Informatique II

Langages - & - Syntaxe Python

Objectifs :

- Connaître les structures de base d'un langage [condition, itération, fonction etc..].
- Savoir manipuler les types de données primitifs en Python.

Introduction

Ordinateur :

Machine qui réalise des commandes très simples dans un ordre bien déterminé.

Intérêt double :

- Il fait la partie laborieuse du travail à notre place
- Il le fait très vite et sans erreur.

- Nous sommes responsable de la partie conception.
- Toute erreur nous est donc imputable.

COMPUTER IS A DUMBBELL

vs.

errare humanum est

Objectif de la programmation :

Il s'agit d'écrire dans un **langage de programmation** la suite d'opérations qui seront réalisées par le processeur. (unité logique, unité de calcul, accès mémoire, ...)

Il existe des milliers de langages :
Le processeur ne comprend que le langage machine compilé en binaire ...
... 100101110100l... etc

Chaque programme doit donc être traduit par un compilateur en fonction :
- du langage utilisé pour rédiger le code.
- du processeur qui va exécuter le programme.

Ecrire du CODE est donc un exercice d'écriture, et de communication :
- le programme est un simple document textuel (code).
- il doit pouvoir être relu/modifié/transformé rapidement par un tiers.

=> les instructions du code seules ne suffisent pas !
- Un code doit être commenté
- Il doit être rédigé selon une forme claire bien spécifique (indentation)

Qu'est-ce qui caractérise un langage de programmation ?

Ex : calcul des nombres premiers en Python

Crible d'Eratosthène On élimine tous les non-premiers

```
# calcul des nombres premiers inférieurs à N
# initialisation
#!/usr/bin/python
# -*- coding: utf-8 -*-
N       = 200
liste   = range(2, N)                           # liste de 2 à N
nombre = 2
while nombre*nombre <= N:                       # tant que le nb premier < à la
                                                # racine carrée de N
    for i in liste[liste.index(nombre) + 1:]: # parcourt la liste à partir de ce nb
        if i % nombre == 0:                     # un multiple du nombre est trouvé
            liste.remove(i)                     # ou "del liste[ liste.index(i) ]" :
                                                # on le raye de la liste
    nombre = liste[liste.index(nombre) + 1]   # on prend le nombre suivant non rayé

print liste                                     # affichage du résultat
```

Autre langage : C

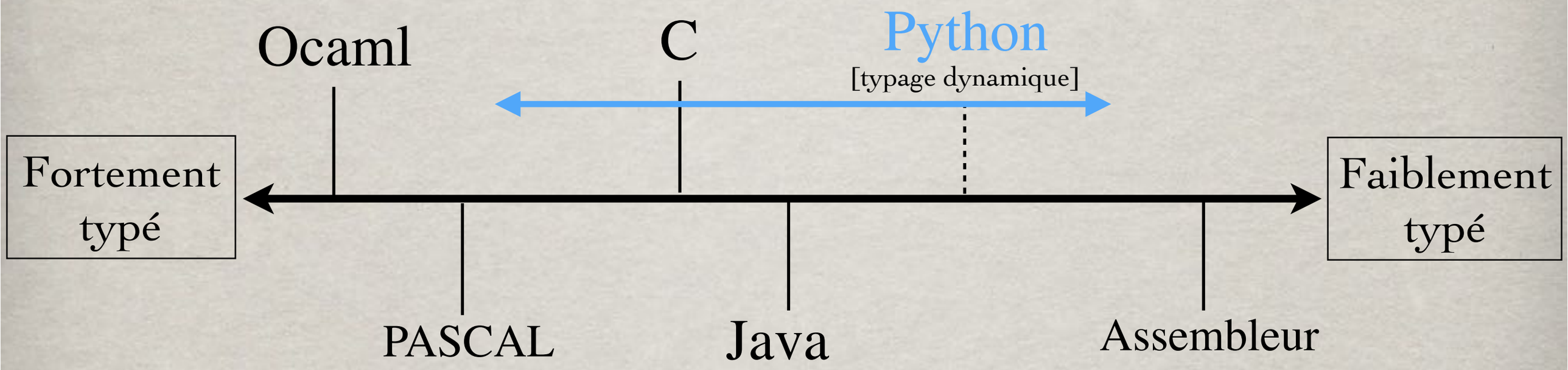
- Chaque variable est définie au préalable
  [avec un type défini une fois pour toute]

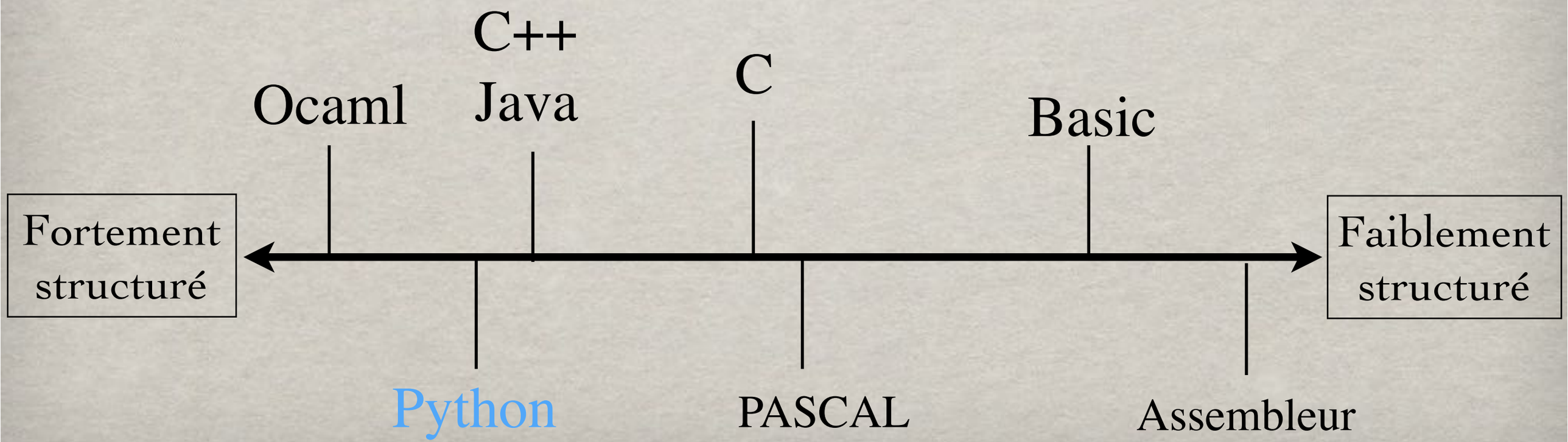
- La syntaxe n'est pas la même, on retrouve les mêmes idées mais l'algorithme n'est pas exactement celui du crible d'Eratosthène

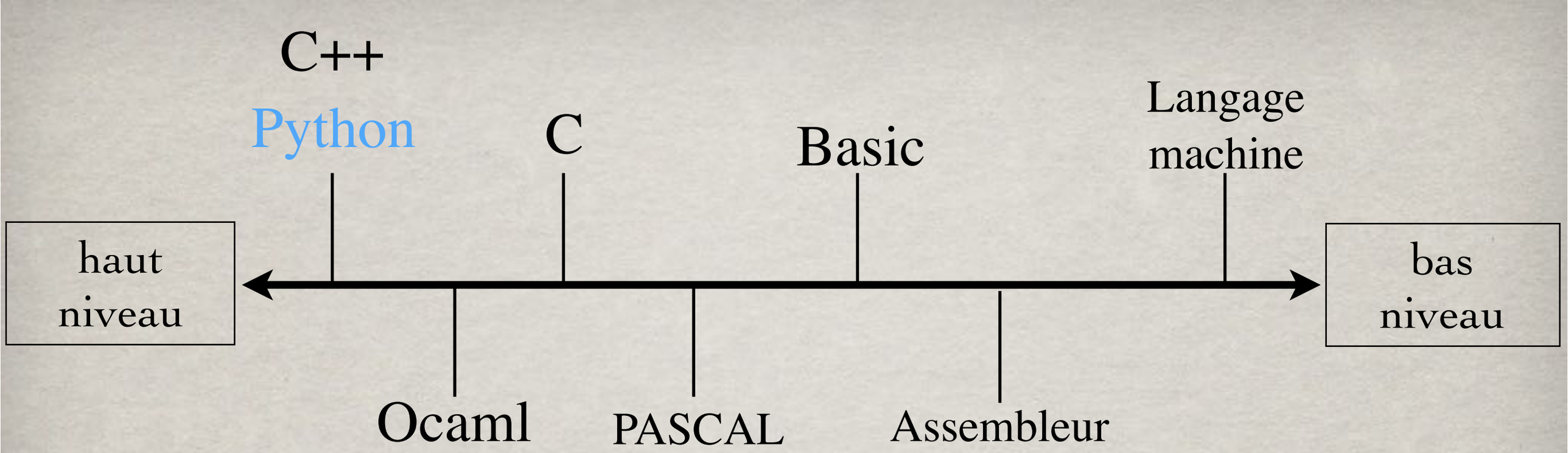
On sélectionne tous les nombres premiers

```c
#include <stdio.h>
#include <time.h>

//*Nombres Premiers*//

int main (void){

  long N,i,j;
  short P;

  printf("Enter a number : ");
  scanf("%ld",&N);

  for (i=2;i<N;i++){
    P=1;
    for (j=2;j<i;j++){
      if (i%j==0){
        P=0;
        break;
      }
    }
        if(P==1){
        printf("%7ld",i);
        }
  }


}
```

Quelle(s) hiérarchie(s) dans les langages ?







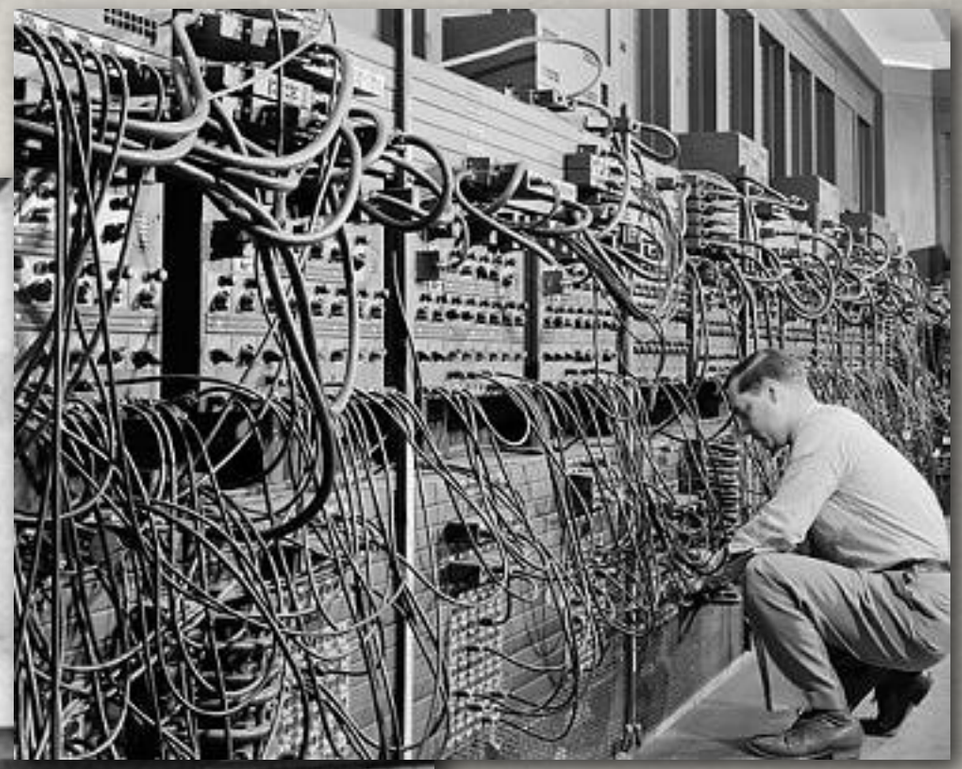
Haut niveau :

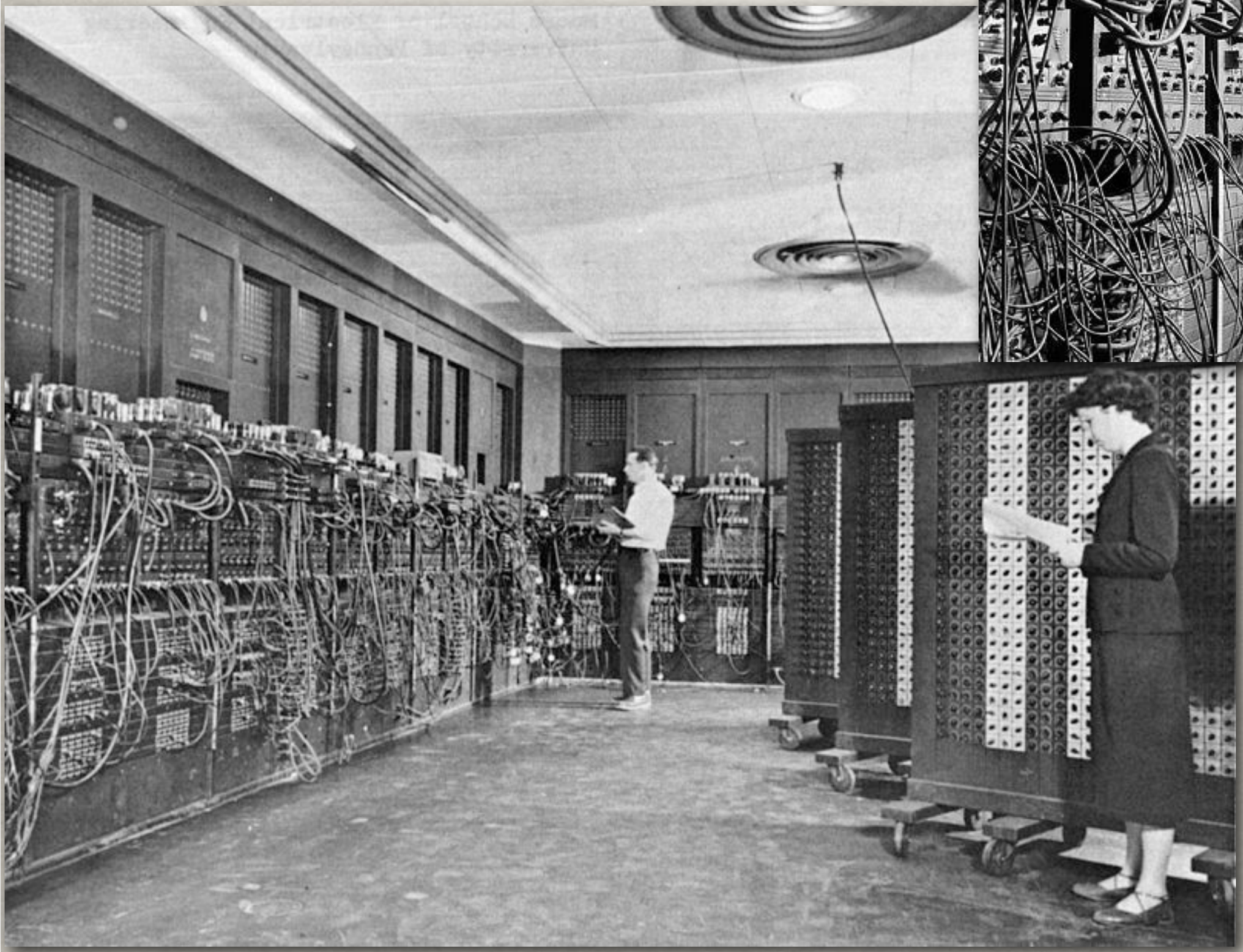
abstrait, proche de notre façon de penser => on ne sait rien de la couche matérielle
=> on ne sait pas ce que fait le processeur

Bas niveau :

Concret, très «mécanique» => on doit indiquer chaque opération réalisée par le processeur.

