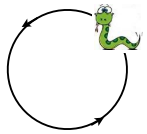
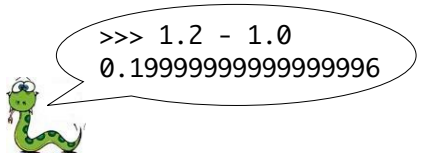
Itérations (while)

Les nombres approchés





Problème -> Algorithme -> Programme

- Partons d'un **problème**.
- Soit à faire **calculer** la somme des entiers de [1,n], avec n entier ≥ 1.

$$S = 1 + 2 + 3 + \cdots + n$$

- Cette somme S dépendant de n, intéressons-nous à la **fonction** S(n).

$$S(n) = 1 + 2 + 3 + \cdots + n$$

- Il s'agit de construire un **algorithme** de calcul de S(n) : une méthode ***mécanique*** permettant d'arriver au résultat.
- Une fois un tel algorithme obtenu, il s'agira de **coder** la fonction S dans un langage de programmation, ici Python.
- Pour produire au final un **programme exécutable** sur machine.

Les calculs répétitifs

- Je veux calculer S(1000). Faire le calcul à la main est facile, long et prodigieusement ennuyeux : 1 + 2 + 3 + 4 + 5 + 6 +7 + ... + 1000. Ouf.
- Or je ne veux pas FAIRE le calcul, je veux le FAIRE FAIRE par un ordinateur (***computer***). Il me faut donc écrire un programme **court** qui explique à la machine par quelles étapes passer. La taille du programme ne dépendra pas de n, mais le temps de calcul probablement.
- Il y a essentiellement trois manières *classiques* de construire notre algorithme de calcul de S(n) :

  - le ***calcul direct***
  - la ***récurrence***
  - la ***boucle*** ***ou*** ***itération***

Les calculs répétitifs : CALCUL DIRECT !

- Rarement possible, demande souvent un peu d'astuce, comme celle de Gauss vers ses 14 ans :

```
 S(n) = 1 + 2     + ... + n
+S(n) = n + (n-1) + ... + 1
```

$\Rightarrow$ `2 S(n) = n(n + 1)`

`S(n) = n(n + 1)/2`

- En Python :

IDLE

```
def S(n) :
    return n * (n + 1) // 2

print('S(1000) =',S(1000))
```

Run → `S(1000) = 500500`

- La taille du programme (longueur de son texte) ne dépend pas de n, qui ne sera fourni qu'à l'exécution. Le temps d'exécution du calcul de S(1000) est immédiat (il coûte 3 opérations).

Les calculs répétitifs : RECURRENCE

• Appréciée des matheux, elle permet souvent d'attaquer des problèmes difficiles en... supposant le problème résolu ! Cool.

• Il s'agit de réduire le problème S(n) à un problème S(k) avec k < n. On prend souvent k = n-1 ou n//2. On remarque ici que pour n > 0 :

$$S(n) = \underline{1 + 2 + \cdots + (n-1)} + n = S(n-1) + n$$

• Si l'on sait calculer S(n-1), on sait donc calculer S(n). Or on sait calculer S(1) = 1, d'où un calcul de proche en proche :

$$S(2) = S(1) + 2 = 1 + 2 = 3 \Rightarrow S(3) = S(2) + 3 = 3 + 3 = 6$$

• En Python, en supposant n ≥ 1 :

Le cas de base pourrait être `n == 0`

```
def S(n) :
    if n == 1 :
        return 1
    return S(n-1) + n
```

• **Hélas**, Python n'encourage pas la récurrence et ne l'optimise pas !

Les calculs répétitifs : BOUCLES

• Populaire chez les programmeurs, elle tâche de présenter le calcul comme une succession d'étapes identiques portant sur **des variables qui changent de valeur à chaque étape**.

~~*Qu'allons-nous faire ?*~~
~~*Comment procéder ?*~~

Où en sommes-nous ?
Quelle est la situation générale ?

• L'important est plus la **situation** que l'action !

• J'ai commencé à calculer S(n). En plein milieu du calcul, ***où en suis-je ?*** Par exemple, j'ai déjà calculé 1 + 2 + 3 + ... + i. Introduisons une variable acc représentant cette accumulation. Nous gérons donc deux variables i et acc. Il faudra alors savoir :

① passer à l'étape suivante (ITERATION)
② détecter si le calcul est terminé (TERMINAISON)
③ trouver les valeurs initiales de ces variables (INITIALISATION)

• Une fois la situation générale trouvée, voici **les trois temps de la construction d'une itération** :

① passer à l'étape suivante (ITERATION)

*la **situation générale***

Etant en possession de `acc = 1 + 2 + ... i`, on voudra obtenir la valeur de 1 + 2 + ... + (i+1). Il suffira donc d'ajouter i+1 à acc et ensuite d'ajouter 1 à i. *Ou bien d'ajouter 1 à i et ensuite i à acc.*

