operations
- `P and Q`   logical "and"
- `P or Q`   logical "or"
- `not P`   negation

3. Strings I

Strings
- `"A"`   or   `'A'`   one character
- `"Python"`   or   `'Python'`   a string
- `len(string)`   the string length. Example: `len("Python")` returns 6.
- `string1 + string2`   concatenation.
  Example: `"I love" + "Python"` returns `"I lovePython"`.
- `string[i]`   returns the i-th character of `string` (numbering starts at 0).
  Example with `string = "Python"`, `string[1]` is equal to `"y"`. See the table below.

Letter	**P**	**y**	**t**	**h**	**o**	**n**
Rank	0	1	2	3	4	5

Number/string conversion
- **String.** `str(number)`   converts a number (integer or floating point number) into a string. Examples: `str(7)` returns the string `"7"`; `str(1.234)` returns the string `"1.234"`.
- **Integer.** `int(string)`   returns the integer corresponding to the string. Example: `int("45")` returns the integer 45.
- **Floating point number.** `float(string)`   returns the floating point number corresponding to the string. Example: `float("3.14")` returns the number 3.14.

Substrings
- `string[i:j]`   returns the substring of characters with from rank i to rank $j-1$ of `string`.
  Example: with `string = "This is a string"`, `string[2:7]` returns `"is is"`.
- `string[i:]`   returns characters from rank i until the end of `string`.
  Example: `string[5:]` returns `"is a string"`.
- `string[:j]`   returns characters from the beginning to rank $j-1$ of `string`. Example: `string[:4]` returns `"This"`.

Format

The `format()` method allows you to format text or numbers. This function returns a string.

- **Text**

<div align="center">

Test␣␣␣␣␣␣   ␣␣␣␣␣␣Test   ␣␣␣Test␣␣␣

</div>

  - `'{:10}'.format('Test')`   left alignment (on 10 characters)
  - `'{:>10}'.format('Test')`   right alignment
  - `'{:^10}'.format('Test')`   centered

- **Integer**

<div align="center">

456   ␣␣␣456   000456

</div>

  - `'{:d}'.format(456)`   integer
  - `'{:6d}'.format(456)`   right aligned (on 6 characters)
  - `'{:06d}'.format(456)`   adding leading zeros (on 6 characters)

- **Floating point number**

<div align="center">

3.141593   3.14159265   ␣␣3.1416   003.1416

</div>

  - `'{:f}'.format(3.14159265653589793)`   floating point number
  - `'{:.8f}'.format(3.14159265653589793)`   8 decimal places
  - `'{:8.4f}'.format(3.14159265653589793)`   on 8 characters with 4 numbers after the decimal point
  - `'{:08.4f}'.format(3.141592653589793)`   adding leading zeros

4. Strings II

Encoding

- `chr(n)`   returns the character associated with the ASCII/unicode code number n. Example: `chr(65)` returns `"A"`; `chr(97)` returns `"a"`.
- `ord(c)`   returns the ASCII/unicode code number associated with the character c. Example: `ord("A")` returns 65; `ord("a")` returns 97.

The beginning of the ASCII/unicode table is given below.

33	!	43	+	53	**5**	63	**?**	73	**I**	83	**S**	93	**]**	103	**g**	113	**q**	123	**{**
34	"	44	,	54	**6**	64	**@**	74	**J**	84	**T**	94	**^**	104	**h**	114	**r**	124	**|**
35	#	45	-	55	**7**	65	**A**	75	**K**	85	**U**	95	**_**	105	**i**	115	**s**	125	**}**
36	$	46	.	56	**8**	66	**B**	76	**L**	86	**V**	96	'	106	**j**	116	**t**	126	**~**
37	%	47	/	57	**9**	67	**C**	77	**M**	87	**W**	97	**a**	107	**k**	117	**u**	127	-
38	&	48	**0**	58	:	68	**D**	78	**N**	88	**X**	98	**b**	108	**l**	118	**v**		
39	'	49	**1**	59	;	69	**E**	79	**O**	89	**Y**	99	**c**	109	**m**	119	**w**		
40	(	50	**2**	60	<	70	**F**	80	**P**	90	**Z**	100	**d**	110	**n**	120	**x**		
41	)	51	**3**	61	=	71	**G**	81	**Q**	91	**[**	101	**e**	111	**o**	121	**y**		
42	*	52	**4**	62	>	72	**H**	82	**R**	92	****	102	**f**	112	**p**	122	**z**		

Upper/lower-case
- `string.upper()` returns a string in uppercase.
- `string.lower()` returns a string in lowercase.

Search/replace
- `substring in string` returns "true" or "false" depending on if `substring` appears in `string`.
  Example: `"NOT" in "TO BE OR NOT TO BE"` returns `True`.
- `string.find(substring)` returns the rank at which the substring was found (and `-1` otherwise).
  Example: with `string = "ABCDE"`, `string.find("CD")` returns 2.
- `string.replace(substring,new_substring)` replaces each occurrence of the substring by the new substring.
  Example: with `string = "ABCDE"`, `string.replace("CD","XY")` returns `"ABXYE"`.

