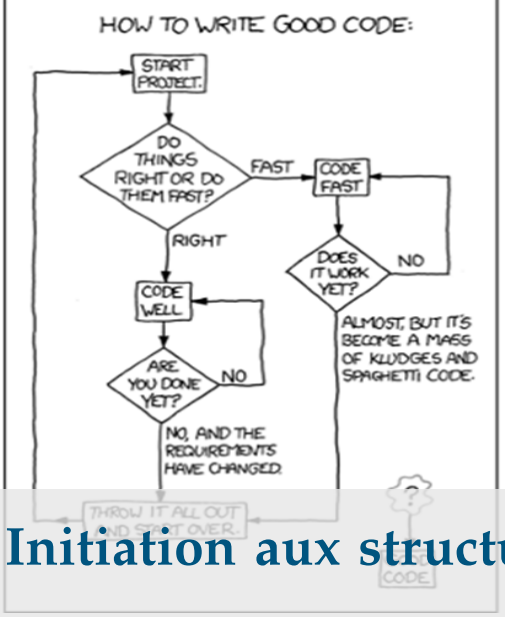

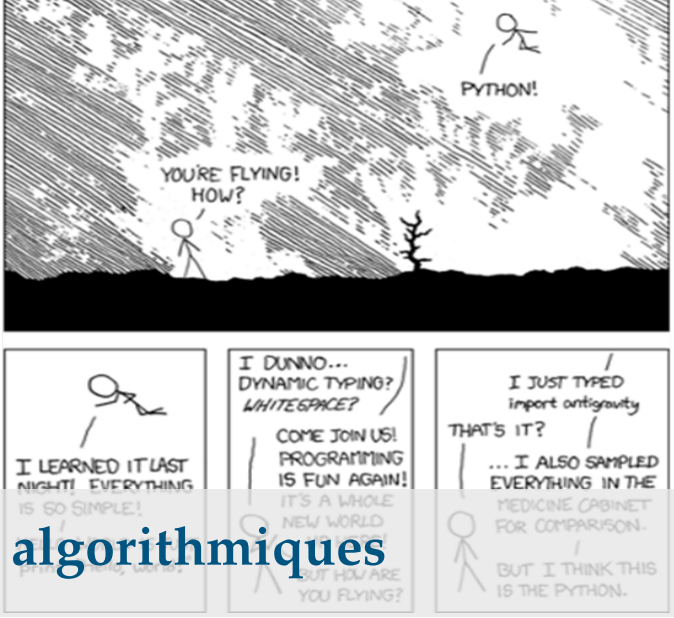

3 Initiation aux structures algorithmiques

3.1 L'instruction conditionnelle

3.1.1 Algorithme

Quand on veut écrire un programme, on souhaite établir des connections logiques entre les instructions. Ainsi, l'instruction conditionnelle a pour objet d'intervenir dans le choix de l'instruction suivante en fonction d'une expression booléenne qu'on désignera par **condition** :

- Choisir un type de variable.
- Concevoir un algorithme utilisant une structure conditionnelle (Si), une structure itérative (while), une structure itérative (for).

> **Si** condition
> **alors** bloc d'instructions 1
> **sinon** bloc d'instructions 2
> **Fin-du-Si**
>
> signifie que
>
> - **Si** la condition est vérifiée (expression booléenne=`True`) **alors** le programme exécute les instructions du bloc 1;
> - **si** la condition n'est pas vérifiée (expression booléenne=`False`) **alors** le programme exécute les instructions du bloc 2.

3.1.2 Syntaxe en Python

```
1 if condition :
2     bloc d instructions 1
3 else :
4     bloc d instructions 2
```

- **Si** et **Sinon** se traduisent par `if` et `else`.
- **Alors** se traduit par « **:** » en bout de ligne et une indentation de toutes les lignes du bloc 1.
- **Fin-du-Si** se traduit par un retour à la ligne sans indentation.

3.1.3 Exemple

On veut tester si un nombre x est proche de 3 à 10^{-4} près. On peut alors écrire la fonction suivante.

```
def est_proche(x):
    """x est proche de 3 à 10**-4 près ?"""
    distance = abs(x-3)
    if distance <= 10**(-4) :
        return True
    else :
        return False
```

Remarque

La partie **sinon** est optionnelle. Sans elle, si la condition n'est pas vérifiée, alors la machine n'exécute rien.

Remarque

On pouvait très bien remplacer cette boucle conditionnelle avec un usage astucieux des booléens, comme suit.

```
def est_proche(x):
    """x est proche de 3 à 10**-4 près ?"""
    distance = abs(x-3)
    return distance <= 10**(-4)
```

La fonction est alors plus concise, mais plus difficilement lisible.

3.1.4 À propos des conditions

L'expression booléenne derrière le **si** joue le rôle de test. Pour exprimer cette condition, on a besoin des **opérateurs de comparaison** (inférieur strict, supérieur strict, inférieur ou égal, supérieur ou égal, égal à, différent de) et des **connecteurs logiques** (non, et, ou).

