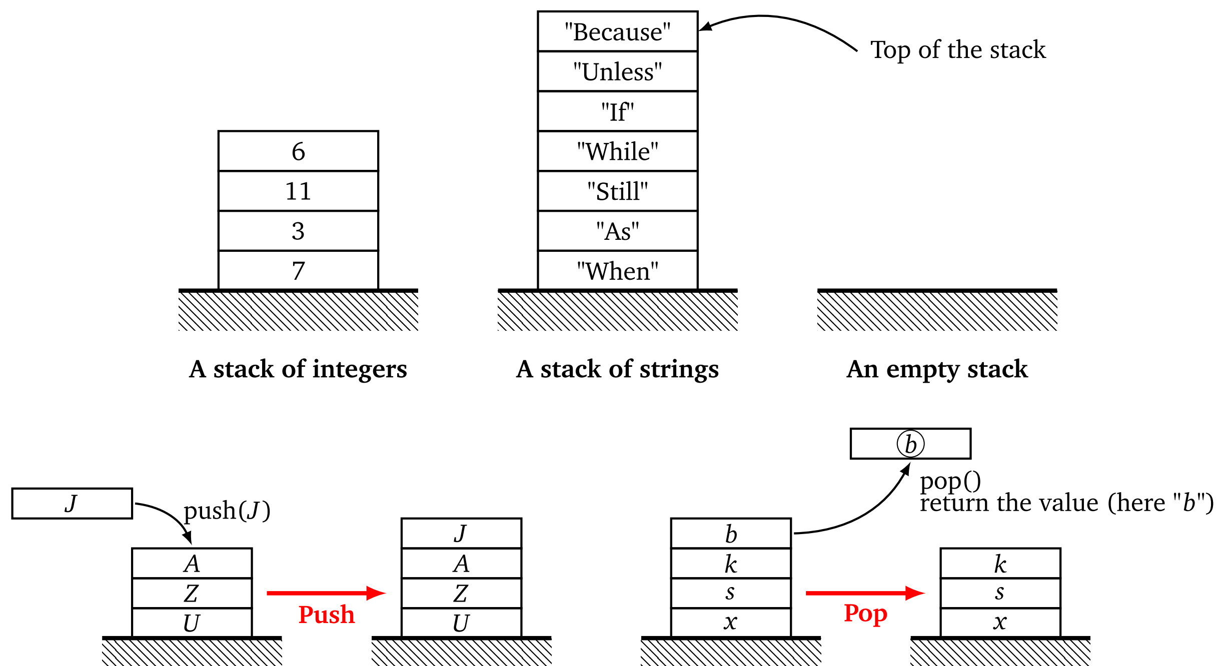
Polish calculator – Stacks

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.



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:

```
gravitation = 9.81
```

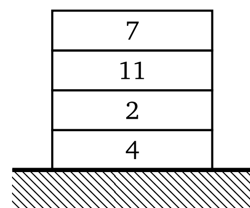
The content of the variable `gravitation` is now available everywhere. On the other hand, if you want to change the value of this variable in a function, you must specify to `Python` that you are aware of modifying a global variable.
For example, for calculations on the Moon, it is necessary to change the gravitational constant, which is much lower there.

```
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.



`stack = [4,2,11,7]`

A stack **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:

```
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.'`

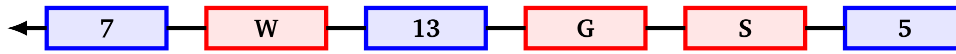
3. The `isdigit()` function is a `Python` method that tests if a string contains only numbers. This allows you to test if a string corresponds to a positive integer. Here are some examples: `"1776".isdigit()` returns `True`; `"Hello".isdigit()` returns `False`.
Remember that you can convert a string into an integer by the `int(string)` command. The following small program tests if a string can be converted into a positive integer:

```
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)**.**

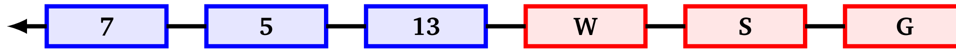
Goal: solve a sorting problem by modeling a storage area by a stack.

A train has blue wagons with a number and red wagons with a letter.



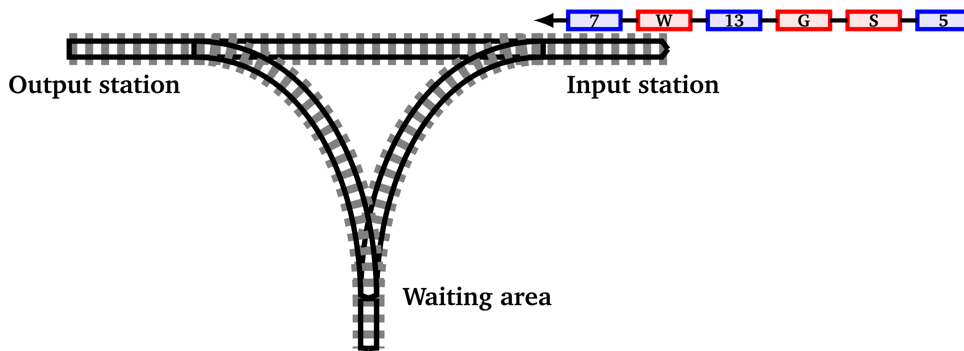
Unsorted train.

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).



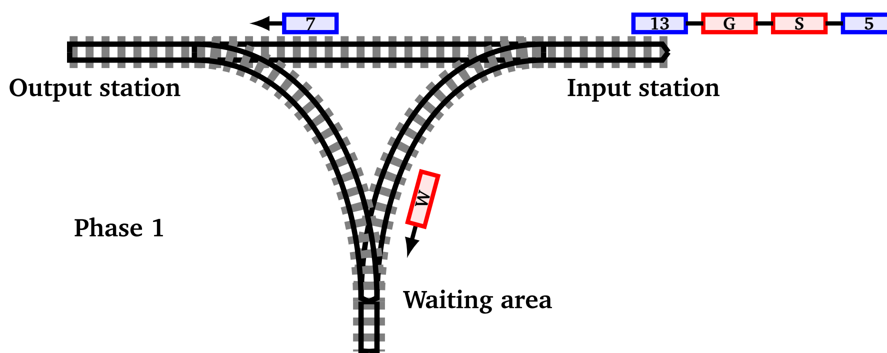
Sorted train.

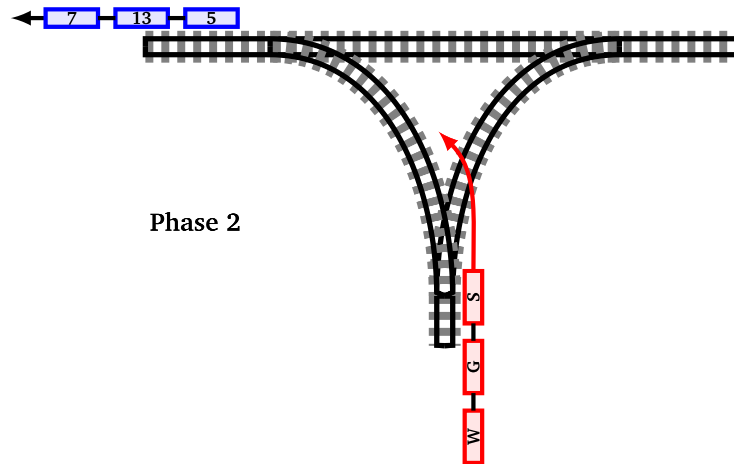
For this purpose, there is an output station and a waiting area: a wagon can either be sent directly to the output station or temporarily stored in the waiting area.



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.





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!

Classic: $7+6$ Polish: $7\ 6\ +$

Other examples:

- Classic: $(10+5)\times 3$; Polish: $10\ 5\ +\ 3\ \times$
- Classic: $10+2\times 3$; Polish: $10\ 2\ 3\ \times\ +$
- Classic: $(2+8)\times(6+11)$; Polish: $2\ 8\ +\ 6\ 11\ +\ \times$

Let's see how to calculate the value of an expression in Polish notation.

- We read the expression from left to right:

$$\overrightarrow{2\ 8\ +\ 6\ 11\ +\ \times}$$

- When you meet a first operator ($+$, $\times$, ...) 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:

$$\underbrace{10}_{\text{result of } 2+8}\ 6\ 11\ +\ \times$$

- We continue reading the expression (we are looking for the first operator and the two terms just before):

$$10\ \underbrace{6\ 11\ +}_{6+11=17}\ \times \quad \text{becomes} \quad 10\ 17\ \times \quad \text{that is equal to} \quad 170$$

- At the end there is only one value left, it's the result! (Here 170.)

Other examples:

- $8\ 2\ \div\ 3\ \times\ 7\ +$

$$\underbrace{8\ 2\ \div}_{8\div 2=4}\ 3\ \times\ 7\ + \quad \text{becomes} \quad \underbrace{4\ 3\ \times}_{4\times 3=12}\ 7\ + \quad \text{becomes} \quad 12\ 7\ + \quad \text{that is equal to} \quad 19$$

- $11\ 9\ 4\ 3\ +\ -\ \times$

$$11\ 9\ \underbrace{4\ 3\ +}_{4+3=7}\ -\ \times \quad \text{becomes} \quad 11\ \underbrace{9\ 7\ -}_{9-7=2}\ \times \quad \text{becomes} \quad 11\ 2\ \times \quad \text{that is equal to} \quad 22$$

Exercise. Compute the value of the expressions:

- $13\ 5\ +\ 3\ \times$
- $3\ 5\ 7\ \times\ +$
- $3\ 5\ 7\ +\ \times$
- $15\ 5\ \div\ 4\ 12\ +\ \times$

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 “$(13+5)\times 7$” 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, $2+3\times5$ means $2+(3\times5)$ and not $(2+3)\times5$. 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
  `"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!