Programmation à fils ....





… ça c'est du concret !

Assembleur (IBM):

- Permet d'écrire du code avec un clavier !!!
- Bas niveau : accès direct aux commandes du processeur

FORTRAN (IBM 1954):

- Toujours très utilisé dans le milieu scientifique pour la qualité de ses bibliothèques numériques
- Compilé => grande rapidité, ce qui en fait le langage informatique ayant eu la plus grande longévité.
- parallélisable : fonctionne sur des super-calculateurs (CERN, IDRIS)

BASIC (1963):

- Premier objectif éducatif : très simple
- Langage interprété : programme rapide après compilation.
- Peu structuré au départ (a été essentiellement abandonné)

PASCAL (~1970) :

- Langage très typé et très structuré
- langage interprété : programme rapide après compilation.
- modulaire : permet de décomposer un projet, de mixer des langages

C (BELL ~ 1970) :

- Créé pour le système d'exploitation UNIX
- Langage typé et très structuré : orienté objet avec C++ (1980).
- langage interprété : programme rapide après compilation.
- modulaire : permet de décomposer un projet, de mixer des langages

Pourquoi Python ?

- C'est gratuit (youpiii !!!!!)

- Multiplateformes (Linux - Mac OSX - Windows)

- Langage interprété : plus pédagogique [prise en main facile]

- Langage orienté objet, typage dynamique [très élaboré]

- Très à la mode (privé-public-écoles ingénieurs-scientifiques)
    => Communauté web très active

Environnements de programmation

cf TD de présentation Python + environnement

Pour commencer : le plus simple possible

→ Python Tutor

Pour écrire un programme simple : un seul fichier

→ Xemacs + Terminal

Pour gérer un projet : logiciel avec interface graphique

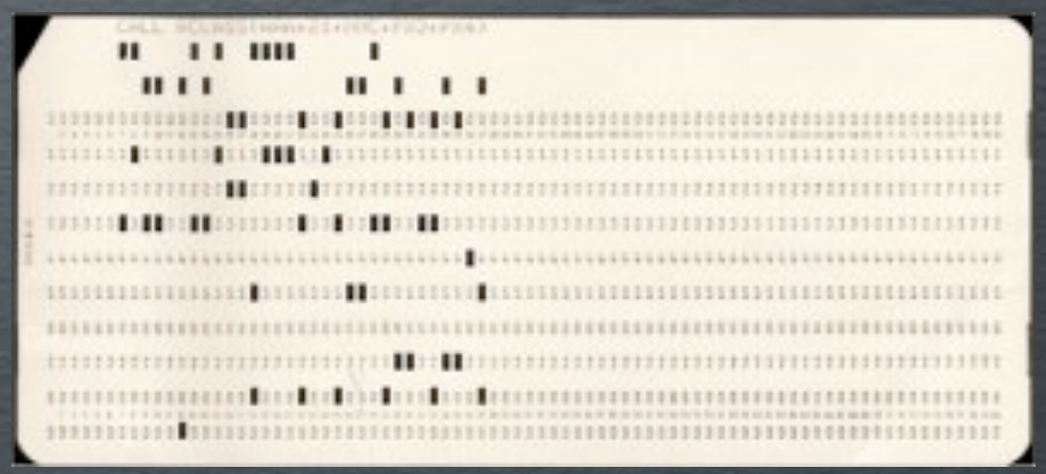
→ Pyzo - Spyder - etc...

(Projets avec bcp.de fichiers, modules, etc..)

Programmation impérative

Trouve son origine dans les mécanographes

--> cartes perforées



Punch Card :

(Carte perforée)

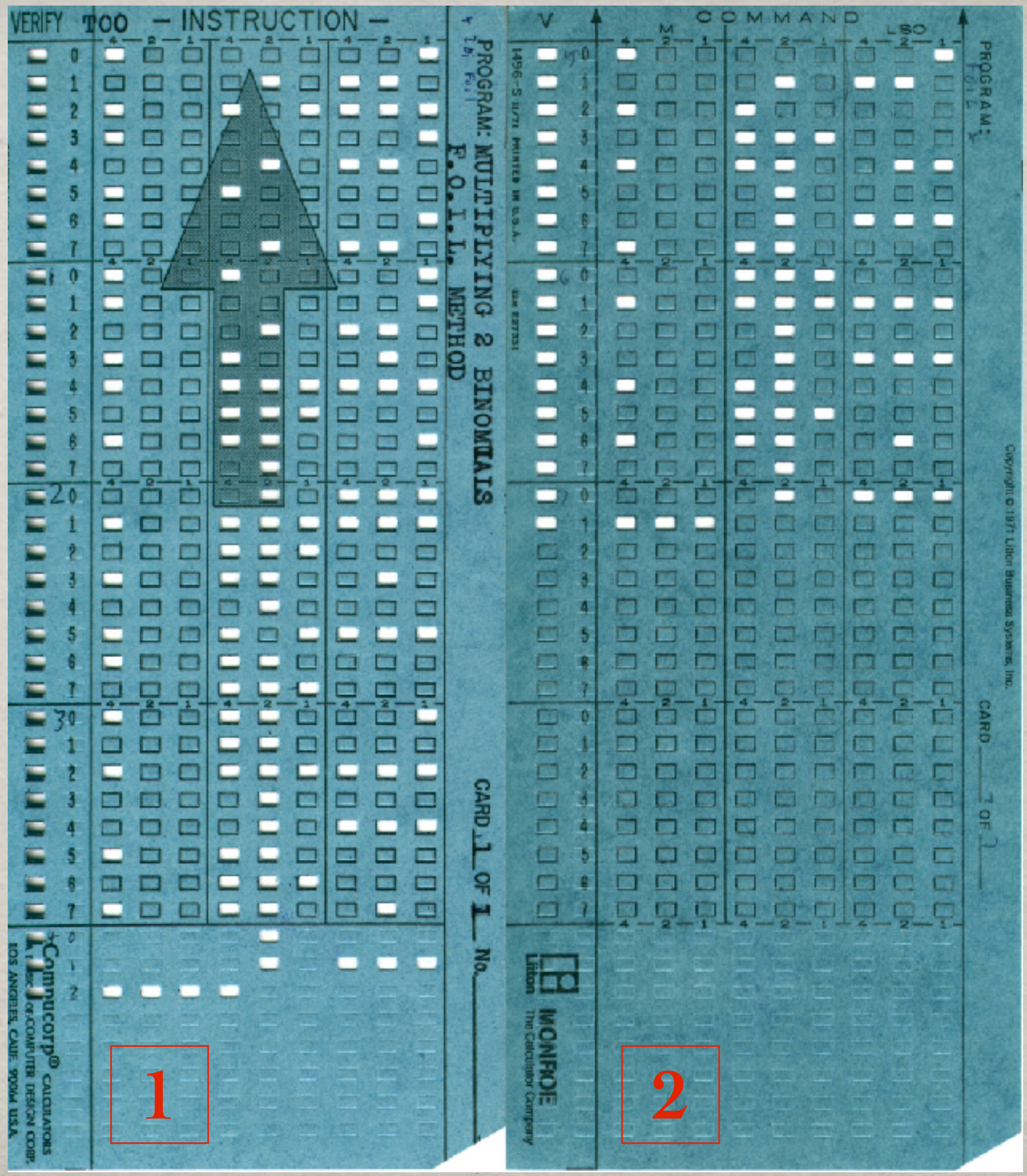
0

10

20

30

40



50

60

70

80

etc

Le processeur ne sait exécuter qu'un nombre limité de commandes simples :

- accès mémoire (cache [rapide], stockage [plus lent])
- opérations de base (+, -, *, /)
- opérations logiques (et, ou, non >, <, ==)

Ces opérations simples sont réalisées les unes à la suite des autres, selon un **compteur ordinal** et à une très grande cadence imposée par **l'horloge**.

On parle de **paradigme de la programmation impérative** par opposition à la programmation structurée ou à la programmation fonctionnelle qui sont possibles dans les langages de haut niveau.

Rq :
Le terme ordinateur provient de cette idée d'une succession de processus simples : on passe des instructions de manière ordinale

Programmation en Python

Objectifs :

- Connaître les structures de base d'un langage [condition, itération, fonction etc..].
- Savoir manipuler les types de données primitifs en Python.

1 - Déclaration des variables, notion de type primitif

Dans les langages de bas niveau (Fortran, C), toutes les variables doivent être déclarées, au début du programme ou au début de chaque fonction, de chaque procédure.

Cette étape permet de réserver un emplacement mémoire où la valeur de la variable sera stockée. L'emplacement mémoire ainsi que sa taille dépend de la nature de la variable, que l'on nomme son «type» [ou par extension sa «classe» en programmation objet].

ex : - BOOLEEN : BOOL -> vrai ou faux [1 bit]

- INTEGER : INT -> Un entier long sera stocké en 32 bits [4 octets]
  LONG
- FLOAT : FLOAT -> simple précision en 32 bits [4 octets]
  DOUBLE -> double précision en 64 bits [8 octets]

Dans les langages de plus haut niveau, Python, Ocaml, cette déclaration est automatisée.
La variable prend automatiquement le type de la valeur qu'on lui affecte.
La variable change de type et d'identité objet si on lui affecte une valeur de type différent.

Fortran/ C / Pascal : typage +/- strict
Python : typage dynamique (une même variable peut changer de type)
Ocaml : typage très strict

La programmation Python, sans «garde fou», très souple et agréable est toutefois très risquée.
[Penser à ne pas monter dans une nouvelle fusée programmée en Python....]

factorielle.c

```c
#include <stdio.h>
//* Fonction factorielle en C *//

long factorielle(long n)
{
  long resultat = 1;
  unsigned long i;

  if (n < 0)
    return -1;
  for (i=1; i < n+1; ++i)
    resultat *= i;
  return resultat;
}

int main (void)
{
  long RES;
  long N;

  printf("Enter a number : ");
  scanf("%ld",&N);

  RES = factorielle(N);
  printf("%ld",N); printf("! = ");
  printf("%ld", RES,"\n");
  printf("  \n");
}
```

Déclaration des variables

```
$ gcc factorielle.c
$ ./a.out
Enter a number : 17
17! = 355687428096000
```

factorielle.py

```python
#fonction factorielle en python

def facto(n):
    if n < 0:
        return -1
    P=1
    for i in range(1,n+1):
        P*=i
    return P


N=int(input("Enter a number :\n"))
print("{}! = {}".format(N, facto(N)))
```

```
$ python3 facto.py
Enter a number :
17
17! = 355687428096000
```

Rq :

factorielle en version itérative

2 - Les types de données primitives en Python

En Python chaque variable possède un **type** et une **identité** :

On parle de **class** en programmation orientée objet :

- Entier 'int' (pas de limite autre que la mémoire disponible)
- Réel 'float' (64 bits = s1+ m52 + e11),
- Complexe
- Booléen 'bool'
- liste 'list'
- Chaîne 'str'
- t-uplet 'tuple'
- fichier
- dictionnaire
- Il y en a beaucoup, et on peut en inventer d'autres

On peut de plus créer de toutes pièces de nouvelles class's ou types de données, ainsi que leur propriétés, filiations, etc...