Split/join
- `string.split(separator)` separates the string into a list of substrings (by default the separator is the space).
  Examples:
    - `"To be or not to be.".split()` returns `['To', 'be', 'or', 'not', 'to', 'be.']`
    - `"12.5;17.5;18".split(";")` returns `['12.5', '17.5', '18']`
- `separator.join(mylist)` groups the substrings into a single string by adding the separator between each.
  Examples:
    - `"".join(["To", "be", "or", "not", "to", "be."])` returns the string `'Tobeornottobe.'` Spaces are missing.
    - `" ".join(["To", "be", "or", "not", "to", "be."])` returns `'To be or not to be.'` It's better when the separator is a space.
    - `"--".join(["To", "be", "or", "not", "to", "be."])` returns the string `'To--be--or--not--t`

5. Lists I

Construction of a list

Examples:

- `mylist1 = [5,4,3,2,1]`   a list of five integers.
- `mylist2 = ["Friday","Saturday","Sunday"]`   a list of three strings.
- `mylist3 = []`   the empty list.
- `list(range(n))`   list of integers from 0 to $n-1$.
- `list(range(a,b))`   list of integers from a to $b-1$.
- `list(range(a,b,step))`   list of integers from a to $b-1$, with a step given by the integer `step`.

Get an item

- `mylist[i]`   returns the element at rank i. Be careful, the rank starts at 0.
  Example: `mylist = ["A","B","C","D","E","F"]` then `mylist[2]` returns `"C"`.

Letter	"A"	"B"	"C"	"D"	"E"	"F"
Rank	0	1	2	3	4	5

- `mylist[-1]`   returns the last element, `mylist[-2]` returns the second last element...
- `mylist.pop()`   removes the last item from the list and returns it.

Add one element (or more)

- `mylist.append(element)`   adds the item at the end of the list. Example: if `mylist = [5,6,7,8]` then `mylist.append(9)` adds 9 to the list, `mylist` is now `[5,6,7,8,9]`.
- `new_mylist = mylist + [element]`   provides a new list with an extra element at the end. Example: `[1,2,3,4] + [5]` is `[1,2,3,4,5]`.
- `[element] + mylist`   returns a list where the item is added at the beginning. Example: `[5] + [1,2,3,4]` is `[5,1,2,3,4]`.
- `mylist1 + mylist2`   concatenates the two lists. Example: with `mylist1 = [4,5,6]` and `mylist2 = [7,8,9]` then `mylist1 + mylist2` is `[4,5,6,7,8,9]`.

Example of construction. Here is how to build the list that contains the first squares:

```
list_squares = []                # We start from an empty list
for i in range(10):
    list_squares.append(i**2)    # We add squares one by one
```

At the end `list_squares` is:

$$[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]$$

Browse a list

- `len(mylist)`   returns the length of the list. Example: `len([5,4,3,2,1])` returns 5.
- Browse a list (and here display each item):

```
for element in mylist:
    print(element)
```

- Browse a list using the rank.

```
    n = len(mylist)
    for i in range(n):
        print(i,mylist[i])
```

6. Lists II

Mathematics
- `max(mylist)`   returns the largest element. Example: `max([10,16,13,14])` returns 16.
- `min(mylist)`   returns the smallest element. Example: `min([10,16,13,14])` returns 10.
- `sum(mylist)`   returns the sum of all elements. Example: `sum([10,16,13,14])` returns 53.

Slicing lists
- `mylist[a:b]`   returns the sublist of elements from rank a to rank $b-1$.
- `mylist[a:]`   returns the list of elements from rank a until the end.
- `mylist[:b]`   returns the list of items from the beginning to rank $b-1$.

Letter	"A"	"B"	"C"	"D"	"E"	"F"	"G"
Rank	0	1	2	3	4	5	6

For example if `mylist = ["A","B","C","D","E","F","G"]` then:
- `mylist[1:4]`   returns `["B","C","D"]`.
- `mylist[:2]`   is like `mylist[0:2]` and returns `["A","B"]`.
- `mylist[4:]`   returns `["E","F","G"]`. It's the same thing as `mylist[4:n]` where n = `len(mylist)`.

Find the rank of an element
- `mylist.index(element)` returns the first position at which the item was found. Example: with `mylist = [12, 30, 5, 9, 5, 21]`, `mylist.index(5)` returns 2.
- If you just want to know if an item belongs to a list, then the statement:

$$\texttt{element in mylist}$$

  returns `True` or `False`. Example: with `mylist = [12, 30, 5, 9, 5, 21]`, "9 in mylist" is true, while "8 in mylist" is false.