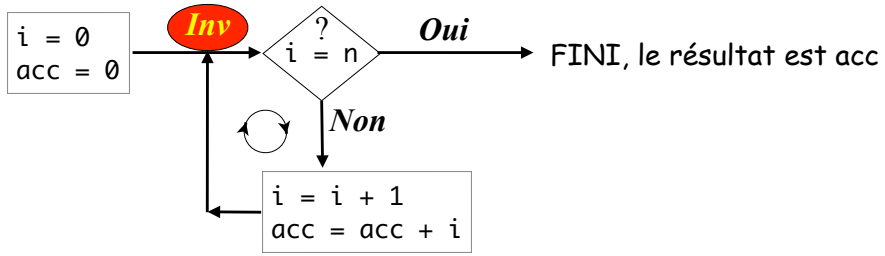
② détecter si le calcul est terminé (TERMINAISON)

On aura terminé lorsque i = n puisqu'alors acc = S(n).

③ trouver les valeurs initiales de ces variables (INITIALISATION)

Au début du calcul, je peux prendre i = 1 et acc =1. Je pourrais aussi prendre i = 0 et acc = 0... Le tout, c'est que ma situation générale acc = 1 + 2 + ... + i soit vraie en début de boucle.

*NB : **Si** l'une de ces étapes s'avère trop difficile, **alors** il faudra envisager de trouver une autre situation générale !*



• On visualise bien la **boucle** de calcul !

• A chaque entrée dans la boucle, au point Inv, la situation générale acc = 1 + 2 + ... i est vérifiée !

• On ***boucle*** tant que i < n. Une des manières d'exprimer une boucle consiste à utiliser le mot-clé **while** qui signifie "tant que".

```
def S(n) :
    i = 0
    acc = 0
    while i < n :
        i = i + 1
        acc = acc + i
    return acc
```

• **IMPORTANTE MISE EN GARDE**. La suite d'instructions :

```
i = i + 1        (1)
acc = acc + i    (2)
```

$\left|\begin{matrix} i = 2 \\ acc = 3 \end{matrix}\right. \Rightarrow \left|\begin{matrix} i = 3 \\ acc = 6 \end{matrix}\right.$

n'est PAS équivalente à celle-ci, qui serait fausse :

```
acc = acc + i    (3)
i = i + 1        (4)
```

$\left|\begin{matrix} i = 2 \\ acc = 3 \end{matrix}\right. \Rightarrow \left|\begin{matrix} i = 3 \\ acc = 5 \end{matrix}\right.$

• Il s'agit d'un ***point noir*** des boucles dans les langages classiques (C, Java, etc). Nous verrons plus tard une réponse élégante de Python à ce dilemne. Wait and see...

• Deux instructions peuvent ***commuter*** si elles sont indépendantes. Or ci-dessus, l'instruction (1) modifie la valeur de i dans (2). Alors que l'instruction (3) n'a aucun ***effet*** sur l'instruction (4).

• Deux instructions ne commutent donc pas *en général* !

• *Les variables* i *et* acc *sont des* ***variables locales*** *(cours 4 pages 19-21).*

• LA MISE AU POINT. Il arrive aux programmeurs de produire des programmes incorrects. Comment puis-je m'aider à détecter une erreur ? Réponse : en espionnant la boucle...

• On fait afficher les **variables de boucle** (les variables qui varient dans la boucle), ici acc et i. Observe-t-on une anomalie ?

```
while i < n :
    print('i =',i,'acc =',acc)
    .....
```

```
>>> S(4)
i = 0 acc = 0
i = 1 acc = 1
i = 2 acc = 3
i = 3 acc = 6
10
```

• Il est essentiel de s'assurer que le programme **termine** et n'entre pas dans une **boucle infinie** (brrr). Ce n'est pas toujours facile. Personne ne sait si la ***fonction de Collatz*** termine pour tout entier n positif !

```
def collatz(n) :
    while n != 1 :
        print(n,end=' ')
        if n % 2 == 0 : n = n // 2
        else : n = 3 * n + 1
```

```
>>> collatz(6)
6 3 10 5 16 8 4 2
```


Comment savoir si un nombre est premier ?

• Les **nombres premiers** 2, 3, 5, 7, 11... jouent un rôle important dans les codes secrets (espionnage, cartes bancaires, etc).

• Un entier n est toujours divisible par 1 et n, qui sont ses ***diviseurs triviaux***. Par exemple 12 est divisible par 1 et 12 (et par d'autres)...

• **Un entier n est premier s'il est ≥ 2 et si ses seuls diviseurs sont les diviseurs triviaux**. Par exemple 13 est premier, mais pas 12.