A - Les types scalaires :

```
1  a=3
2  print('val :', a,'\nid :', id(a),'\ntype :' , type(a))
3  print()
4
5  b=12345678901234567890123.3456
6  print('val :', b,'\nid :', id(b),'\ntype :' , type(b))
7  print()
8
9  z=complex(1,1)
10 print('val :', z,'\nid :', id(z),'\ntype :' , type(z))
11 print(abs(z))
12 print(complex.conjugate(z))
```

```
val : 3
id : 1824352
type : <class 'int'>

val : 1.2345678901234568e+21
id : 7993888
type : <class 'float'>

val : (1+1j)
id : 381709920
type : <class 'complex'>
1.4142135623730951
(1-1j)
```

Chaque objet possède une **valeur** [désignée par son nom], une **identité** [id(nom)], et un type ou une class **type**[(nom)]

Le code

les variables dans le code

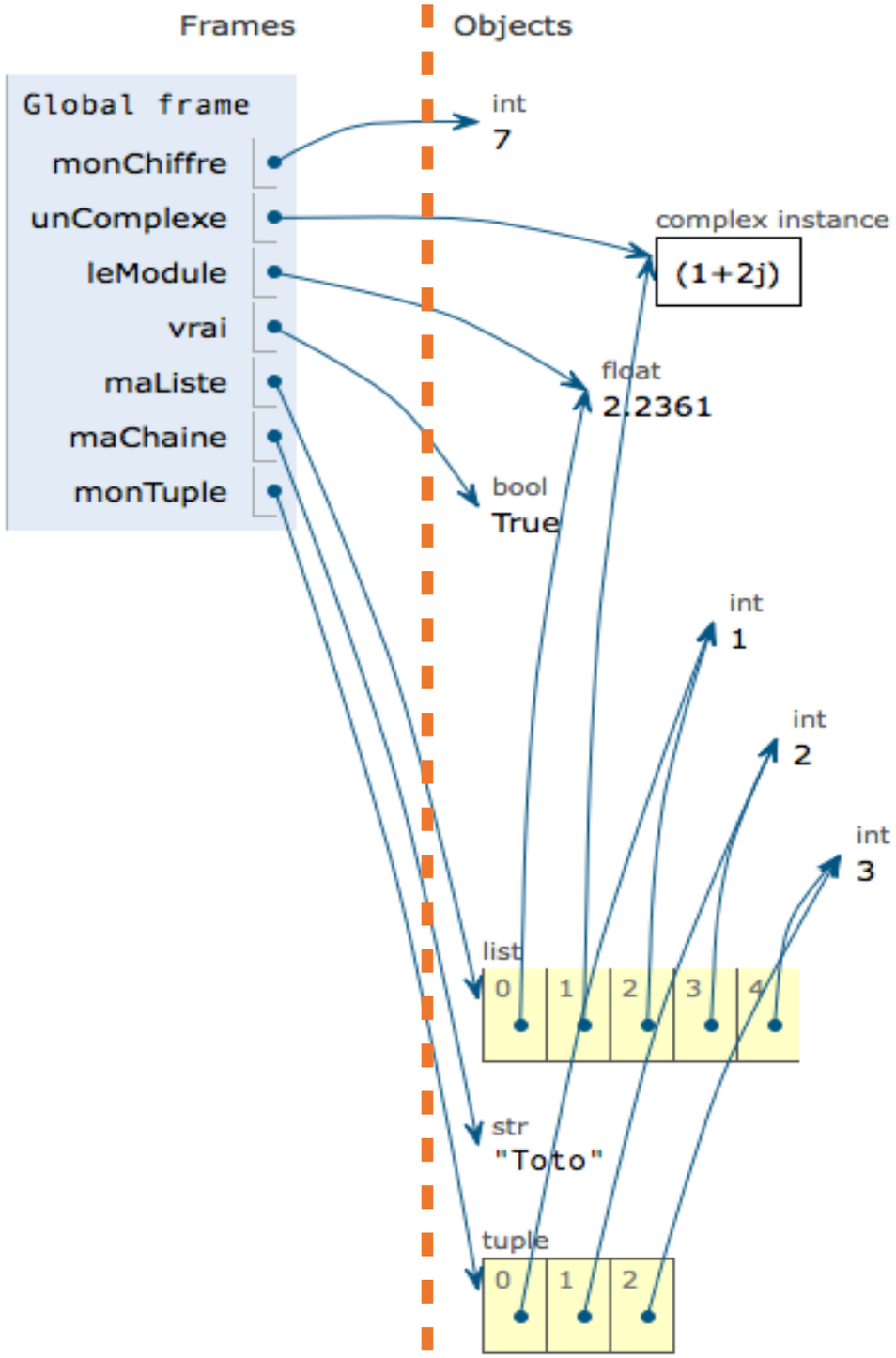
L'espace objet => en mémoire

Write code in Python 3.3 (drag lower right corner to resize code editor)

```
1  monChiffre = 7  #porte bonheur
2
3  unComplexe = 1 + 2J #nombre complexe
4
5  leModule = abs(unComplexe) #son module
6
7  #un booléen :
8  vrai = (leModule==5**0.5)
9
10 maListe=[leModule, unComplexe, 1, 2, 3]
11
12 maChaine="Toto"
13 monTuple=(1,2,3)
```

line that has just executed
next line to execute

<< First | < Back | Done running (7 steps) | Forward > | Last >>



Exemple : échange de valeurs

Write code in Python 2.7

(drag lower right corner to resize code editor)

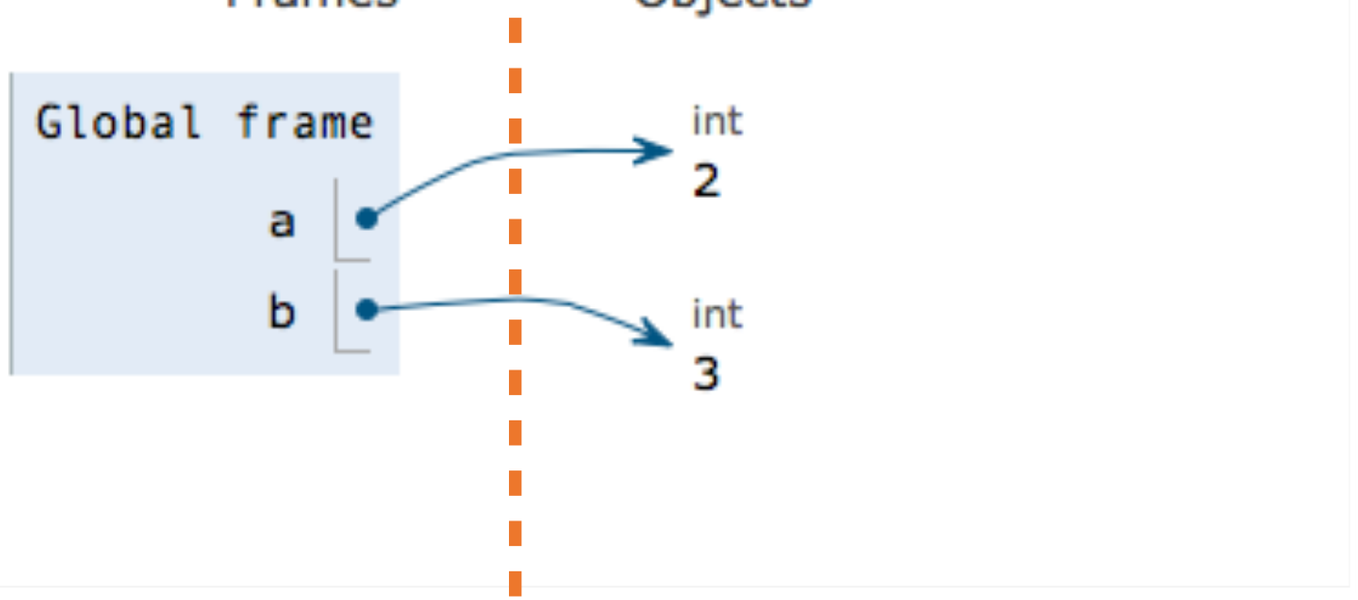
```
1  a=2
2  b=3
3
4  print(id(a), id(b))
5
6  (a,b) = (b,a)
7
8  print(id(a), id(b))
9  del(b)
```

line that has just executed

next line to execute

Print output (drag lower right corner to resize)

```
(15442240, 15442216)
```



Exemple : échange de valeurs

Write code in Python 2.7

(drag lower right corner to resize code editor)

```
1  a=2
2  b=3
3
4  print(id(a), id(b))
5
6  (a,b) = (b,a)
7
8  print(id(a), id(b))
9  del(b)
```

line that has just executed

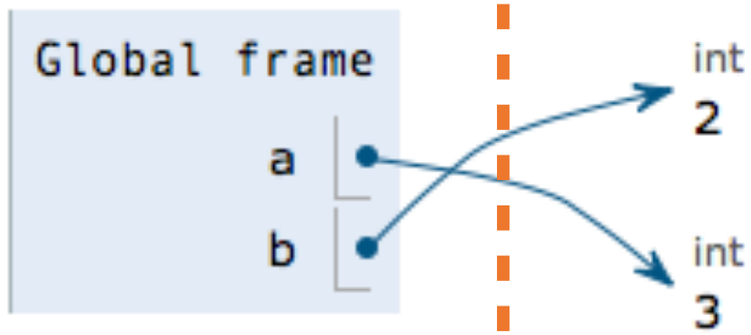
next line to execute

Print output (drag lower right corner to resize)

```
(15442240, 15442216)
(15442216, 15442240)
```

Frames

Objects



Exemple : échange de valeurs

Write code in Python 2.7

(drag lower right corner to resize code editor)

```
1  a=2
2  b=3
3
4  print(id(a), id(b))
5
6  (a,b) = (b,a)
7
8  print(id(a), id(b))
9  del(b)
```

line that has just executed

next line to execute

Print output (drag lower right corner to resize)

```
(15442240, 15442216)
(15442216, 15442240)
```

Frames

Global frame

a

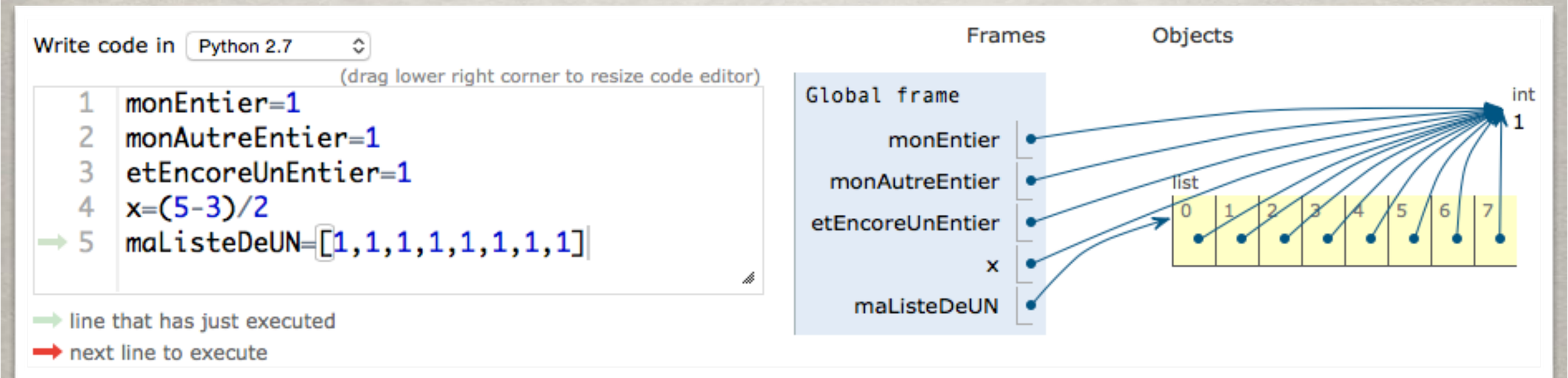
Objects

int

3

2 n'est plus «pointé» => il est supprimé de la mémoire

L'intérêt premier de la représentation objet est d'éviter des duplications inutiles en mémoire :



On voit ici que l'entier `1` est pointé par beaucoup de variables.
Toutefois il n'est mis en mémoire qu'une seule fois !

S'ajoute à cela une gestion plus efficace de la mémoire : un objet qui n'est plus « pointé » est automatiquement effacé.

Opérations de base :

Outre les quatre opérations élémentaires (+, -, x, /) le processeur renvoie le quotient et le reste dans la division euclidienne :

// : renvoie le quotient
/ : renvoie le résultat le FLOAT de la division [inexact : 16 chiffres]
% : opérateur modulo, renvoie le reste de la division euclidienne
** : opération puissance, ou pow(a, b)

```
#Notations python3

>>> 7 // 3, 7 % 3
(2, 1)

>>> 7/3
2.3333333333333335

>>> divmod(7,3)
(2, 1)


>>> 2**3, 2.0**3
(8, 8.0)


>>> pow(2, 3), pow(2., 3)
(8, 8.0)
```

Opérations d'incrémentations :

```
#Python 3

>>> i=1 ; print(i)
1
>>> i=i+1 ; print(i)
2
>>> i+=1 ; print(i)
3
```

i = i + 1 ne signifie pas l'égalité mathématique, mais :

1- je calcul i+1,
2- je l'affecte à i

Les écritures (+= ; -= ; *= ; /=) abrègent et généralisent cette écriture.

```
>>> i=1 ; print(i)
1
>>> i=2*i ; print(i)
2
>>> i*=2 ; print(i)
4
>>> i//=2 ; print(i)
2
>>> i/=2 ; print(i)
1.0
```

On peut concevoir l'opération contraire à l'affectation, comme celle consistant à supprimer une variable.

Celle-ci se réalise à l'aide de la commande del()

```
>>> x=4; id(x)
4297273376

>>> del(x)
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

del() ne supprime pas nécessairement l'objet de la mémoire [références partagées]

Exercices pratiques :

Chercher les exercices sur les types de variables et l'affectation

B - Les types logiques :

Il s'agit simplement d'une variable portant la valeur **vrai : 1**
ou **faux : 0**

```
>>> V=True ; bin(V)
'0b1'

>>> F=False ; bin(F)
'0b0'
```

Les tables de la vérité !

OR	True	False
True		
False		

AND	True	False
True		
False		

NOT	True	False

Exemples python :

```
>>> not(True) ; not(False)
False
True
>>> True or False
True
>>>True and False
False
>>> True or (True and False)
True
>>> (True or False) and (False and True)
False
```

Attention : Les parenthèses (algébriques) sont prioritaires sur les opérations logiques.

Rq : Il existe aussi des formes contractées pour les opérateurs logiques

OR : «|» -> «pipe» unicode 2502
AND : «&» -> AMPERSAND
XOR : «^» -> Chapeau chinois ?

Ceux-ci agissent directement sur l'état booléen du bit de valeur

OR

```
>>> for i in [0,1]:
...     for j in [0,1]:
...         print(i|j,end='')
...     print()
...
01
11
```

XOR

```
>>> for i in [0,1]:
...     for j in [0,1]:
...         print(i^j,end='')
...     print()
...
01
10
```

AND

```
>>> for i in [0,1]:
...     for j in [0,1]:
...         print(i&j,end='')
...     print()
...
00
01
```

↳ Affiche les tables de vérité

Rq : Python possède également le **OU_exclusif :** XOR : a^b
Celui-ci vaut False si a et b sont True--> C'est le sens du mot exclusif !

Exercices : a - compléter les tables suivantes

C = NOT(A or B) or (A and B)

A \ B	True	False
True		
False		

C = (A or B) or (A and B)

A \ B	True	False
True		
False		

C = A or B and not(A)

A \ B	True	False
True		
False		

C = (A or B) and not(A)

A \ B	True	False
True		
False		

b - XOR

Proposer une expression logique sur A et B avec les opérateurs or, and et not, pour exprimer la fonction boolèenne Xor

c - Voices down the corridor

Vous vous réveillez ! Vous êtes dans un couloir avec une porte à chaque extrémité gardée chacune par un homme en arme. L'un ment toujours l'autre jamais !
L'une des portes mène au paradis, l'autre en enfer !

Quelle question poser à un gardien pour savoir quelle porte choisir ?

d - A table !

Montrer que cette table de vérité peut s'écrire :

$$f(a,b,c,d) = (a\bar{b} + \bar{a}b)(c\bar{d} + \bar{c}d)$$

où la barre signifie «not» , + «or» et la multiplication «and».

A vérifier en python...

$f(a,b,c,d)$

f		0	1	0	1	a
		0	0	1	1	b
0	0	0	0	0	0	
1	0	0	1	1	0	
0	1	0	1	1	0	
1	1	0	0	0	0	
c	d					

