



UNIVERSITÉ
CÔTE D'AZUR

Programmation impérative en Python

Cours 2. Boucles while et récursions

Olivier Baldellon

Courriel : `prenom.nom@univ-cotedazur.fr`

Page professionnelle : `https://upinfo.univ-cotedazur.fr/~obaldellon/`

LICENCE I — FACULTÉ DES SCIENCES ET INGÉNIERIE DE NICE — UNIVERSITÉ CÔTE D'AZUR

- 🍃 Partie I. Remarques générales
- 🍃 Partie II. Calculs répétitifs
- 🍃 Partie III. Boucles while
- 🍃 Partie IV. Les nombres premiers
- 🍃 Partie V. Les nombres flottants
- 🍃 Partie VI. Exemples
- 🍃 Partie VII. Pour aller plus loin
- 🍃 Partie VIII. Table des matières

- ▶ Beaucoup connaissent des choses qu'ils ont vu avant :
 - ▶ les listes,
 - ▶ les boucles `for`,
 - ▶ les f-string : `f"blabla"`

- ▶ Beaucoup connaissent des choses qu'ils ont vu avant :
 - ▶ les listes,
 - ▶ les boucles `for`,
 - ▶ les f-string : `f"blabla"`
- ▶ Un outil python est interdit tant que l'on ne l'a pas vu en cours !

- ▶ Beaucoup connaissent des choses qu'ils ont vu avant :
 - ▶ les listes,
 - ▶ les boucles `for`,
 - ▶ les f-string : `f"blabla"`
- ▶ Un outil python est interdit tant que l'on ne l'a pas vu en cours !
 - ▶ Le but est-il vraiment de connaître plein d'astuces en Python ?

- ▶ Beaucoup connaissent des choses qu'ils ont vu avant :
 - ▶ les listes,
 - ▶ les boucles `for`,
 - ▶ les `f-string` : `f"blabla"`
- ▶ Un outil python est interdit tant que l'on ne l'a pas vu en cours !
 - ▶ Le but est-il vraiment de connaître plein d'astuces en Python ?
 - ▶ Le but est d'apprendre à programmer.

- ▶ Beaucoup connaissent des choses qu'ils ont vu avant :
 - ▶ les listes,
 - ▶ les boucles `for`,
 - ▶ les `f-string` : `f"blabla"`
- ▶ Un outil python est interdit tant que l'on ne l'a pas vu en cours !
 - ▶ Le but est-il vraiment de connaître plein d'astuces en Python ?
 - ▶ Le but est d'apprendre à programmer.
- ▶ Nous prenons les notions dans l'ordre.
 - ▶ Il faut comprendre les méthodes générales et pénibles avant de prendre les raccourcis.

- ▶ Beaucoup connaissent des choses qu'ils ont vu avant :
 - ▶ les listes,
 - ▶ les boucles `for`,
 - ▶ les `f-string` : `f"blabla"`
- ▶ Un outil python est interdit tant que l'on ne l'a pas vu en cours !
 - ▶ Le but est-il vraiment de connaître plein d'astuces en Python ?
 - ▶ Le but est d'apprendre à programmer.
- ▶ Nous prenons les notions dans l'ordre.
 - ▶ Il faut comprendre les méthodes générales et pénibles avant de prendre les raccourcis.
 - ▶ Ces contraintes sont normales et ont un intérêt pédagogique.
 - ▶ Comment faire sans ?

- ▶ Beaucoup connaissent des choses qu'ils ont vu avant :
 - ▶ les listes,
 - ▶ les boucles `for`,
 - ▶ les `f-string` : `f"blabla"`
- ▶ Un outil python est interdit tant que l'on ne l'a pas vu en cours !
 - ▶ Le but est-il vraiment de connaître plein d'astuces en Python ?
 - ▶ Le but est d'apprendre à programmer.
- ▶ Nous prenons les notions dans l'ordre.
 - ▶ Il faut comprendre les méthodes générales et pénibles avant de prendre les raccourcis.
 - ▶ Ces contraintes sont normales et ont un intérêt pédagogique.
 - ▶ Comment faire sans ?
- ▶ Une fois l'UE terminée, faite ce que vous voulez !

- ▶ Je lis souvent des comparaisons de la forme :

```
if 0 < x < 1:  
    bla bla
```

SCRIPT

- ▶ Une telle expression a-t-elle un sens ?