Order
- `sorted(mylist)`   returns the ordered list of items.
  Example: `sorted([13,11,7,4,6,8,12,6])` returns the list $[4,6,6,7,8,11,12,13]$.
- `mylist.sort()`   does not return anything but the list `mylist` is now ordered.

Invert a list
Here are three methods:
- `mylist.reverse()`   modifies the list in place;
- `list(reversed(mylist))`   returns a new list;
- `mylist[::-1]`   returns a new list.

Delete an item
Three methods.

- `mylist.remove(element)` deletes the first occurrence found.
  Example: `mylist = [2,5,3,8,5]`, the instruction `mylist.remove(5)` modifies the list which is now `[2,3,8,5]` (the first 5 has disappeared).
- `del mylist[i]` deletes element at rank i (the list is modified).
- `element = mylist.pop()` removes the last item from the list and returns it.

List comprehension

- Let's start from a list, for example `mylist = [1,2,3,4,5,6,7,6,5,4,3,2,1]`.
- `list_doubles = [ 2*x for x in mylist ]` returns a list that contains the doubles of the items of `mylist`. So this is the list `[2,4,6,8,...]`.
- `liste_squares = [ x**2 for x in mylist ]` returns the list of squares of the items in the list `mylist`. So this is the list `[1,4,9,16,...]`.
- `partial_list = [ x for x in mylist if x > 2 ]` extracts from the list only the elements greater than 2. So this is the list `[3,4,5,6,7,6,5,4,3]`.

List of lists
Example:

$$\text{array} = [\ [2,14,5], \ [3,5,7], \ [15,19,4], \ [8,6,5] \ ]$$

corresponds to the table:

Then `array[i]` returns the sublist of rank i, and `array[i][j]` returns the element located in the sublist number i, at rank j of this sublist. For example:

- `array[0]` returns the sublist `[2,14,5]`.
- `array[1]` returns the sublist `[3,5,7]`.
- `array[0][0]` returns the integer 2.
- `array[0][1]` returns the integer 14.
- `array[2][1]` returns the integer 19.

A table of n rows and p columns.

- `array = [[0 for j in range(p)] for i in range(n)]` initializes an array and fills it with 0.
- `array[i][j] = 1` modifies a value in the table (the one at the location (i, j)).

7. Input/output

Display

- `print(string1,string2,string3,...)` displays strings or objects. Example: `print("Value =",14)` displays Value = 14. Example: `print("Line 1 \n Line 2")` displays on two lines.
- **Separator.** `print(...,sep="...")` changes the separator (by default the separator is the space character). Example: `print("Bob",17,13,16,sep="; ")` displays Bob; 17; 13; 16.
- **End of line.** `print(...,end="...")` changes the character placed at the end (by default it is the line break character \n). Example `print(17,end="")` then `print(76)` displays 1776 on a single line.

Keyboard entry

`input()` pauses the program and waits for the user to send a message on the keyboard (ended by pressing the "Enter" key). The message is a string.

Here is a small program that asks for the user's first name and age and displays a message like "Hello Kevin" then "You are a minor/adult" according to age.

```
first_name = input ("What's your name? ")
print("Hello",first_name)

age_str = input("How old are you? ")
age = int(age_str)

if age >= 18:
    print("You're an adult!")
else:
    print("You're a minor!")
```

8. Files

Order

- `fi = open("my_file.txt","r")` opening in reading ("r" for read).
- `fi = open("my_file.txt","w")` opening in writing ("w" for write). The file is created if it does not exist, if it existed the previous content is first deleted.
- `fi = open("my_file.txt","a")` opening for writing, the data will be written at the end of the current data ("a" for append).
- `fi.write("one line")` write to the file.
- `fi.read()` reads the whole file (see below for another method).
- `fi.readlines()` reads all the lines (see below for another method).
- `fi.close()` file closing.

Write lines to a file

```
fi = open("my_file.txt","w")

fi.write("Hello world!\n")

line = "Hi there.\n"
```

```
fi.write(line)

fi.close()
```

Read lines from a file

```
fi = open("my_file.txt","r")

for line in fi:
    print(line)

fi.close()
```

Read a file (official method)

```
with open("my_file.txt","r") as fi:
    for line in fi:
        print(line)
```

9. Turtle

The `turtle` module is called by the command:

$$\text{from turtle import *}$$

Main commands
  • `forward(length)` advances a number of steps
  • `backward(length)` goes backwards
  • `right(angle)` turns to the right (without advancing) at a given angle in degrees
  • `left(angle)` turns left
  • `setheading(direction)` points in a direction ($0 = $ right, $90 = $ top, $-90 = $ bottom, $180 = $ left)
  • `goto(x,y)` moves to the point (x, y)
  • `setx(newx)` changes the value of the abscissa
  • `sety(newy)` changes the value of the ordinate
  • `down()` sets the pen down
  • `up()` sets the pen up
  • `width(size)` changes the thickness of the line
  • `color(col)` changes the color: `"red"`, `"green"`, `"blue"`, `"orange"`, `"purple"`...
  • `position()` returns the (x, y) position of the turtle
  • `heading()` returns the direction `angle` to which the turtle is pointing
  • `towards(x,y)` returns the angle between the horizontal and the segment starting at the turtle and ending at the point (x, y)
  • `speed("fastest")`   maximum travel speed
  • `exitonclick()` ends the program as soon as you click