```
>>> for i in [False, True]:
...     for j in [False, True]:
...         for k in [False, True]:
...             for l in [False, True]:
...                 print((l and not(k) or not(l) and k) and (j and not(i) or not(j) and i),end=" ")
...         print("")
...
False False False False
False True True False
False True True False
False False False False
```

Exercices pratiques :

Chercher les exercices sur le type booléen

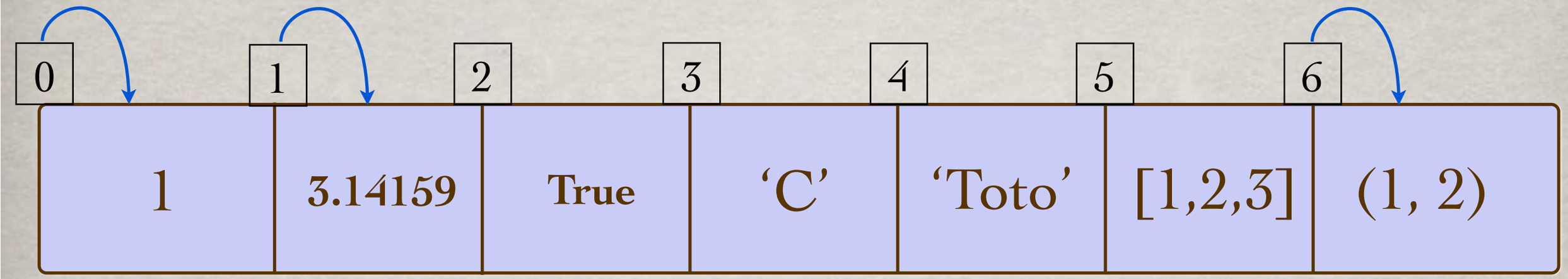
C - Les types séquentiels : Listes et chaînes, tuple

L'un des objets les plus utiles en python est la liste. C'est un objet à part entière avec son propre identifiant, mais qui contient d'autres objets, indexés de 0 à N-1 où N est la longueur de la liste.

```
>>> liste1=[1,2,3,4,5]
>>> liste2['a','b','c','d','e']
>>> liste3=[1, 3.14159, True, 'C', 'Toto', [1, 2, 3], (1, 2)]
>>>
>>> id(liste1),liste1   ;   id(liste2),liste2   ;   id(liste3),liste3;
(4301800872, [1, 2, 3, 4, 5])
(4301671528, ['a', 'b', 'c', 'd', 'e'])
(4301801016, [1, 3.14159, True, 'C', 'Toto', [1, 2, 3], (1, 2)])
>>>
>>> type(liste1) , type(liste2) , type(liste3);
(<class 'list'>, <class 'list'>, <class 'list'>)

>>> for i in [0 ,1 ,2 ,3 ,4 ,5 ,6]:
...       print("liste3[{}]: {} : {} :    {}".format(i,id(liste3[i]), type(liste3[i]), str(liste3[i])))
...
liste3[0]: 4297273280 : <class 'int'> :    1
liste3[1]: 4298486360 : <class 'float'> :    3.14159
liste3[2]: 4296864096 : <class 'bool'> :    True
liste3[3]: 4300983632 : <class 'str'> :    C
liste3[4]: 4302040624 : <class 'str'> :    Toto
liste3[5]: 4301684608 : <class 'list'> :    [1, 2, 3]
liste3[6]: 4301730088 : <class 'tuple'> :    (1, 2)
```

- Les éléments de la liste ont chacun leur propre indice et peuvent être de n'importe quel type.
- L'intérêt principal des listes est qu'on peut créer un ITÉRATEUR qui passe en revue toute la liste.

La liste a une longueur N = len(liste) mais attention **l'indexation commence à 0** :



Chaque objet peut être extrait de la liste grâce à son indice i : $0 \leq i \leq N-1$

```
>>> len(liste3)
7

>>> liste3[1]
3.14159

>>> liste3[2]
True

>>> liste3[7]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Syntaxe python :

Objet_séquentiel[indice]

Attention à ne pas chercher l'élément de la N-ième case -> elle n'existe pas !

Si un objet d'indice i de la liste est lui même séquentiel, on peut alors chercher son j-ième élément grâce la même syntaxe :

objet_séquentiel[i][j]

En effet, l'élément objet_séquentiel[i] est de type 'list'.
On peut donc rechercher ses éléments.

```
>>> objet=liste3[5] ; type(objet)
<class 'list'>
>>> objet
[1, 2, 3]

>>> objet[2], liste3[5][2]
(3,3)

>>> liste3[5][2] is objet[2]
True
```

Les objets séquentiels sont les listes, les chaînes et les tuples.

Attention : on ne peut toutefois pas modifier directement les éléments d'une chaines ou d'un tuple, contrairement aux listes.

Astuce :

[On convertie la chaîne en liste,
avec la commande **list( )**]

```
>>> liste3[5] , type(liste3[5]) ; liste3[5][0]
([1, 2, 3], <class 'list'>)
1

>>> liste3[6] , type(liste3[6]) ; liste3[6][1]
((1, 2), <class 'tuple'>)
2

>>> liste3[4] , type(liste3[4]) ; liste3[4][2]
('Toto', <class 'str'>)
't'
```

Listes et le saucisson :

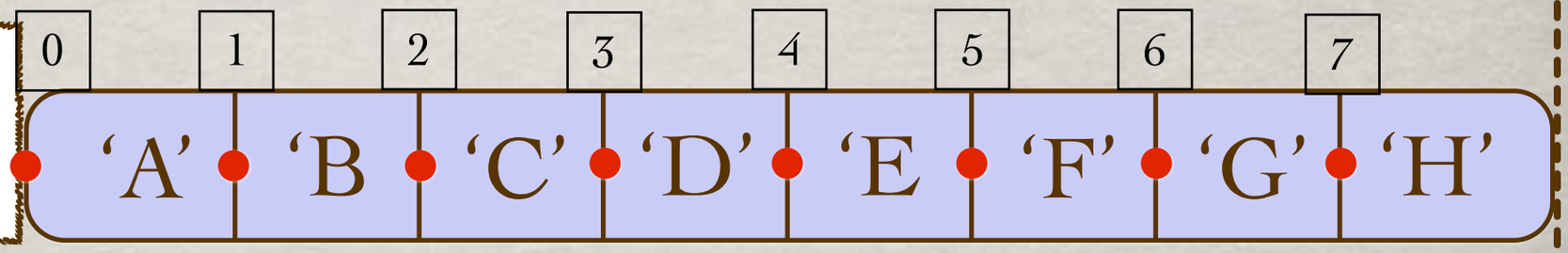
On peut trancher, une liste partielle de la liste de départ,
-> pour l'afficher, pour l'affecter à une nouvelle variable etc...

Les indices représentent en fait les positions des coupes **liste[i:j]** avec i<j
Je tranche en i et en j :

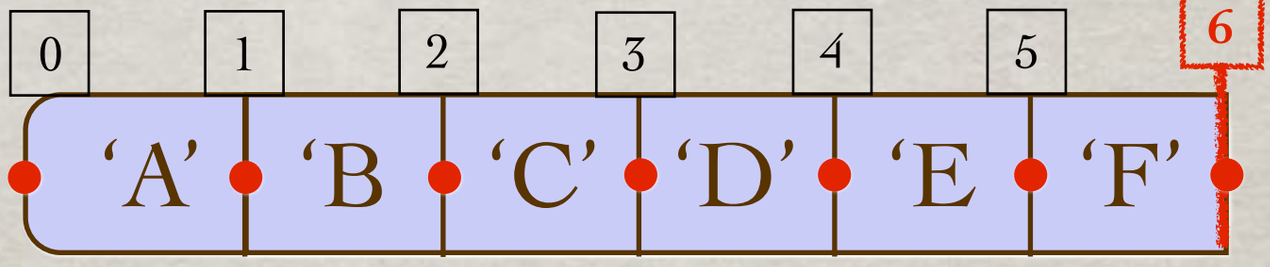
=> on obtient les cases i incluses ----> j exclue.

```
>>> len(liste)
8
```

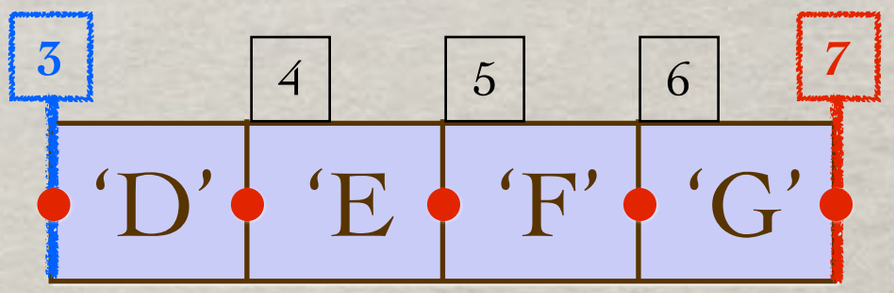
```
>>> liste=['A','B','C','D','E','F','G','H']
>>> id(liste), type(liste), len(liste)
(4302070704, <class 'list'>, 8)
```



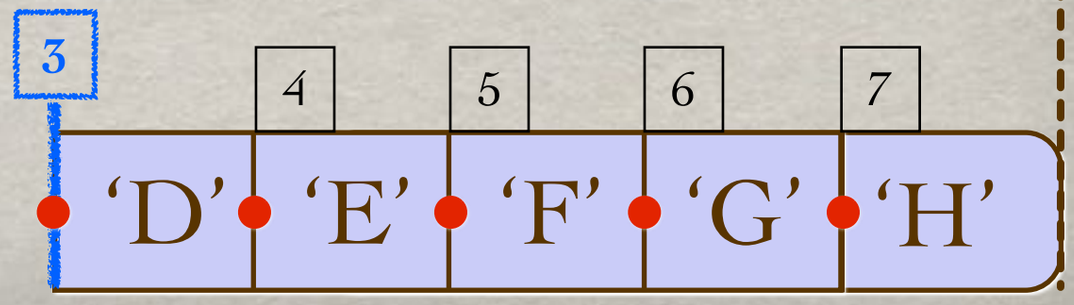

```
>>> liste[:6]
['A', 'B', 'C', 'D', 'E', 'F']
```
[Je tranche du début à 6]




```
>>> liste[3:7]
['D', 'E', 'F', 'G']
```
[Je tranche de 3 jusqu'à 7]

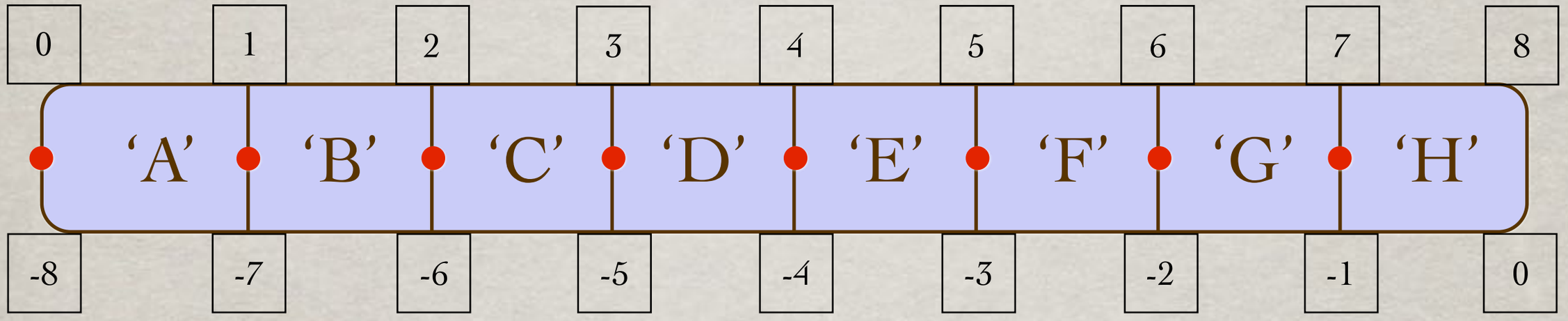



```
>>> liste[3:len(liste)]
['D', 'E', 'F', 'G', 'H']
```
[Je tranche de 3 jusqu'à la fin]



Les indices négatifs sont possibles : on raisonne alors en modulo N (N= len(liste))

$$-N \leq i \leq N-1$$



Astuce :

```
>>> liste[:5]
['A', 'B', 'C', 'D', 'E']
```

Je prend les 5 premiers éléments

```
>>> liste[-3:]
['F', 'G', 'H']
```

Je prend les 3 derniers éléments

```
>>> liste[-6:-2]
['C', 'D', 'E', 'F']

>>> -6%len(liste), -2%len(liste)
(2, 6)
```

```
>>> liste[-2:2]
[]

>>> -2%len(liste), 2%len(liste)
(6, 2)
```

La liste ne contient pas les valeurs mais les « pointeurs mémoires » c-à-d les flèches !