- Calcul du carré d'un nombre positif.

```
>>> x=4
>>> if x >= 0 :
        car = x**2

>>> car
    16
```

- Condition avec un « et ».

```
x = 0.5
if x >= -1 and x <= 1 :
    print("Il existe un angle theta tel que x = cos(theta).")
```

3.1.5 Imbrication de plusieurs conditions

On peut se trouver face à un problème qui se scinde en plus de deux cas (par exemple, dans le cas des équations du second degré, on teste si le discriminant est strictement positif, nul ou strictement négatif). Dans ce cas, voici comment procéder.

```
 if condition 1 :
    bloc d instructions 1
 elif condition 2 :
    bloc d instructions 2
 elif condition 3 :
    bloc d instructions 3
.
.
.
 else :
    bloc final
```

- **Sinon si** se traduit par `elif`.

Voici une fonction qui calcule le maximum de trois entiers `a, b, c` :

```
def max3 (a, b, c) :
    """ renvoie le maximum de a, b ,c.
    précondition : a, ,b et c sont 3 entiers """
    if a <= c and b <= c :
        return c
    elif a <= b and c < b :
        return b
    else :
        return a
```

3.2 Boucles définies

	Pour variable **dans** liste **répeter** bloc d'instructions b **Fin-de-la-boucle**
	signifie que
pour chaque élément de la liste `liste`, le programme exécute les instructions du bloc b.	

3.2.1 Syntaxe en Python

```
for variable in liste :
    instructions
```

Ici encore, la ligne contenant le mot-clé **for** doit se finir par un « **:** » et les instructions du bloc doivent être indentées. La fin de la boucle est marquée par un retour à la ligne non indenté.

3.2.2 Les intervalles d'entiers en Python

.

Pour répondre à la première question, il suffit de remarquer que les entiers de 0 à 19 par exemple, sont en fait les éléments d'une liste : `[0, 1, 2, ··· , 19]`. En Python, cette liste s'écrit de la manière suivante : `range(20)`.

Voir *Why numbering should start at zero*, E. W. Dijkstra, EWD831. Disponible en ligne.

Précisément, si a et b sont deux entiers, `range(a,b)` contient les éléments de l'intervalle semi-ouvert $[\![a, b[\![$, dans l'ordre croissant. Avec un seul argument, `range(b)` signifie `range(0,b)`.

Redisons-le, car c'est un fait important en Python : `range(a,b)` est intervalle **fermé** à gauche, **ouvert** à droite

Avec `range`, nous pouvons maintenant itérer sur une liste d'entiers :

```
x = 5
for k in range(20):
    x = 2 * x - k - 3

print(x)
#  Résultat obtenu : 1048600.
```

3.3 Boucles indéfinies ou conditionnelles

3.3.1 Algorithme

On peut aussi être amené à répéter un bloc d'instructions sans savoir combien de fois on devra le répéter.

Disposant d'une suite croissante, non majorée, on cherche à trouver le plus petit entier p tel que la valeur au rang p dépasse 10000.

Dans ce cas, on utilise la boucle **Tant que** qui permet de répéter le bloc d'instructions tant qu'une certaine condition est vérifiée.

Tant que condition
    faire bloc d'instructions
Fin-du-Tant-que

signifie que

Tant que la condition est vérifiée (expression booléenne=*True*)
Faire le bloc d'instructions.

3.3.2 Syntaxe en Python

```
while condition :
    instructions
```

Rechercher le premier entier n tel que la somme des entiers de 1 à n dépasse 11.

```
n = 1
s = 1
while s < 11 :
    n = n + 1
    s = s + n

n
# REPONSE  : n=5  (dans ce cas s=15)
```

Application 1
Sujet

Exercice – Structures conditionnelles

Question 1 Implémenter une fonction `est_plus_grand(a:int,b:int)-> bool` renvoyant `True` si $a > b$, `False` sinon.

Question 2 Écrire une fonction `neg(b)` qui renvoie la négation du booléen `b` sans utiliser `not`.

Question 3 Écrire une fonction `ou(a,b)` qui renvoie le ou logique des booléen `a` et `b` sans utiliser `not`, `or` ni `and`.

Question 4 Écrire une fonction `et(a,b)` qui renvoie le et logique des booléen `a` et `b` sans utiliser `not`, `or` ni `and`.

Exercice – Un tout petit peu d'arithmétique

Question 5 Implémenter la fonction `unite(n:int)->int` renvoyant le chiffre des unités de l'entier `n`.

```
>>> unite(123)
    3
```

Question 6 Implémenter la fonction `dizaine(n:int)->int` renvoyant le chiffre des dizaines de l'entier `n`.

```
>>> dizaine(123)
    2
```

Question 7 Implémenter la fonction `unites_base8(n:int)->int` renvoyant le chiffre des unités de l'entier `n` en base 8.

Exercice – Structures itératives

Question 8 Écrire la fonction `somme_inverse(n:int)->float` calculant la somme des inverses des `n` premiers entiers non nuls.

Exercice – Suites d'entiers

Question 9 Implémenter la fonction `impairs(n:int)` permettant d'afficher la liste des `n` premiers entiers naturels impairs.

Question 10 Implémenter la fonction `multiples_5(d:int, f:int)` permettant d'afficher la liste de tous les multiples de 5 compris entre `d` et `f` (bornes incluses si se sont des multiples de 5).

Question 11 Implémenter la fonction `cube(f:int)` permettant d'écire la liste de tous les cubes d'entiers naturels inférieurs ou égaux à `f` (inclus si `f` est un cube).