Several turtles
Here is an example of a program with two turtles.

```
turtle1 = Turtle()    # with capital 'T'!
turtle2 = Turtle()

turtle1.color('red')
turtle2.color('blue')

turtle1.forward(100)
turtle2.left(90)
turtle2.forward(100)
```

10. Matplotlib

With the `matplotlib` module it is very easy to draw a list. Here is an example.
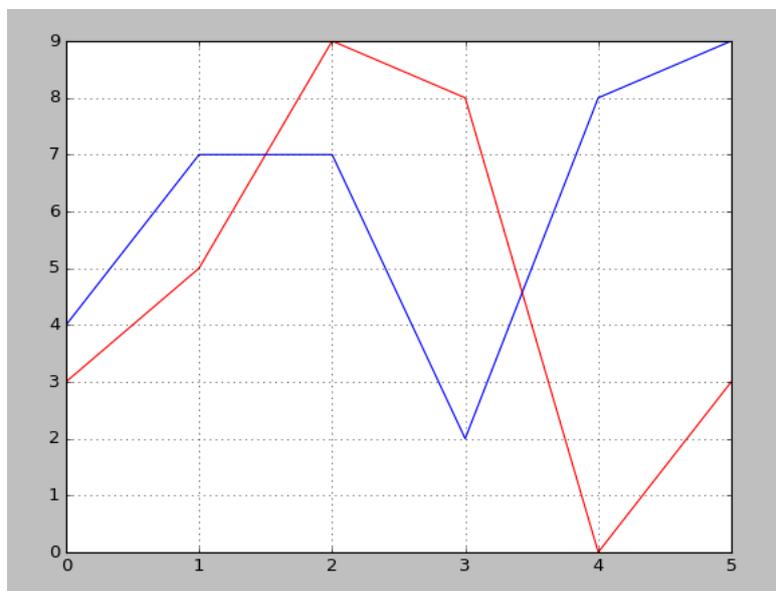

```
import matplotlib.pyplot as plt

mylist1 = [3,5,9,8,0,3]
mylist2 = [4,7,7,2,8,9]

plt.plot(mylist1,color="red")
plt.plot(mylist2,color="blue")
plt.grid()
plt.show()
```

Main functions.

- `plt.plot(mylist)` traces the points of a list (in the form (i, ℓ_i)) that are joined by segments.
- `plt.plot(list_x,list_y)` traces the points of a list (of the form (x_i, y_i) where x_i browses the first list and y_i the second).
- `plt.scatter(x,y,color='red',s=100)` displays a point at (x, y) (of a size s).
- `plt.grid()` draws a grid.
- `plt.show()` displays everything.
- `plt.close()` exits the display.

- `plt.xlim(xmin,xmax)` defines the interval for the x.
- `plt.ylim(ymin,ymax)` defines the interval for the y.
- `plt.axis('equal')` imposes an orthonormal basis.

11. Tkinter

11.1. Graphics

To display this:

The code is:

```
# tkinter window
root = Tk()

canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(fill="both", expand=True)

# A rectangle
canvas.create_rectangle(50,50,150,100,width=2)

# A rectangle with thick blue edges
canvas.create_rectangle(200,50,300,150,width=5,outline="blue")

# A rectangle filled with purple
canvas.create_rectangle(350,100,500,150,fill="purple")

# An ellipse
canvas.create_oval(50,110,180,160,width=4)

# Some text
canvas.create_text(400,75,text="Bla bla bla bla",fill="blue")

# Launch of the window
root.mainloop()
```

Some explanations:
- The `tkinter` module allows us to define variables `root` and `canvas` that determine a graphic window (here width 800 and height 600 pixels). Then describe everything you want to add to the window. And finally the window is displayed by the command `root.mainloop()` (at the very end).

- Attention! The window's graphic marker has its y-axis pointing downwards. The origin $(0,0)$ is the top left corner (see figure below).
- Command to draw a rectangle: `create_rectangle(x1,y1,x2,y2)`; just specify the coordinates $(x_1, y_1), (x_2, y_2)$ of two opposite vertices. The option `width` adjusts the thickness of the line, `outline` defines the color of this line, `fill` defines the filling color.
- An ellipse is traced by the command `create_oval(x1,y1,x2,y2)`, where $(x_1, y_1), (x_2, y_2)$ are the coordinates of two opposite vertices of a rectangle framing the desired ellipse (see figure). A circle is obtained when the corresponding rectangle is a square!
- Text is displayed by the command `canvas.create_text(x,y,text="My text")` specifying the coordinates (x, y) of the point from which you want to display the text.