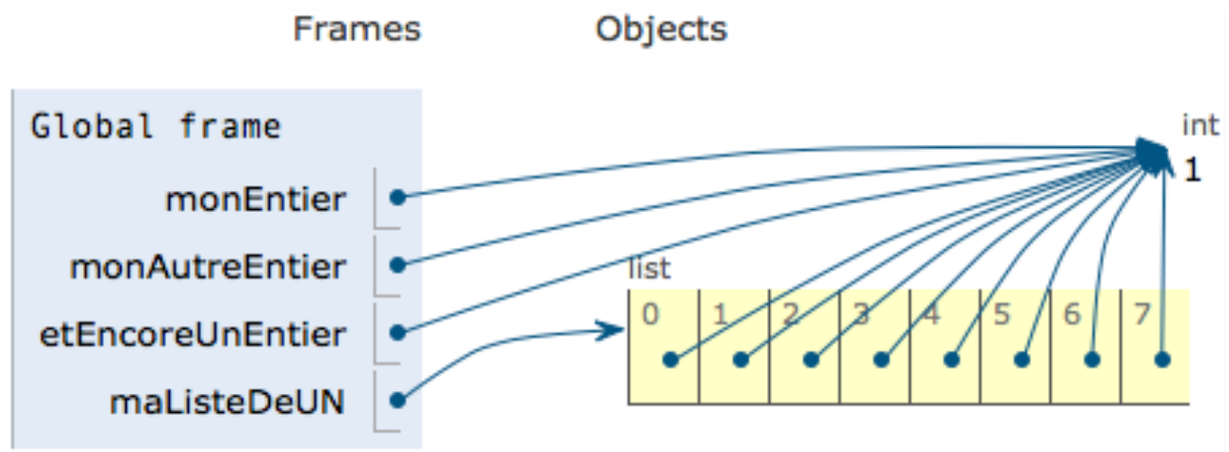
Write code in Python 2.7

(drag lower right corner to resize code editor)

```
1 monEntier=1
2 monAutreEntier=1
3 etEncoreUnEntier=1
4
5 maListeDeUN=[1,1,1,1,1,1,1,1]
```

line that has just executed

next line to execute



Sa taille en mémoire ne dépend pas de ce que contient la liste.

Et les objets dans tout ça ?

Le code

Write code in Python 3.3 (drag lower right corner to resize code editor)

```
1  maListe=[1,2,3]
2  monAutreListe=[1,2,3]
3
4  maListeBis=maListe
5
6  print(maListe==monAutreListe)
7
8  print(id(maListe))
9  print(id(monAutreListe))
10 print(id(maListeBis))
11
```

line that has just executed

next line to execute

les variables dans le code

Print output (drag lower right corner to resize)

```
True
139639478993264
139639417167240
139639478993264
```

Frames

Global frame

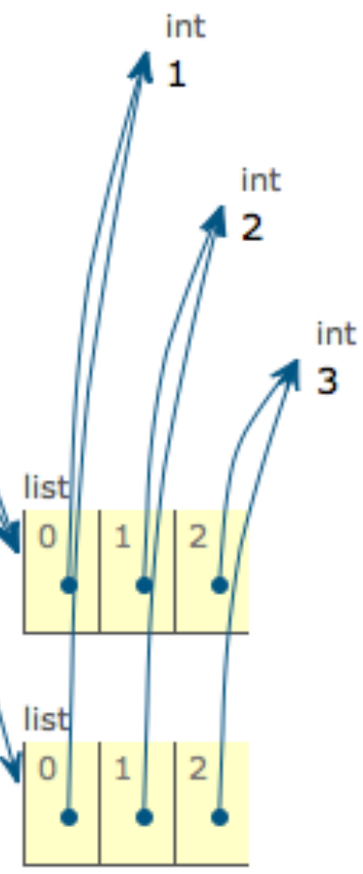
maListe

monAutreListe

maListeBis

L'espace objet => en mémoire

Objects



LES AUTRES SEQUENTIELS :

Les chaînes :

C'est une séquence de caractères alpha-numériques :

```
>>> chaine='MA CHAINE'
>>> chaine ; print(chaine)
'MA CHAINE'
MA CHAINE

>>> chaine[0] ; chaine[3:] ; chaine[3:-3]
'M'
'CHAINE'
'CHA'
```

RQ : C'est un objet itérable tout comme la liste, mais contrairement à la liste, on ne peut pas en modifier le contenu directement.

Opérations de base sur les chaînes :

L'opération de base spécifique aux chaînes est la concaténation :
Cela consiste a mettre bout-à-bout deux chaînes ou plus.

syntaxe python '+' : **str = str1 + str2**

```
>>> ma='MA'
>>> str='CHAINE'
>>> sp=' '
>>> verb='CONCATENEE'
>>> ma+sp+str+sp+verb
'MA CHAINE CONCATENEE'
```

Python généralise la concaténation à la «multiplication» c-à-d «concaténer n fois» :

```
>>> chaine = 'ri'*2 + ', ' + 'fi'*2 + ' et ' + 'lou'*2
>>> chaine
'riri, fifi et loulou'
```

<u>On peut évaluer la valeur numérique associée à l'écriture d'une quantité :</u>

```
>>> eval('4321') ; eval('3.14159')
4321
3.14159

>>> int('12') ; float('3.14')
12
3.14
```

```
>>> str(4321) ;  str(3.14159)
'4321'
'3.14159'
```

Pour des opérations plus complexes, on peut passer par le code ASCII :

```
>>> ord('A'), ord('a'), ord(' ')
(65, 97, 32)

>>> ord('&'), ord('@'), ord('$')
(38, 64, 36)
```

Les lettres de ‘A’ à ‘Z‘ sont codées entre 65 et 90
Les lettres de ‘a’ à ‘z‘ sont codées entre 97 et 122

Dans l’ordre alphabétique !

```
>>> chr(38), chr(64), chr(126)
('&', '@', '~')

>>> for i in range(33, 128):
...     if (i%16 !=0):
...             print(chr(i), end='')
...     else:
...             print(chr(i))
...

! " # $ % & ' ( ) * + , - . / 0
1 2 3 4 5 6 7 8 9 : ; < = > ? @
ABCDEFGHIJKLMNOP
QRSTUVWXYZ[\]^_`
a b c d e f g h i j k l m n o p
q r s t u v w x y z { | } ~
```

Les tuples ou t-uplet :

```
#python tuple :

>>> a=1
>>> b=3.14
>>> c='A'
>>> z=a, b, c
>>> z
(1, 3.14, 'A')

>>> z
(1, 3.14, 'A')

>>> z[0]; z[1]; z[2]
1
3.14
'A'

>>> z[0]=2
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
TypeError: 'tuple' object does not
 support item assignment
```

On ne peut modifier le contenu d'un tuple.
Il faut le recrée, ce qui change son identité d'objet

Usage classique :

```
# échange de contenu

>>> a=1; b=2;  a,b;  id((a,b))
(1, 2)
4301801664

>>> a,b = b,a

>>> a,b;  id((a,b))
(2, 1)
4301728584
```

Toutefois, on peut toujours transformer une chaîne ou un tuple en liste à l'aide de la commande list() :

```
>>> chaine='Toto' ; chaine
'Toto'

>>> Liste_str = list(chaine) ; Liste_str
['T', 'o', 't', 'o']
```

```
>>> tuple=(1,2,'Toto', 3.14) ; tuple
(1, 2, 'Toto', 3.14)

>>> liste_tuple=list(tuple) ; liste_tuple
[1, 2, 'Toto', 3.14]
```

```
>>> Liste_str[0] + Liste_str[1] + Liste_str[2] + Liste_str[3]
'Toto'
```

→ La concaténation permet de faire l'opération inverse pour les chaînes et les tuples

Méthodes agissant sur les listes :

Définition :

On appelle méthodes d'un objet, l'ensemble des opérations réalisables à partir d'un objet. Celles-ci sont des fonctions, définies dans un module externe, qui peuvent agir sur la structure de l'objet ou renvoyer une information sur l'objet.

L'ensemble de ces méthodes détermine le type ou la 'class' d'un objet.

[Concrètement : on peut connaître l'ensemble exhaustif des méthodes d'un objet en tapant «help(objet)»]

Extrait de : >>> help([1,'toto'])

```
class list(object)
 |  list() -> new empty list
 |  list(iterable) -> new list initialized from iterable's items
 |
 |  Methods defined here:
 |
 |  append(...)    L.append(object) -> None -- append object to end
 |
 |  count(...)     L.count(value) -> integer -- return number of occurrences of value
 |
 |  extend(...)    L.extend(iterable) -> None -- extend list by appending elements from the iterable
 |
 |  index(...)     L.index(value, [start, [stop]]) -> integer -- return first index of value.
 |                 Raises ValueError if the value is not present.
 |
 |  insert(...)    L.insert(index, object) -- insert object before index
 |
 |  pop(...)       L.pop([index]) -> item -- remove and return item at index (default last).
 |                 Raises IndexError if list is empty or index is out of range.
 |
 |  remove(...)    L.remove(value) -> None -- remove first occurrence of value.
 |                 Raises ValueError if the value is not present.
 |
 |  reverse(...)   L.reverse() -- reverse *IN PLACE*
 |
 |  sort(...)      L.sort(key=None, reverse=False) -> None -- stable sort *IN PLACE* Tim Peters in 2002
```

Timsort !

```
>>> L=list() ; L
[]
>>> L.append('3.14') ; L ; L.append('TOTO') ; L
['3.14']
['3.14', 'TOTO']

>>> L.extend([1,2,['C']])  ;  L
['3.14', 'TOTO', 1, 2, ['C']]

>>> L.insert(4,1) ; L
['3.14', 'TOTO', 1, 2, 1, ['C']]

>>> L.remove(1) ; L
['3.14', 'TOTO', 2, 1, ['C']]

>>> L.reverse() ; L
[['C'], 1, 2, 'TOTO', '3.14']

>>> L.index('TOTO')
3
>>> L.pop(L.index('TOTO')) ; L
'TOTO'
[['C'], 1, 2, '3.14']

>>> L.insert(0,2);  L.insert(0,2);  L
[2, 2, ['C'], 1, 2, '3.14']
>>> L.count(2)
3
```

list() : créé une liste, appellée ici L

L.append(x) : ajoute l'élément x en fin de liste

L.extend(liste) : idem mais avec une liste

L.insert(i, x) : x sera inséré en position i

L.remove(x) : supprime la 1ère occurence de x

L.reverse() : comme son nom l'indique !

L.index(x) : renvoie l'indice de x [1ère occurence]

L.pop(i) : éjecte et affiche x à l'écran

L.count(x) : indique le nb. d'occurence de x

Exercices pratiques :

Chercher les exercices sur les séquentiels

3 - Les blocs d'instructions

Les blocs sont permis par une opération de saut du compteur ordinal :

A l'origine GOTO en BASIC

Exemple de code qbasic :

```
5  CLS
10 PRINT "Guess a number between 1 and 10: ";
15 INPUT num
20 IF (num < 1 OR num > 10) THEN
25 PRINT "That is not between 1 and 10"
30 GOTO 10
35 END IF
40
45 IF (num = 6) THEN
50 PRINT "Correct!!!"
55 ELSE
65 PRINT "Try again"
70 PRINT
75 GOTO 10
80 END IF
```

blocs conditionnels

```
Guess a number between 1 and 10: ? 13
That is not between 1 and 10
Guess a number between 1 and 10: ? 2
Try again

Guess a number between 1 and 10: ? 7
Try again

Guess a number between 1 and 10: ? 6
Correct!!!
```

Ex :

```
Nom du bloc
        instruction_1
        instruction_2
Fin du bloc
```

```
if  (conditions) then
        instruction_1
        instruction_2
end if
```

BASIC

```
do while(conditions)
        instruction_1
        instruction_2
enddo
```

FORTRAN

Python

```
if (conditions):
    instruction_1
    instruction_2
```

Indentation obligatoire

C, Java, JS

```
for (conditions) {
        instruction_1
        instruction_2
}
```

Maple : Boucle do-od

```
> for i from 1 to 10 do
      j := i*i;
      print(j);
  od:
```

```
1
4
9
16
25
36
49
64
81
100
```

Pascal : Boucle if-else

```
var Age : Integer;

begin
  write('Entrez votre age : ');
  read(Age);
  if (Age >= 18) then
     begin        {1er bloc d'instructions}
       writeln('C''est bon, vous êtes majeur.');
       writeln('Vous pouvez rentrer');
     end
  else
     begin        {2eme bloc d'instructions}
       writeln('Rentre chez toi !');
       writeln('Et que je ne te vois plus');
     end;
end.
```

Maple : Boucle do-od

```
> for i from 1 to 10 do
      j := i*i;
      print(j);
  od:
```

```
1
4
9
16
25
36
49
64
81
100
```

Pascal : Boucle if-else

```
var Age : Integer;

begin
  write('Entrez votre age : ');
  read(Age);
  if (Age >= 18) then
     begin        {1er bloc d'instructions}
       writeln('C''est bon, vous êtes majeur.');
       writeln('Vous pouvez rentrer');
     end
  else
     begin        {2eme bloc d'instructions}
       writeln('Rentre chez toi !');
       writeln('Et que je ne te vois plus');
     end;
end.
```

En python, les blocs d'instructions sont naturellement délimités par l'**indentation** du code.

marge obligatoire de 4 espaces

```
r=eval(input("Entrez votre age :"))

if (r>=18):
    print("C'est bon, vous etes majeur")
    print("vous pouvez entrer")
else:
    print("Rentre chez toi")
    print("Et que je ne te vois plus")
```

```
Entrez votre age : 15

Rentre chez toi
Et que je ne te vois plus
```

```
Entrez votre age : 20

C'est bon, vous etes majeur
vous pouvez entrer
```