· **ALGORITHME**. Pour savoir si un nombre n ≥ 2 est premier, il suffit donc d'examiner les nombres entiers d ∈ [2,n-1] à la recherche d'un diviseur de n. On s'arrête au premier trouvé avec le résultat False. Si l'on n'en trouve aucun, le résultat sera True.

• Quelle est la **situation générale** ? J'en suis au nombre d que je n'ai pas encore examiné, et je n'ai toujours pas trouvé de diviseur.


```
def est_premier(n) :
    if n < 2 : return False      # 0 et 1 ne sont pas premiers
    d = 2                        # le premier candidat
    while d < n :                # aucun diviseur dans [2,d[
        if n % d == 0 : return False      # STOP !
        d = d + 1                # else implicite
    return True

print('2001 est premier :',est_premier(2001))
print('2003 est premier :',est_premier(2003))
```

Run

```
2001 est premier : False
2003 est premier : True
```

• Mais :

```
>>> est_premier(152734782063723562326183089878176004074l)
```

Ctrl-C Keyboard interrupt

Les nombres réels approchés

• Ou nombres réels **inexacts**. On parle de ***nombres flottants*** (`float`).

• Rappel : le calcul entier est exact, en précision infinie...

```
>>> 1 // 2
0
```

```
>>> 1 / 2
0.5
```

```
>>> 2 / 3
0.6666666666666666
```

• Les nombres réels approchés n'ont qu'un nombre limité de chiffres après la "virgule" (le point décimal). Donc aucun nombre irrationnel !

• On n'aura qu'une **approximation** de π ou de $\sqrt{2}$:

```
>>> from math import *
>>> pi
3.141592653589793
>>> type(pi)
<class 'float'>
```

```
>>> sqrt(2)
1.4142135623730951
>>> sqrt(2) ** 2
2.0000000000000004
>>> sqrt(2) ** 2 == 2
False
```


• Le problème vient du codage des nombres en **binaire**. Le nombre 0.1 par exemple est *simple* dans le système décimal mais possède une infinité de chiffres après la virgule dans le système binaire !

$0.0001100110011001100110011001100110011001100110011001100110011\ldots_2$

• Lorsque Python affiche une valeur approchée, ce n'est qu'une approximation de la véritable valeur internée dans la machine :

```
>>> 0.1      # quelle est la valeur de 0.1 ?
0.1          # ceci est une illusion !
```

• La fonction `Decimal` permet de voir (en décimal) la véritable représentation en machine de 0.1 qui n'est pas 0.1 mais :

```
>>> from decimal import Decimal
>>> Decimal(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

?

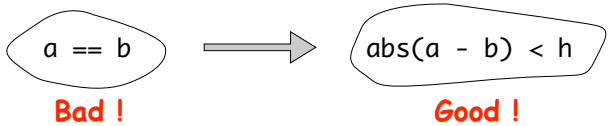
• Le calcul sur des nombres **approchés** étant par définition INEXACT, on évitera sous peine de surprises désagréables de questionner **l'égalité** en présence de nombres approchés !

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 0.7
True
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

• Le domaine du calcul **approché** est TRES difficile, et prévoir à l'avance le nombre exact de décimales correctes lors d'un résultat de calcul reste réservé aux spécialistes d'Analyse Numérique (brrr)...

• Alors que faire ? Réponse : remplacer l'égalité par une **précision** h :

Si a et b sont approchés :



Bad ! **Good !**

h = 0.001 par ex.

Exemple : approximation de $\sqrt{r}$

• Par la ***méthode des tangentes de Newton*** (1669).

• Soit à calculer la **racine carrée approchée d'un nombre réel r > 0**, par exemple calculer $\sqrt{2} = 1.4142\ldots$ sans utiliser `math.sqrt` !

• NEWTON : si a est une approximation de $\sqrt{r}$, alors :

$$b = \frac{1}{2}(a + \frac{r}{a})$$

est une approximation encore meilleure ! Pourquoi ? Cf TD ou page 19...

• Nous allons développer cet algorithme en répondant à trois questions (voir page 6) :

- comment *améliorer* l'approximation courante ? **ITERATION**
- mon approximation courante a est-elle *assez bonne* ? **TERMINAISON**
- comment initialiser la première approximation ? **INITIALISATION**

• Pour ***améliorer*** l'approximation, il suffit d'appliquer la formule de Newton, qui fait approcher a de $\sqrt{r}$:

```
a = 0.5 * (a + r / a)
```
ITERATION

• Mon approximation courante a est-elle ***assez bonne*** ? Elle est assez bonne lorsque a est très proche de $\sqrt{r}$. Notons h la variable dénotant la précision, par exemple `h = 0.001`

$$a \approx \sqrt{r} \;\Leftrightarrow\; a^2 \approx r \;\Leftrightarrow\; |a^2 - r| < h$$

• En Python : `abs(a*a - r) < h` TERMINAISON

• Comment ***initialiser*** l'approximation ? En fait, les maths sous-jacentes à la technique de Newton montrent qu'ici n'importe quel réel a > 0 convient :