11.2. Buttons

It is more ergonomic to display windows where actions are performed by clicking on buttons. Here is the window of a small program with two buttons. The first button changes the color of the rectangle, the second button ends the program.

The code is:

```
from tkinter import *
from random import *
```

```
root = Tk()
canvas = Canvas(root, width=400, height=200, background="white")
canvas.pack(fill="both", expand=True)

def action_button():
    canvas.delete("all")      # Clear all
    colors = ["red","orange","yellow","green","cyan","blue","purple"]
    col = choice(colors)      # Random color
    canvas.create_rectangle(100,50,300,150,width=5,fill=col)
    return

button_color = Button(root,text="View", width=20, command=action_button)
button_color.pack(pady=10)

button_quit = Button(root,text="Quit", width=20, command=root.quit)
button_quit.pack(side=BOTTOM, pady=10)

root.mainloop()
```

Some explanations:

- A button is created by the command `Button`. The `text` option customizes the text displayed on the button. The button created is added to the window by the method `pack`.

- The most important thing is the action associated with the button! It is the option `command` that receives the name of the function to be executed when the button is clicked. For our example `command=action_button`, associate the click on the button with a change of color.

- Attention! You have to give the name of the function without brackets: `command = my_function` and not `command = my_function()`.

- To associate the button with "Quit" and close the window, the argument is `command = root.quit`.

- The instruction `canvas.delete("all")` deletes all drawings from our graphic window.

11.3. Text

Here's how to display text with `Python` and the graphics window module `tkinter`.

The code is:

```
from tkinter import *
from tkinter.font import Font
# tkinter window
root = Tk()
canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(fill="both", expand=True)
# Font
myfont = Font(family="Times", size=30)
# Some text
canvas.create_text(100,100, text="Text with Python!",
```

```
anchor=SW, font=myfont, fill="blue")
# Launch the window
root.mainloop()
```

Some explanations:

- `root` and `canvas` are the variables that define a graphic window (here of width 800 and height 600 pixels). This window is launched by the last command: `root.mainloop()`.

- We remind you that for the graphic coordinates, the y-axis is directed downwards. To define a rectangle, simply specify the coordinates (x_1, y_1) and (x_2, y_2) from two opposite vertices (see figure below).

- The text is displayed by the command `canvas.create_text()`. It is necessary to specify the (x, y) coordinates of the point from which you want to display the text.

- The `text` option allows you to pass the string to display.

- The `anchor` option allows you to specify the text anchor point, `anchor=SW` means that the text box is anchored to the Southwest point (*SW*) (see figure below).

- The `fill` option allows you to specify the text color.

- The `font` option allows you to define the font (i.e. the style and size of the characters). Here are some examples of fonts, it's up to you to test them:
    - `Font(family="Times", size=20)`
    - `Font(family="Courier", size=16, weight="bold")` in **bold**
    - `Font(family="Helvetica", size=16, slant="italic")` in *italic*
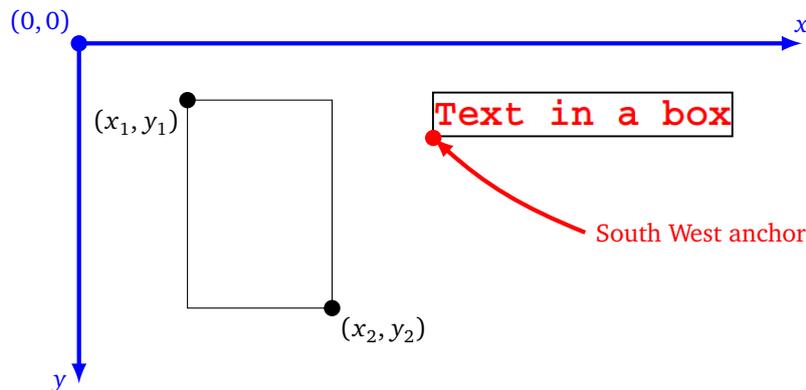
11.4. Mouse click

Here is a small program that displays a graphic window. Each time the user clicks (with the left mouse button) the program displays a small square (on the window) and displays "Click at $x = \ldots$, $y = \ldots$" (on the console).

```
from tkinter import *

# Window
root = Tk()
canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(side=LEFT, padx=5, pady=5)

# Catch mouse click
def action_mouse_click(event):
    canvas.focus_set()
```

```
    x = event.x
    y = event.y
    canvas.create_rectangle(x,y,x+10,y+10,fill="red")
    print("Click at x =",x,", y =",y)
    return

# Association click/action
canvas.bind("<Button-1>", action_mouse_click)

# Launch
root.mainloop()
```

Here are some explanations:

- The creation of the window is usual. The program ends with the launch using the command `mainloop()`.
- The first key point is to associate a mouse click to an action, that's what the line does:

  `canvas.bind("<Button-1>", action_mouse_click)`

  Each time the left mouse button is clicked, Python executes the `action_mouse_click` function. (Note that there are no brackets for the call to the function.)
- Second key point: the `action_mouse_click` function retrieves the click coordinates and then does two things here: it displays a small rectangle at the click location and prints the (x, y) coordinates in the terminal window.
- The coordinates x and y are expressed in pixels; $(0, 0)$ refers to the upper left corner of the window (the area delimited by `canvas`).