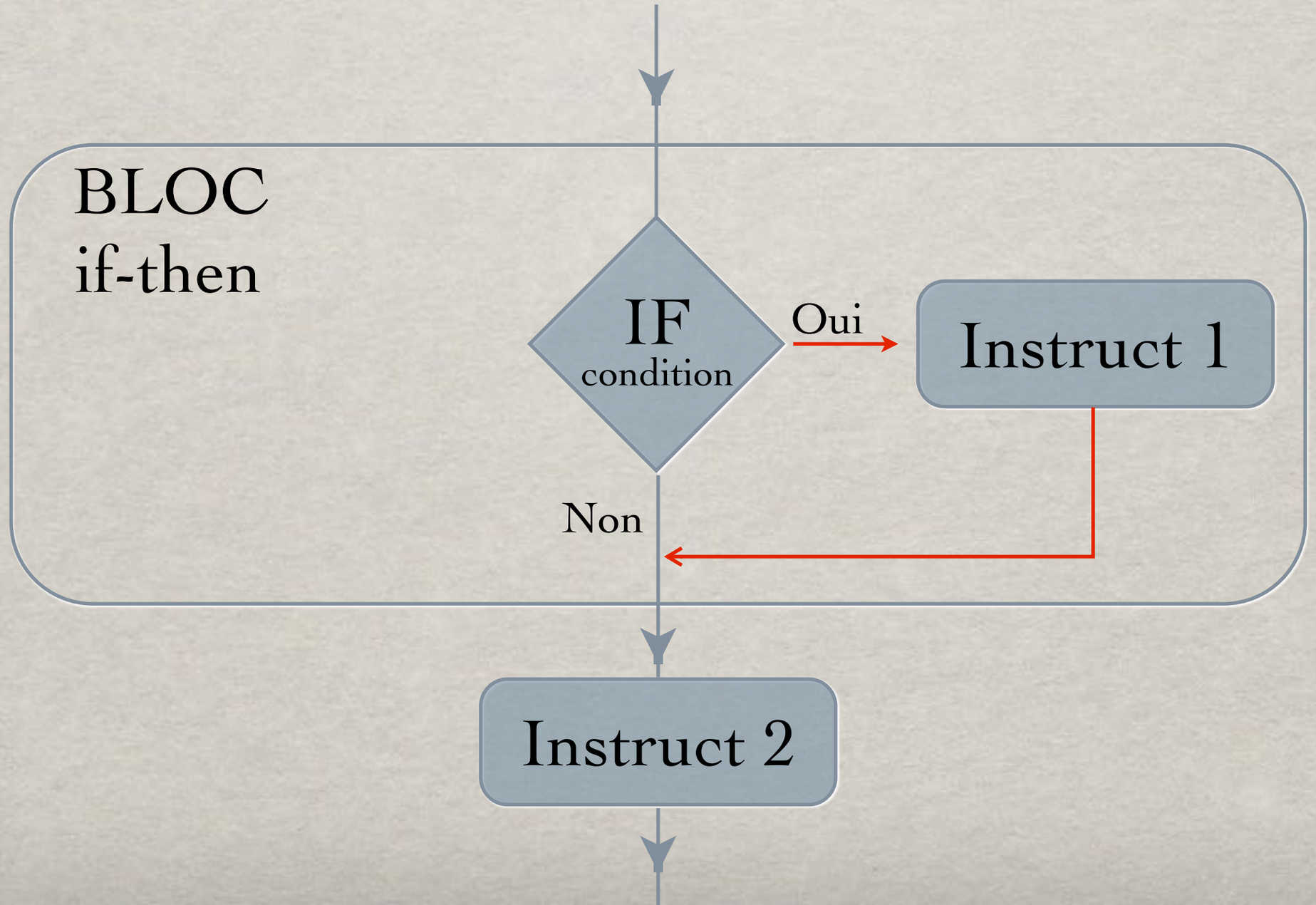
Contrairement à d'autres langages où l'indentation reste «optionnelle…», **Python ne comprend pas le code qui n'est pas indenté.**

Toutefois quelque soit le langage, le non respect de l'indendation et l'absence de commentaires rend la lecture du code difficile.

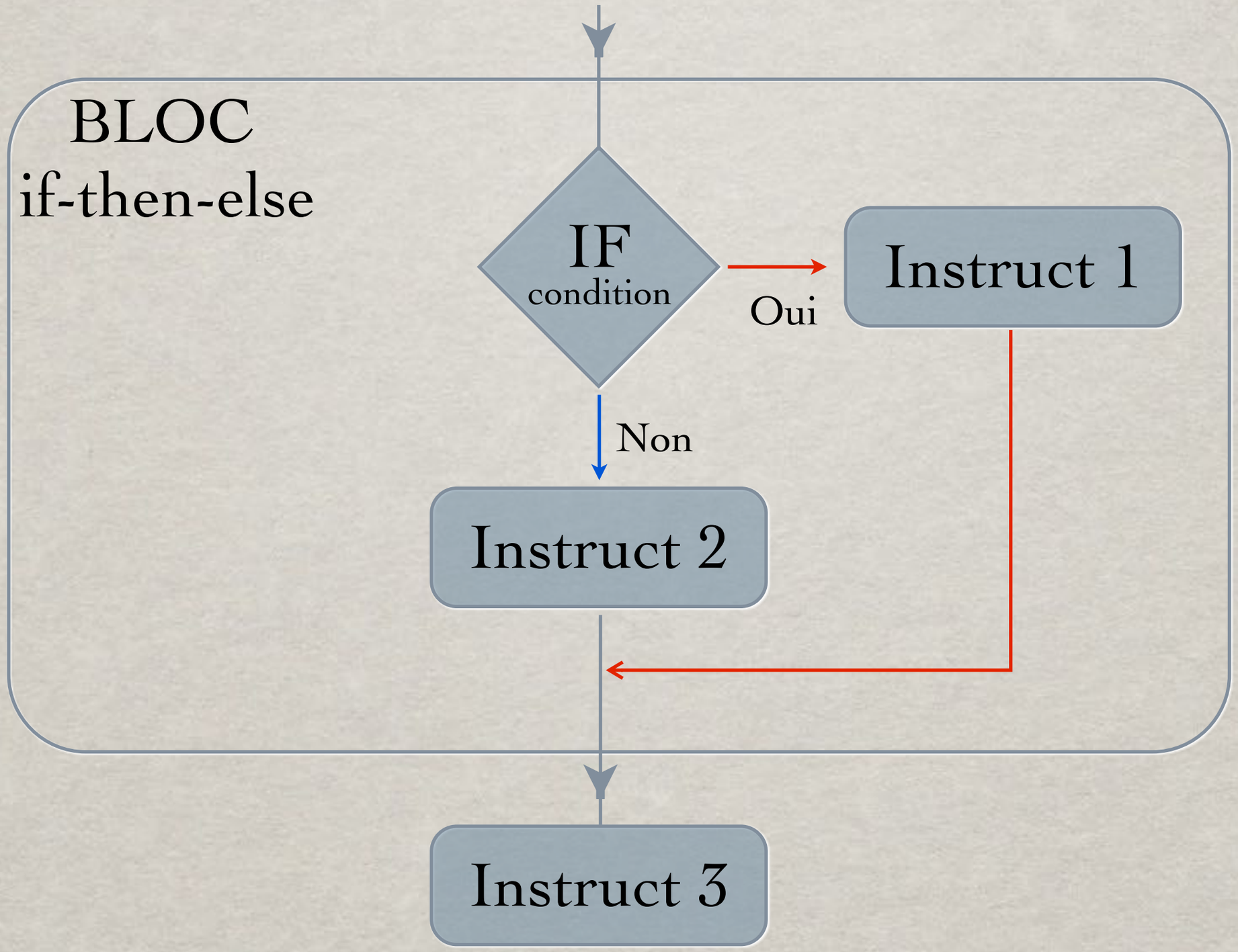
Les BLOCS conditionnels

Il s'agit d'un «aiguillage» qui peut envoyer ou non le déroulement du programme vers une instruction de coté en fonction de la valeur d'une condition :

True : condition vérifiée
False : condition non vérifiée



Selon les langages différentes syntaxes sont possibles :
La plupart utilisent la forme if-then-else équivalente à if-then suivie de ifnot-then



D'autres structures sont encore possibles :

if-then-elseif_1-elseif_2-....etc.....-elseif_N-else

Ou une variante en Visual-BASIC select-case :

Select:

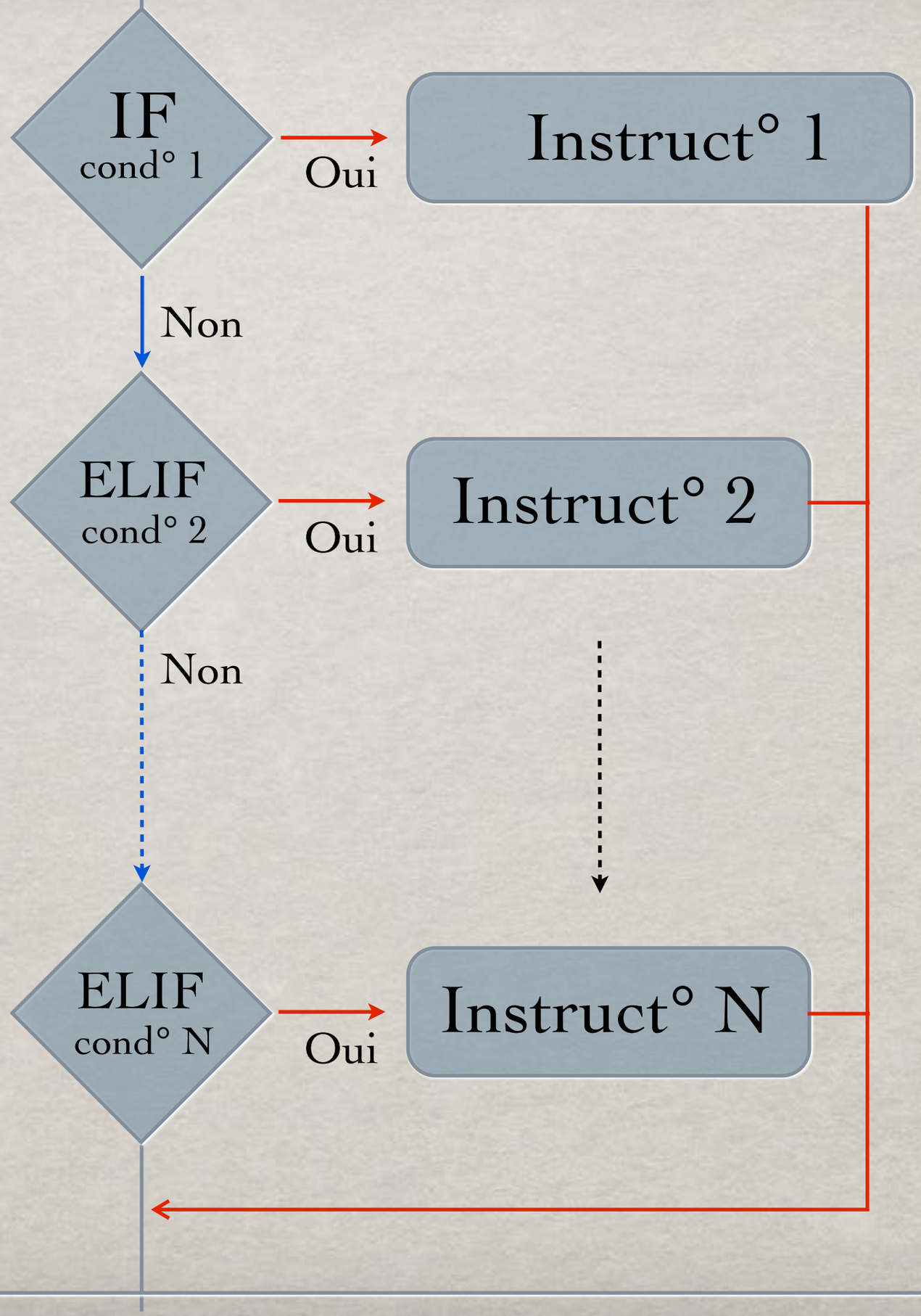
Case (condition_1)
instruction_1
instruction_2
.....

Case (condition_N)
instruction_1
instruction_2

.....

Toutes correspondent aux mêmes mécanismes au niveau du compteur ordinal :

BLOC
if - then
elseif
...
elseif



Exemples concrets de conditions :

```
>>> 10 > 5
True
```

```
>>> 1.25 < 2
True
```

```
>>> 2 == 3
False
```

```
>>> 2!=3
True
```

```
>>> (2==3, 2!=3, 5==5)
(False, True, True)

>>> (2 < 3, 2 > 3, 2 >= 3, 5 >= 5)
(True, False, False, True)

>>> ("Me" is "You", "Me" is "Me", 1 is 2, 2 is 2)
(False, True, False, True)

>>> L=[1,2,3, [4,5], "toto", "riri","fifi"]

>>> (2 in L, 4 in L, "toto" in L)
(True, False, True)

>>> 4 in L[3]
True
```

```
>>> x='Toto'; 't' in x
True
```

```
>>> x='Toto'; 'a' in x
False
```

```
>>> x='Toto'; 'oto' in x
True
```

```
>>> 'a'<'v'
True
```

Comparaisons et relations d'ordre

On appelle **<u>relation d'ordre</u>** sur un ensemble une relation binaire (nécessite deux éléments) qui vérifie :

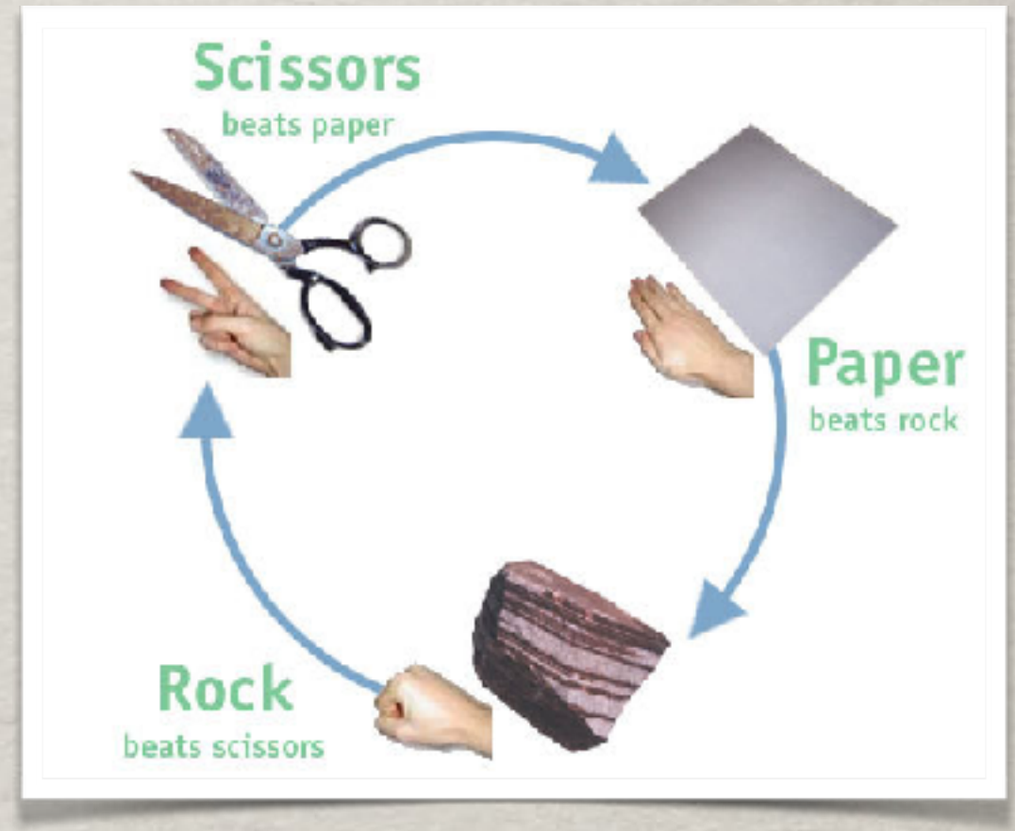
- Réflexivité : $A \mathfrak{R} A$
- Antisymétrie : $\left(A \mathfrak{R} B \quad et \quad B \mathfrak{R} A \right) \Rightarrow A = B$
- Transitivité : $\left(A \mathfrak{R} B \quad et \quad B \mathfrak{R} C \right) \Rightarrow A \mathfrak{R} C$