```
a = 1
```
INITIALISATION

IDLE

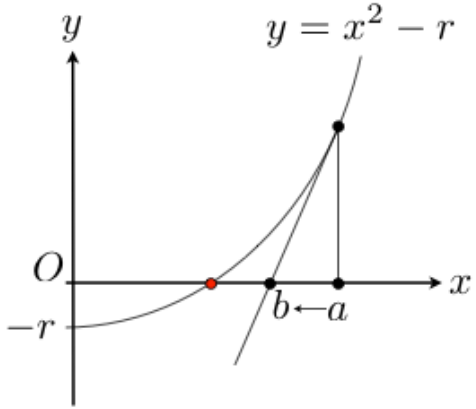
```
def rac2(r,h) :        # on suppose r > 0, et h > 0 petit
    a = 1
    while abs(a * a - r) >= h :
        print('a =',a)                    # juste pour visualiser !
        a = 0.5 * (a + r / a)
    return a

print('Racine carrée approchée de 2 :',rac2(2,0.0000001))
from math import sqrt
print('La vraie en Python : sqrt(2) =',sqrt(2))
```

Run

```
a = 1
a = 1.5
a = 1.4166666666666665
a = 1.4142156862745097
Racine carrée approchée de 2 : 1.4142135623746899
La vraie en Python : sqrt(2) = 1.4142135623730951
```
En 4 coups !

- Mais d'où vient la formule de Newton $b = \frac{1}{2}(a + \frac{r}{a})$?
- D'un simple calcul de tangentes (cf TD)...



Les boucles infinies et l'instruction break

• Il est essentiel dans une boucle `while` de s'assurer que la boucle **termine**, donc que le `<test>` finit par prendre la valeur `False`. Sinon le programme entre dans une ***boucle infinie*** et... on perd la main, il ***plante*** !

```
while <test> :
    <instruction>
    <instruction>
    .....
```

• Pourtant certains programmeurs optent pour un style de **boucle infinie** dans lequel la décision d'***échappement*** est prise parmi les instructions du corps de la boucle avec l'instruction `break`. Exemple :

```
while True :
    <instruction>
    <instruction>
    .....
```
dangereux...

```
x = 1
while x <= 5 :
    print(x)
    x = x + 1
```
⇔
```
x = 1
while True :
    if x > 5 : break
    print(x)
    x = x + 1
```
←

• L'instruction break provoque une ***sortie brutale de la boucle***, mais le programme continue son exécution ***après*** la boucle !

```
while True :
    <instr>
    if x > 5 : break
    <instr>
<instr>
```

• Quel intérêt ? Il se trouve que certaines boucles de ce type pourront avoir le test de sortie en ***milieu*** ou en ***fin*** de boucle. Il pourrait même y avoir plusieurs tests de sortie...

```
x = 1
while True :
    print(x,end=' ')
    if x == 5 : break
    print(',',end=' ')
    x = x + 1
print()
```

Run → `1 , 2 , 3 , 4 , 5`

• Cette boucle est donc la plus générale mais elle suppose que l'on garantisse bien sa terminaison, sinon GARE !

OPTIONNEL$_1$: Décomposer son code

• Du strict point de vue de la **méthodologie d'écriture des programmes**, il est souvent intéressant et apprécié des programmeurs qui lisent beaucoup de code, de découper une fonction en **sous-fonctions** réalisant des tâches déconnectées.

```
def solve(f,a,h) :
    def ameliore(a) :
        dfa = (f(a+h)-f(a))/h
        return a - f(a) / dfa
    def assez_bonne(a) :
        return abs(f(a)) < h
    while not assez_bonne(a) :
        a = ameliore(a)
    return a
```

les sous-fonctions

le programme principal

• Les paramètres f et h sont constants et visibles dans chacune des deux sous-fonctions.

OPTIONNEL$_2$: Programmer une stratégie !

• La stratégie de calcul de la racine carrée à la Newton sera retravaillée en TD. En regardant de près cette stratégie, elle se résume en :

a) Deviner une approximation de la solution
b) TANT QUE l'approximation n'est pas assez-bonne :
Améliorer l'approximation

• Ceci est une **stratégie de résolution de problème**. On peut passer les trois mots soulignés sous la forme de paramètres d'un **résolveur général de problèmes approchés** :

```
def solveur(approx,assez_bonne,améliore) :
    while not assez_bonne(approx) :
        approx = améliore(approx)
    return approx
```

• Pour résoudre un problème approché, il suffira de trouver la valeur approx et les fonctions améliore et assez_bonne.