11.5. Movement

Here is a program that moves a small square and bounces it off the edges of the window.
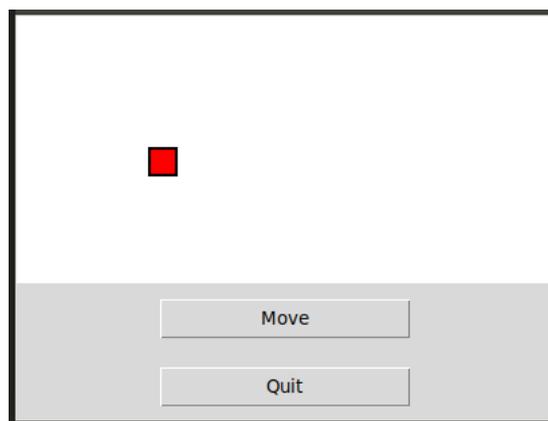
Here are the main points:

- An object `rect` is defined, it is a global variable, as well as its coordinates x0, y0.
- This object is (a little bit) moved by the function `mymove()` which shifts the rectangle by (dx,dy).
- The key point is that this function will be executed again after a short period of time. The command:

  `canvas.after(50,mymove)`

  requests a new execution of the function `mymove()` after a short delay (here 50 milliseconds).
- The repetition of small shifts simulates movement.

```
from tkinter import *
```

```python
the_width = 400
the_height = 200

root = Tk()
canvas = Canvas(root,width=the_width,height=the_height,background="white")
canvas.pack(fill="both", expand=True)

# Coordinates and speed
x0, y0 = 100,100
dx = +5  # Horizontal speed
dy = +2  # Vertical speed

# The rectangle to move
rect = canvas.create_rectangle(x0,y0,x0+20,y0+20,width=2,fill="red")

# Main function
def mymove():
    global x0, y0, dx, dy

    x0 = x0 + dx  # New abscissa
    y0 = y0 + dy  # New ordinate

    canvas.coords(rect,x0,y0,x0+20,y0+20)  # Move

    if x0 < 0 or x0 > the_width:
        dx = -dx  # Change of horizontal direction
    if y0 < 0 or y0 > the_height:
        dy = -dy  # Change of vertical direction

    canvas.after(50,mymove)  # Call after 50 milliseconds

    return

# Function for the button
def action_move():
    mymove()
    return

# Buttons
button_move = Button(root,text="Move", width=20, command=action_move)
button_move.pack(pady=10)

button_quit = Button(root,text="Quit", width=20, command=root.quit)
button_quit.pack(side=BOTTOM, pady=10)

root.mainloop()
```

Notes and references

You will find here comments and readings for each activity.

General resources

- The official `Python` documentation contains tutorials and explanations of each function.

  docs.python.org/3/

- *Wikipedia* is a reliable source for learning more about some concepts (mainly projects) but the level is not always suitable for a high school student.

- *Internet* and in particular the forums often have answers to questions you ask yourself!

- The most experienced among you can participate to the *Euler project* which offers a list of mathematics-with-computer puzzles.

  projecteuler.net

1. Hello world!

Learning a programming language can be very difficult. It's hard enough to learn alone and it is not uncommon to stay blocked for several hours for a stupid syntax error. You have to start small, don't hesitate to copy code written by others, be persistent and quickly ask for help!

2. Turtle (Scratch with Python)

The ideal is to master *Scratch* before attacking `Python`. `Python` offers a *Turtle* module that works on the same principle as *Scratch*. Of course instead of moving blocks, you have to write the code! It is a good exercise to transcribe into `Python` all the activities you can do with *Scratch*.

3. If ... then ...

We really start the programming with the "if/else" test. The computer therefore acts in one way or another depending on the situation. It is no longer an automaton that always does the same thing. In addition to the keyboard entry, we can start having interactive programs.

The classic syntax errors are:

- forget the colon after `if condition:` or `else:`,
- incorrectly indent the blocks.

These errors will be reported by `Python` with the line number where there is a problem (normally your editor places the cursor on the wrong line). On the other hand, if the program starts but does not do the right thing, it is surely the condition that is incorrectly formulated. It is more complicated to understand the right condition: a little logic and reflection with paper and pencil are welcome. Some editors also allow a step-by-step execution of the `Python` program.

There is also the test

```
if ... elif ... elif ... else ...
```

which allows you to run the tests in sequence and which is not covered here. Understanding `if ... else ...` is enough.