Exemple :

relation **inférieur ou égal** : $\leqslant$

Attention : l'inégalité stricte n'est pas reflexive $\not<$

Contre exemple : non transitif



On souhaite aussi une **relation d'ordre totale**, c-à-d que deux éléments soient toujours comparables :

$$A \mathfrak{R} B \quad ou \quad B \mathfrak{R} A$$

(contre exemple : $\subset$ inclusion ensembliste)

Exemples d'ordres :

- inégalité (⩽) entre deux entiers, deux réels
- ordre chronologique (dates & heure)
- ordre alphabétique entre deux caractères

- ordre lexicographique : $(a,b) \leq (c,d) \Rightarrow a < c \;\; ou \;\; (a = c \;\; et \;\; b \leq d)$

MQ : $(a,b) = (c,d) \;\; \Leftrightarrow \;\; (a = c \;\; et \;\; b = d)$

```
>>> [1,2,3]<[0,0,0,0]
False
>>> [1,2,3]<[2,0,0,0]
True
>>> [1,2,3]<[1,0,0,0]
False
>>> [1,2,3]<[1,3,0,0]
True
```

L'ordre lexicographique se généralise à tout n-uplet

- ex : ordre alphabétique entre deux chaînes :

```
>>> 'parle' > 'parla'
True
```

- ex : ordre entre deux complexes ; ordre entre deux points dans le plan

Les applications sont considérables pour les bases de données (algorithmes de tri)

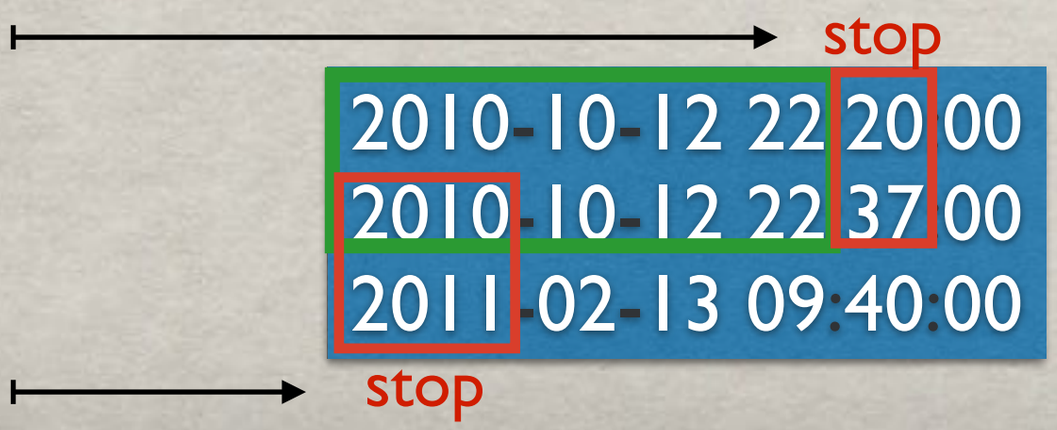
Application de l'ordre lexicographique

La représentation des dates dans une base de données est toujours faite selon la norme ISO 8601 : YMD hms

Year > Month > Day > hour > minute > second

date heure

Avec l'ordre lexicographique, on lit de gauche à droite :
Il est ainsi facile et efficace en comparant deux dates champ par champ de savoir qui est antérieure ou ultérieure. (cf Dichotomie)



2010-07-01 12:00:00
2010-07-14 20:20:00
2010-07-28 04:40:00
2010-08-10 13:00:00
2010-08-23 21:20:00
2010-09-06 05:40:00
2010-09-19 14:00:00
2010-10-02 22:20:00
2010-10-16 06:40:00
2010-10-29 15:00:00
2010-11-11 23:20:00
2010-11-25 07:40:00
2010-12-08 16:00:00
2010-12-22 00:20:00
2011-01-04 08:40:00
2011-01-17 17:00:00
2011-01-31 01:20:00
2011-02-13 09:40:00
2011-02-26 18:00:00
2011-03-12 02:20:00

Syntaxe if-then-else en python :

```
if (condition):          #condition est a valeur booléenne : True ou False
    instruction
        ...
        ...
elif (condition):
    instruction
        ...
        ...
else:
    instruction
        ...
```

Formulaire des conditions de base :

Syntaxe python			
		a == b	a est égal à b
		a != b	a est différent de b
	a < b	a > b	inégalité stricte
	a <= b	a >= b	inégalité ou égalité
L'objet a est aussi l'objet b :		a is b	id(a)==id : identité au sens objet
		i in L	i appartient à la liste L

Exemple python :

Python3 :

```python
# syntaxe if en Python

N = 123
if (N % 2 == 0):
    print("N est pair\n")
else:
    print("N est impair\n")

```

Résultat :

```
>>> (executing lines 1 to 7 of "<tmp 2>")
N est impair
```

Exercices pratiques :

Chercher les exercices sur les blocs conditionnels

Les BLOCS d'itérations : les boucles

Il s'agit ici d'utiliser l'opération de saut du compteur ordinal pour répéter un «motif algorithmique» en boucle.

Rq : Cette idée simple avait déjà été imaginée par Babbage pour des machines mécaniques et a d'abord été réalisée par des mécanographes qui lisaient des cartes perforées en Fortran.

On distingue deux types de boucles algorithmiques :

Celles-ci peuvent être conditionnelles : Boucle While

On répète un bloc d'instruction tant qu'une condition à valeur booléenne est remplie.
Pb : on ne sait pas a priori combien d'itérations seront réalisées

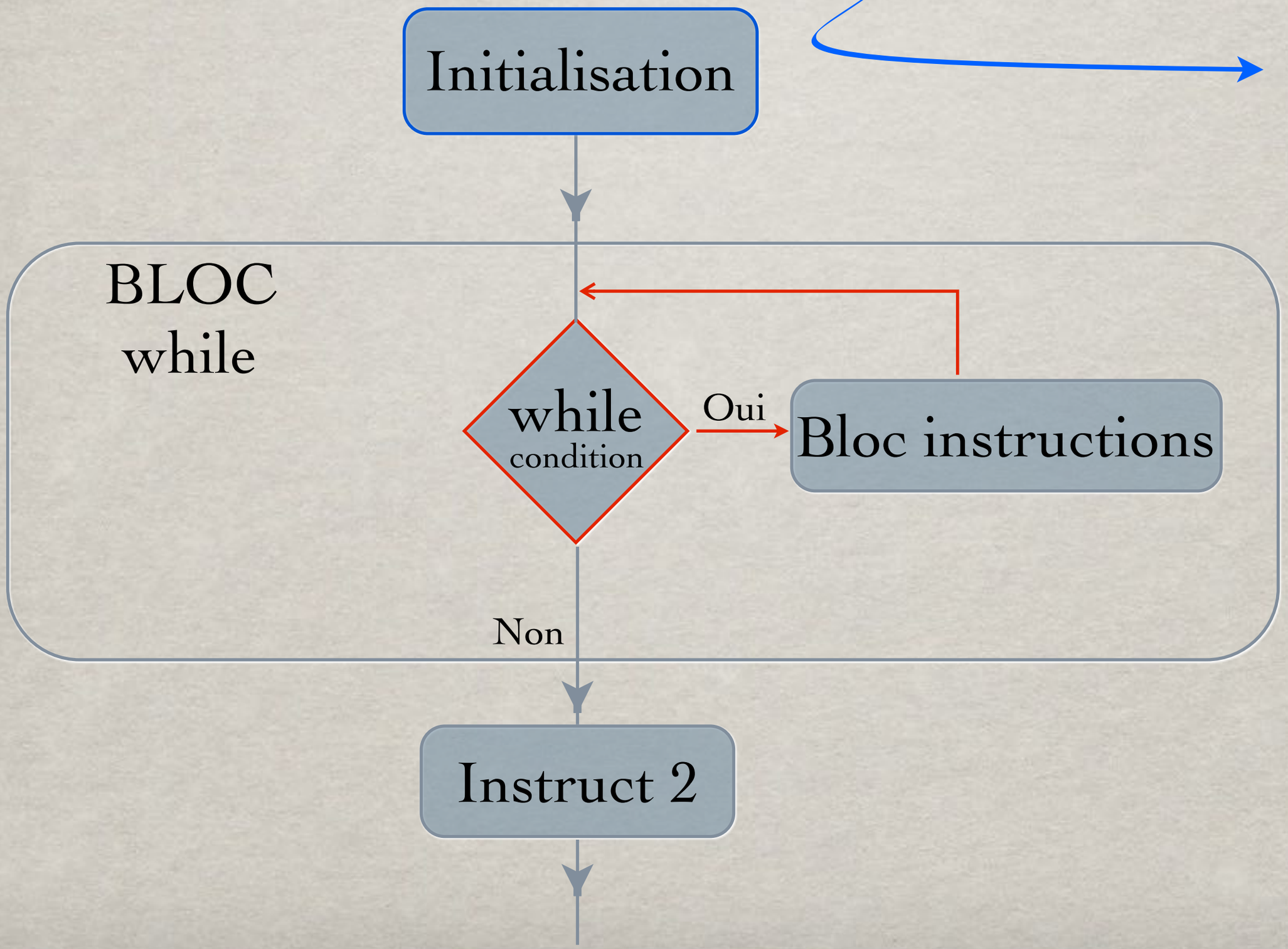
(ni même si il y aura une fin !)

Ou bien inconditionnelles : Boucle For

Un «itérateur» va compter la succession de boucles réalisées.
Pb : il faut connaître le nb. d'itérations dès le départ.

Boucle conditionnelle :

Celle-ci est généralement précédée par l'**initialisation** des variables ou paramètres impliqués dans la condition et la boucle elle-même.




```
i = 0
N=100
while(i<N):
    print(i)
    i+=1
```

Boucle inconditionnelle :

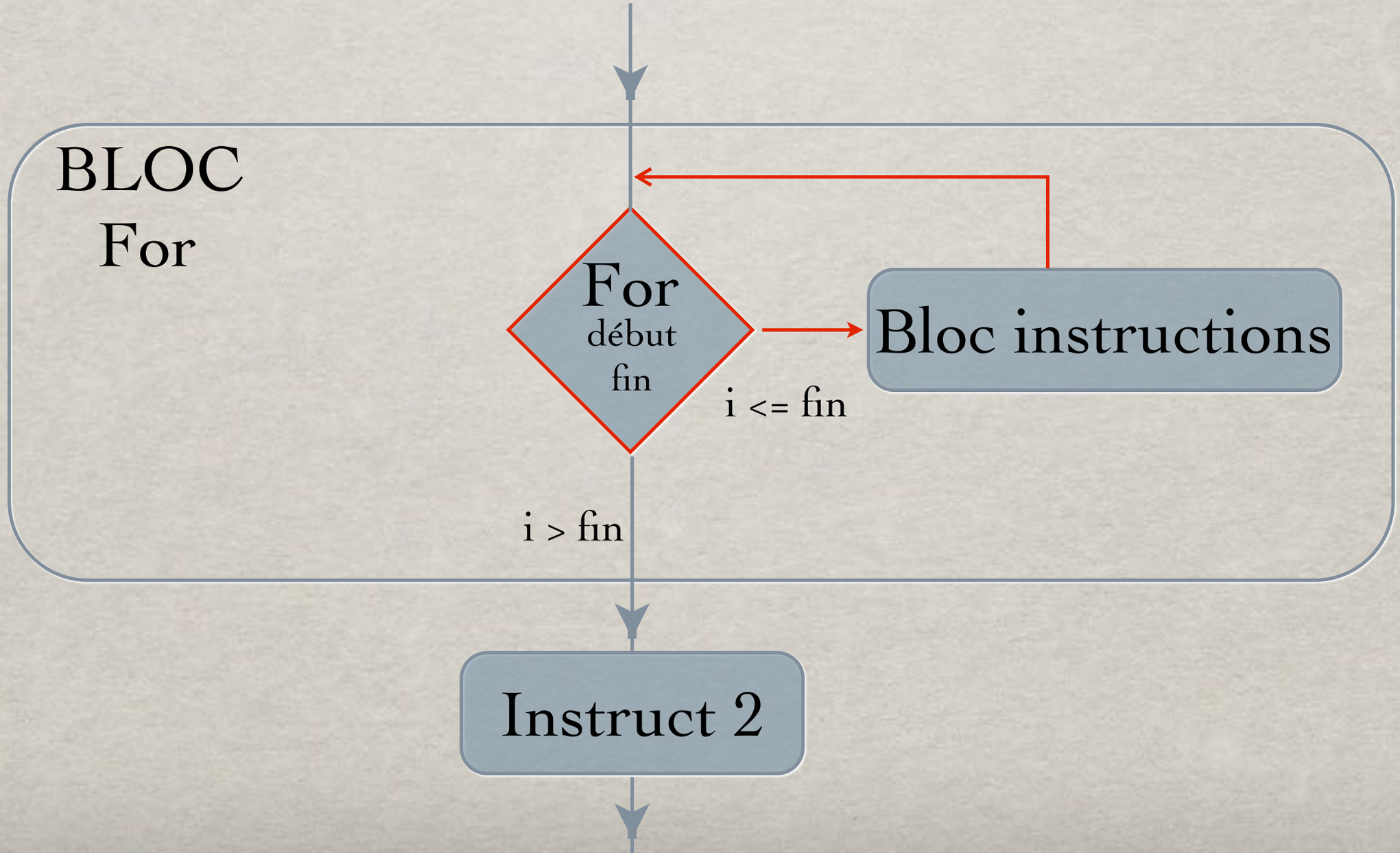
Celle-ci est en fait une boucle while mais qui agit sur un **compteur actif d'itération : i**

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

<=>

```
i = 0
while(i<10):
    print(i)
    i+=1
```

On parle aussi d'itérateur i



Python3 : boucle while

```
 9
10
11 i=0
12 while (i<10):
13     print("itération {}".format(i))
14     i+=1
15
```

Résultat :

```
>>> (executing lines 10 to 15 of "<tmp 1>")
itération 0
itération 1
itération 2
itération 3
itération 4
itération 5
itération 6
itération 7
itération 8
itération 9
```

Plus long à écrire mais les cas traités peuvent être plus subtiles.