4. Functions

Quite quickly it is necessary to understand the structure of a computer program: the program is broken down into blocks of definitions and simple actions. Simple actions are grouped into intermediate actions. And at the end the main program is just to perform some good functions.

The arguments/parameters of a function are a delicate learning process. You can define a function by `def func(x):` and call it by `func(y)`. The x corresponds to a dummy mathematical variable. In computing, we prefer to talk about the *scope* of the variable which can be local or global.

What we can remember is that nothing that happens within a function is accessible outside the function. You just have to use what the function returns. It is necessary to avoid the use of `global` in the first instance. Last but not least, it is really important to comment extensively on your code and explain what your functions are doing. If you define a function `func(x)`, plan three lines of comments to (a) say what this function does, (b) say what input is expected (x must be an integer? a floating point number ? positive?...), (c) say what the function returns.

It is also necessary to comment on the main points of the code, give well-chosen names to the variables. You'll be happy to have a readable code when you come back to your code again later!

A good computer scientist should check that the parameter passed as an argument verifies the expected hypothesis. For example if `func(x)` is defined for an integer x and the user transmits a string, then a nice warning message should be sent to the user without causing the whole program to fail. The commands `assert`, `try/except` allow to manage this kind of problem. For our part, we will refrain from these checks assuming that the user/programmer uses the functions and variables in good intelligence.

5. Arithmetic – While loop – I

We will only use `Python3`. If you don't know which version you have, type `7/2`: `Python3` returns `3.5` while `Python2` returns `3` (in version 2, `Python` considered that the division of two integers should return an integer).
With `Python3` it is clearer, `a/b` is the usual division (real numbers) while `a//b` is the Euclidean division between two integers and returns the quotient.

6. Strings – Analysis of a text

Handling strings allows you to do fun activities and leave the mathematical world a little bit. You can code quizzes, programs that chat with the user. . . Strings are especially a good introduction to the notion of list, which is an essential tool later on.

7. Lists I

`Python` is particularly flexible and agile for using lists. "Old" languages often only allowed lists containing a single type of element, and to browse a list you always had to do this:

```
for i in range(len(mylist)):
    print(mylist[i])
```

While:

```
for element in mylist:
    print(element)
```

is much more natural.
Sorting a list is a fundamental operation in computer science. Would you imagine a dictionary containing 60 000 words, but not sorted in alphabetical order? Sorting a list is a difficult operation, there are many sorting algorithms. The bubble sort algorithm presented here is one of the simplest. Programming faster algorithms in high school is a great challenge: you have to understand recursivity and the notion of complexity.

8. Statistics – Data visualization

If all readers can program the calculations of sum, mean, standard deviation, median. . . and their visualization, then the objective of this book is achieved! This demonstrates a good understanding of basic mathematical and computer tools.
The `tkinter` module allows a graphic display. To begin with, you have to copy lines of code that works without asking yourself too many questions, then adapt it to your needs.

9. Files

The files make it possible to communicate `Python` with the outside world: for example, you can retrieve a file of grades to calculate the averages and produce a report card for each student.

The activities on the images are nice and these image formats will be used later, even if this format has the big disadvantage of producing large files. Nevertheless they are standard and recognized by image software (*Gimp* for example). Note that some software write files with only one data per line (or all data on a single line). It is a good exercise to implement the reading of all possible formats.

10. Arithmetic – While loop – II

Arithmetic in general and prime numbers in particular are very important in computer science. They are the foundation of modern cryptography and ensure the security of transactions on the Internet. The algorithms presented here are of course elementary. There are sophisticated techniques to say in a few seconds if a number of several hundred digits is prime or not. However, factoring a large integer remains a difficult problem. A computer shows its power when it handles a large quantity of numbers or very large numbers. With arithmetic, we have both at the same time!

11. Binary I

The first difficulty with binary notation is to make a good distinction between a number and writing the number. We are so used to decimal numeral system that we forgot its origin, 1234 is just $1 \times 1000 + 2 \times 100 + 3 \times 10 + 4 \times 1$. As computers work with 0 and 1 you have to be comfortable with binary numeral system. The transition to binary notation is not very difficult, but it is still better to make a few examples by hand before starting the programming.

Binary writing will be used for other problems on the following principle: you have 4 switches so 1.0.0.1 means that you activate the first and last switch and not the others.

12. Lists II

Lists are so useful that `Python` has a whole efficient syntax to manage them: list slicing and list comprehension. We could of course do without it (which is the case for most other languages) but it would be a pity. We will also need lists of lists and in particular arrays to display beautiful images.

13. Binary II

There are 10 kinds of people, those who understand binary and others!

14. Probabilities - Parrondo's paradox

Here is a project with some probabilities and a nice paradox, it's surprising (like all paradoxes) and have been recently discovered. This activity is based on the article "Parrondo's paradox" by Hélène Davaux (*La gazette des mathématiciens*, 2017).