On ne connaît pas le nombre d'itérations mais une condition quelconque sur i.

Python3 : boucle for

```
10
11 for i in range(10):
12     print("iteration {}".format(i))
13
```

Résultat :

```
>>> (executing lines 11 to 12 of "<tmp 1>")
iteration 0
iteration 1
iteration 2
iteration 3
iteration 4
iteration 5
iteration 6
iteration 7
iteration 8
iteration 9
```

Plus simple à écrire, mais il faut connaître à l'avance le nombre d'itérations

Cela n'est pas toujours possible !

Exemple :

```
>>> i=0
>>> Res=0
>>> while (Res < 1000):
...     Res=i**3-i**2
...     print(Res)
...     i+=1
...
0
0
4
18
48
100
180
294
448
648
900
1210
```

```
>>> Res=0
>>> for i in range(10):
...     Res=i**3-i**2
...     print(Res)
...


0
0
4
18
48
100
180
294
448
648
```

On ne sait pas à l'avance pour quelle valeur de i, le résultat de $i^3 - i^2$ va dépasser 1000

Python3 : Exercice : Palindrome

Ecrire un programme qui prend en entrée une chaine de caractères,
et qui détermine si cette chaine est un palindrome : (exemple KAYAK)

A chercher

Résultat :

```
>>> (executing lines 1 to 10 of "<tmp 2>")
C'est un palindrome
```

Rq :

L'instruction **break**, permet de mettre fin à tout instant à la boucle **for** en cours d'exécution.

Dans certains cas c'est donc une astuce pour réaliser une boucle dont on ne connait pas la condition de terminaison. [Il faut toutefois en avoir une majoration finie]

Invariant de boucle

Définition :

On appelle invariant de boucle, toute affirmation qui reste vraie d'une itération à la suivante.

Exemple de l'algorithme d'Euclide :

$$PGCD(A_n, B_n) = PGCD(A, B)$$

est un invariant de boucle (cf Démo).

Certains invariants de boucle, ne changent ni dans leur valeur, ni même dans leur expression.
On a donc tout intérêt à les sortir de la boucle.
[En général, le compilateur optimise l'exécution du programme en sortant ces invariants inutiles]

```
>>> S=0
>>>
>>> for i in range(10000):
...         x=5    #Cette instruction répétée 10000 fois n'a rien a faire dans la boucle
...         S+=i
...
```

```
>>> x,S
(5, 49995000)
```

Les boucles en Python

Comment choisir la bonne structure itérative ?

Je connais le nombre d'itération ?

OUI → **FOR IN**

NON → **WHILE**

Structure :
while (condition):

J'ai besoin des indices ?

OUI →

Structure : **i est un entier**
for i in range(N):

NON →

Structure : **elem est un objet du séquentiel**
for elem in sequentiel:

Exemples de syntaxes « for in » : for itérateur in itérable

En python <u>toutes les boucles for</u> se font sur séquentiels, pas sur des entiers (C, Fortran, Java).
Essayez les exemples suivants pour bien comprendre :

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

« faux ami » → engendre la liste itérable [0,1,2,3,4,5,6,7,8,9]
(c'est un cas particulier)

```
for  (x,y)  in  [(1,2), (2,2), (3,2)]:
```

```
for nom in {'Einstein': 'Albert', 'Newton': 'Isaac', 'Gauss': 'Carl-Friedriech', 'Tesla': 'Nicolas'}:
```

```
for key in dictionnaire.keys():
```

→ méthode qui engendre la **[liste des clefs]**

Rq : Privilégier des noms explicites

```
for char in ''maChaineDeCaractères'':
```

```
for tupNoeud in noeudsAVisiter:
```

→ Chaque noeud étant un objet tuple

Quelques remarques :

Rq 1 : **Ne jamais changer la valeur de l'itérateur dans une boucle for !**

[ça ne changerais que pour l'itération en cours mais c'est une très mauvaise pratique]

L'itérateur et surtout l'itérable doivent être anticipés dès la conception du code :

Rq 2 : Toujours se demander au sujet de l'itérateur :
« Quel est l'objet que j'ai dans la main ?» —> quelle information ?

Rq 3 : Toujours se demander au sujet de l'itérable :
« Quel objet est-ce que je voudrais avoir ? »
—> c-à-d anticiper : de quelle information ai-je besoin ?

Nous irons plus loin au second semestre :

Rq 4 : La structure **for in** est beaucoup utilisée avec les **listes par compréhension** :

```
[k**2 for k in range(1,6)]   ————————————>   engendre  [1, 4, 9, 16, 25]
```

```
for monCarré in [k**2 for k in range(1,6)]
```

Exercices pratiques :

Chercher les exercices sur les blocs d'itérations

4 - Les fonctions en Python
Les blocs fonctionnels

Il existe une troisième catégorie de bloc en Python : les blocs fonctionnels.

Ces blocs sont eux-mêmes des objets Python avec un nom de variable mais qui permettent de mettre de coté toute une partie de code que l'on va pouvoir réutiliser à volonté en « **appelant** » la fonction avec les parenthèses ( ) —> **to call** !

Ces objets fonctions sont donc appelés des callables

```
def maFonction(x):
    y = 2*(x+1)**2
    return y
```

```
>>> maFonction
<function maFonction at 0x10e182d90>

>>> maFonction(1)
8
```

De fait pour que la fonction soit utilisée il faut l'appeler : c'est l'usage des **( )** !

Si on ne l'appelle pas la fonction est créée mais son code n'est pas exécuté.

Définition d'un bloc fonctionnel :

nom de la fonction

ensemble des variables

Paramètres optionnels : leur valeur peut être fixée a priori

Bloc de la fonction

```
Function nom (var1, ...,  varN, par1=p1, ....parM=pM)
        instruction
            .....
        instruction
        result = valeur de la fonction
EndFunction
```

«Scope» de la fonction

Valeur qui sera retournée par la fonction

Syntaxe Python :

```
1 def fonction(x, y, p=2):
2     return x*y/p
3
4 print(fonction(1, 1))
5 print(fonction(1, 1, p=3))
6
```

```
>>> (executing lines 1 to 5 of "<tmp 4>")
0.5
0.3333333333333333
```

Fonction qui calcule le carré !!!

```
1 def sqr(x):       #fct. carré
2     return x**2
3
4 print(sqr(5.0))
5
```

```
>>> (executing lines 1 to 4 of "<tmp 4>")
25.0
```

Le bloc fonctionnel peut contenir d'autres blocs d'instruction :

Fonction qui teste la parité

```
1 #définition d'une fonction de parité d'un entier
2 #renvoie une chaîne : 'pair' ou 'impair'
3
4 def paire(N):
5     if (N % 2 == 0):
6         return 'pair'
7     else:
8         return 'impair'
9
10 print('2 est un nombre '+ paire(2))
11 print('3 est un nombre '+ paire(3))
12
```

```
>>> (executing lines 1 to 11 of "<tmp 4>")
2 est un nombre pair
3 est un nombre impair
```

- Boucle for
- bloc if-else
- ou les deux

Fonction qui teste le caractère premier d'un nombre :

```
1  def isprime(N):
2      Res=True
3      for i in range(2,N):
4          if (N % i==0):
5              Res=False
6              break
7      return Res
8
9  print("4 : ",isprime(4))
10 print("5 : ",isprime(5))
11 print("7 : ",isprime(7))
12 print("9 : ",isprime(9))
13 print("17 : ",isprime(17))
```

```
>>> (executing lines 1 to 13 of "<tmp 4>")
4 :  False
5 :  True
7 :  True
9 :  False
17 :  True
```

Exercice :

Ecrire sur cette base un algorithme simple qui trouve tous les nombres premiers inférieurs à 100 ou 1000.

Fonction Python

fonction qui renvoie toujours la même chose

```
1 def RIEN():
2     return 'Je ne fais rien'
3
4 print("*** Resultat ***")
5 print(RIEN, end=';\n')
6 print(RIEN(), end=';\n')
7 print(type(RIEN), end=';\n')
8 print(type(RIEN()), end=';\n')
9
```

```
>>> (executing lines 1 to 8 of "<tmp 2>")
*** Resultat ***
<function RIEN at 0x3d7ec48>;
Je ne fais rien;
<class 'function'>;
<class 'str'>;
```

«Procédure» Python

fonction qui ne renvoie rien, mais déclenche l'affichage

```
1 def nothing():
2     print('Toto')
3
4 print("*** Resultat ***")
5 print(nothing, end=';\n')
  print(nothing(), end=';\n')
6 print(type(nothing), end=';\n')
7 print(type(nothing()), end=';\n')
8
```

```
>>> (executing lines 1 to 8 of "<tmp 2>")
*** Resultat ***
<function nothing at 0x3d7ec90>;
Toto
None;
<class 'function'>;
Toto
<class 'NoneType'>;
```

Elles se distinguent des procédures (PASCAL, Visual-Basic, autre...) par le fait :

- qu'elles demandent a priori une entrée : les variables (paramètres)
- elles renvoient nécessairement une valeur (qui peut être 'NONE' en python)

Variables locales ou globales :

Les variables définies à l'intérieur d'une fonction sont locales, c-à-d qu'elle ne sont pas définies en dehors de la fonction ⟶ [leur nom n'existe pas ou désigne une autre variable].

On parle alors de « **scope** » de la fonction : c'est à dire la «**portée**» des noms de variable.
—> **La variable n'est définie que localement dans le bloc fonctionnel.**
—> **Dès que la fonction renvoie son résultat toutes ses variables sont détruites.**

On peut forcer une variable a être globale : global var

```
>>> x=5


>>> def func():
...     x=3
...     return x
...


>>> x
5

>>> (func(), x)
(3, 5)
```

```
>>> global x
>>> x=5


>>> def func():
...     global x
...     x=3
...     return x
...


>>> x
5

>>> (func(), x)
(3, 3)
```

Dans ce cas x ne peut pas être une variable de la fonction.(ce n'est plus une variable)

Exercices pratiques :

Chercher les exercices sur les fonctions

Histoire des fonctions

Les langages de niveaux structurés : Pascal, Fortran 90, C, et au delà : python etc...
offrent la possibilité de créer au sein du programme ou dans un module externe [fichier ext.],
une fonction qui retournera une valeur si on l'appelle avec des variables et des paramètres.

Exemple historique :
FORTRAN/PASCAL ont introduit la possibilité d'avoir des modules externes

Cette possibilité nous fait sortir du paradigme impératif, proche des mécanismes à l'oeuvre dans l'unité de calcul vers la **programmation fonctionnelle**, beaucoup plus structurée.
=> Ocaml est un exemple type de langage destiné à la programmation fonctionnelle et très utilisé par les chercheurs en informatique [cf λ-calcul]

Exemple historique :
FORTRAN 77 ne gère pas les appels récursifs alors que FORTRAN 90 oui.
[Récursivité : fonction définie à partir d'elle-même dans sa définition]

Concrètement : c'est au compilateur de transcrire les fonctions appelées par le programme dans une forme impérative.

[C'est à dire une succession d'instructions envoyées au calculateur sous forme ordinale.]

L'intérêt des fonctions réside évidement dans l'économie de CODE réalisée (1 et 2), mais plus encore dans la lisibilité du programme (3 et 4) :

1 - Il n'est pas nécessaire de ré-écrire le code de la fonction à chaque fois que l'on l'utilise quelque part. On peut ré-utiliser cette fonction ultérieurement dans le même programme ou dans un autre programme.

2 - On peut importer dans son programme une fonction déjà écrite, par soi-même ou par un tiers. Il existe ainsi des librairies de fonctions dans tous les langages courants. [En particulier en FORTRAN pour la programmation scientifique]

3 - **Le programme est structurée :**

- Le corps principal du code (main.py) est plus court, plus simple à comprendre.
- Les organes périphériques (modules.py) peuvent être étudiés/validés séparément.
- Certains modules peuvent être appelés ou non selon des besoins qui ne seront définis que lors de l'utilisation finale du programme.

4 - La structuration s'inscrit à nouveaux dans une démarche de **communication** :

Programmer c'est avant tout écrire du code qui doit pouvoir

- être compris par un tiers très rapidement
- être modifié en vue d'objectifs qui ont pu changer
- et même être traduit dans un autre langage.

Ex : Problématique scientifique & spécifications du programme

J'ai fait un programme FORTRAN 90 qui intègre les équations du mouvement pour déterminer la trajectoire d'un satellite.

Le programme est dans un fichier principal qui fait une boucle sur le temps. L'intégrateur de «pas à pas» se trouve dans un fichier externe et réalise l'intégration d'un seul pas de temps par la méthode d'Euler.

→ PB : le programme n'est pas assez précis / fiable vis à vis des cond° ini.

Il est possible de gagner en précision à l'aide d'un meilleur intégrateur :

→ Runge-Kutta d'ordre 2, 4 ou 8 :
(disponible gratuitement en ligne dans une librairie Numérical Recipes)

Proposer des solutions :

1 => il suffit de remplacer le module d'intégration Euler par le Runge-Kutta dans le fichier consacré à l'intégration du pas de temps «sans rien changer» (ou presque) au code principal.

La méthode Runge-Kutta est plus précise mais plus lente :

2 => je peux aussi modifier le main.f et proposer à l'utilisateur de choisir :

- un intégrateur Euler rapide mais peu précis.
- un intégrateur RK plus lent mais bien plus précis.

3 => je peux aussi modifier le main.f et proposer à l'utilisateur de choisir l'ordre du RK

Structure générale d'un projet informatique

Fichier «main.py» :

- Contient l'architecture du programme du début à la fin ; le programme appelle des fonctions extérieures.
- C'est le fichier principal et c'est lui qui devra être interprété voire compilé.

Fichier «modules.py» :

- Contient les définitions des fonctions créées pour le programme et qui seront appelées par le fichier main.

Fichier «bibliothèques.py» :

- Contient les définitions standards fournies avec le compilateur ou l'interpréteur.
- Ses différents modules doivent donc être importés depuis main.py.