15. Find and replace

Searching and replacing are two actions so common that you don't always realize how strong they are. First of all, it is a good exercise to program the search and replacement operations yourself. There is a sophisticated version of these operations called regular expressions (*regex*). It is a language in itself, very powerful, but a little esoteric.

There is also the mathematics of "find/replace"! The proposed activities are examples that illustrate the main theorem of the article *A complete characterization of termination of* $0^p 1^q \to 1^r 0^s$, by H. Zantema and A. Geser (*AAECC,* 2000).

16. Polish calculator – Stacks

Polish notation (the exact name is "reverse Polish notation") is another way to write algebraic operations. Once again, the hardest part is to adapt your brain to this change in writing. This allows operations (and their priorities) to be seen in a new light. Another interesting vision is to see an operation in the form of a binary tree:
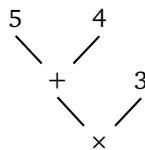
which represents $(5 + 4) \times 3$ or in Polish notation 5 4 + 3 ×.

We tried to postpone as much as possible the change of a global variable in a function, but here this is natural. The use of `global` should be avoided in general.

The notion of stack is a very simple way to structure and access data. However, this is the right way to manage an expression with brackets. The analogy with the sorting station should be illuminating. We will encounter stacks again in the activity on L-systems.

A stack works on the principle of "first in, last out" (*filo*). Another possible data management is on the principle "first in, first out" (*fifo*) as in a queue.

17. Text viewer – Markdown

The purpose of this chapter is twofold: to discover *Markdown* which is a very practical language for formatting a text, but also to understand the justification of a paragraph.

Obviously the *Markdown* language has more tags than those presented here. We could continue the project by doing the justification with fonts of different sizes and shapes, or even create a complete small word processor.

18. L-systems

We come back to the theme "find/replace" but this time with a geometric vision. The figures obtained are nice, easy to program using the turtle, but the most beautiful is to see in live the layout from L-systems. For L-systems defined by expressions containing brackets, we find again the notion of stacks.

The formulas are taken from the book *The algorithmic beauty of plants*, by P. Prusinkiewicz and A. Lindenmayer (Springer-Verlag, 2004) with free access: *The algorithmic beauty of plants*.

The illustrations that begin each part of this book are iterations of the L-system called the Hilbert curve and defined by:

```
start = "X"  rule1 = ("X","lYArXAXrAYl")  rule2 = ("Y","rXAlYAYlAXr")
```

19. Dynamic images

This activity is based on the article "Blurred images, recovered images" by Jean-Paul Delahaye and Philippe Mathieu (*Pour la science*, 1997). This article also looks at the calculation of the number of iterations required to find the original image. The underlying mathematical notion is that of *permutation*: a one-to-one transformation of a finite set (here the set of pixels) into itself.

20. Game of life

A great classic of fun computing! There are dozens of sites on the Internet with configurations having incredible properties and many other ideas. But the most fascinating thing is that extremely simple rules lead to complex behaviors that resemble the life and death of cells.

21. Ramsey graphs and combinatorics

Graphs are very common objects in mathematics and computer science. The problem presented here is simple and fun. But what we should remember from this chapter is how the difficulty increases with the number of vertices. Here we do the calculations up to 6 vertices and we can't go much further with our exhaustive verification method.

A great mathematician Paul Erdös stated that if aliens landed on Earth threatening to destroy our planet unless we could solve the problem of 5 friends/5 foreigners, then by mobilizing all the computers and mathematicians in the world we would manage to get by (we know that the answer is between 43 and 48 people). On the other hand, if the aliens asked us to solve the problem for 6 friends/6 foreigners, then the easiest thing would be to prepare for war!

22. Bitcoin

Before investing all your savings in bitcoins it is better to understand how it works! The activities presented here are intended to present a (very) simplified version of the blockchain that is the basis of this virtual currency. The principle of the blockchain and the proof of work are not so complicated.

23. Random blocks

These random constructions are first of all computer recreations that produce pretty figures, all similar but all different. But they are also the subject of modern and difficult mathematical work. Martin Hairer won the Fields Medal in 2014 for studying the upper boundary of our falling blocks, whose shape is governed by an equation, called "KPZ equation". A good bonus activity would be to drop blocks of the game *Tetris* instead of small squares.

Thanks

I would like to thank Stéphanie Bodin for her encouragement and for testing the activities of the first parts. I would like to thank Michel Bodin for having tested all the first activities and his proofreading. Thanks to François Recher for his enthusiasm and careful review of the book. Thanks to Éric Wegrzynowski for his review and ideas for the activities on the images. Thanks to Philippe Marquet for his review of the first chapters. Thanks to Kroum Tzanev for the model of the book. Thanks to Emily Gubski for the English corrections.

You can retrieve all the activity codes `Python` and all the source files on the Exo7 *GitHub* page: "GitHub: Python in high school".

Index