- ▶ Je lis souvent des comparaisons de la forme :

```
if 0 < x < 1:  
    bla bla
```

SCRIPT

- ▶ Une telle expression a-t-elle un sens?
 - ▶ $1 + 2 + 3$

- ▶ Je lis souvent des comparaisons de la forme :

```
if 0 < x < 1:  
    bla bla
```

SCRIPT

- ▶ Une telle expression a-t-elle un sens ?
 - ▶ $1 + 2 + 3 \rightarrow (1 + 2) + 3$

- ▶ Je lis souvent des comparaisons de la forme :

```
if 0 < x < 1:  
    bla bla
```

SCRIPT

- ▶ Une telle expression a-t-elle un sens ?

▶ $1 + 2 + 3 \rightarrow (1 + 2) + 3 \rightarrow 3 + 3$

- ▶ Je lis souvent des comparaisons de la forme :

```
if 0 < x < 1:  
    bla bla
```

SCRIPT

- ▶ Une telle expression a-t-elle un sens ?

▶ $1 + 2 + 3 \rightarrow (1 + 2) + 3 \rightarrow 3 + 3 \rightarrow 6$

- ▶ Je lis souvent des comparaisons de la forme :

```
if 0 < x < 1:  
    bla bla
```

SCRIPT

- ▶ Une telle expression a-t-elle un sens ?

▶ $1 + 2 + 3 \rightarrow (1 + 2) + 3 \rightarrow 3 + 3 \rightarrow 6$

▶ $0 < x < 1$

- ▶ Je lis souvent des comparaisons de la forme :

```
if 0 < x < 1:  
    bla bla
```

SCRIPT

- ▶ Une telle expression a-t-elle un sens ?

▶ $1 + 2 + 3 \rightarrow (1 + 2) + 3 \rightarrow 3 + 3 \rightarrow 6$

▶ $0 < x < 1 \rightarrow (0 < x) < 1$

- ▶ Je lis souvent des comparaisons de la forme :

```
if 0 < x < 1:  
    bla bla
```

SCRIPT

- ▶ Une telle expression a-t-elle un sens ?

▶ $1 + 2 + 3 \rightarrow (1 + 2) + 3 \rightarrow 3 + 3 \rightarrow 6$

▶ $0 < x < 1 \rightarrow (0 < x) < 1 \rightarrow \text{True} < 1$

- ▶ Je lis souvent des comparaisons de la forme :

```
if 0 < x < 1:  
    bla bla
```

SCRIPT

- ▶ Une telle expression a-t-elle un sens ?

▶ $1 + 2 + 3 \rightarrow (1 + 2) + 3 \rightarrow 3 + 3 \rightarrow 6$

▶ $0 < x < 1 \rightarrow (0 < x) < 1 \rightarrow \text{True} < 1 \rightarrow ?$

- ▶ Je lis souvent des comparaisons de la forme :

```
if 0 < x < 1:  
    bla bla
```

SCRIPT

- ▶ Une telle expression a-t-elle un sens ?

▶ $1 + 2 + 3 \rightarrow (1 + 2) + 3 \rightarrow 3 + 3 \rightarrow 6$

▶ $0 < x < 1 \rightarrow (0 < x) < 1 \rightarrow \text{True} < 1 \rightarrow ?$

- ▶ Bon... pour être honnête Python y arrive très bien.
 - ▶ Mais ça ne marche pas avec les autres langages.
 - ▶ Mauvais habitude que je vous interdis (pour cette UE)

- ▶ Je lis souvent des comparaisons de la forme :

```
if 0 < x < 1:  
    bla bla
```

SCRIPT

- ▶ Une telle expression a-t-elle un sens ?

▶ $1 + 2 + 3 \rightarrow (1 + 2) + 3 \rightarrow 3 + 3 \rightarrow 6$

▶ $0 < x < 1 \rightarrow (0 < x) < 1 \rightarrow \text{True} < 1 \rightarrow ?$

- ▶ Bon... pour être honnête Python y arrive très bien.

- ▶ Mais ça ne marche pas avec les autres langages.
- ▶ Mauvais habitude que je vous interdis (pour cette UE)

- ▶ La bonne méthode :

```
if 0 < x and x < 1:  
    bla bla
```

SCRIPT

- Partie I. Remarques générales
- Partie II. Calculs répétitifs
- Partie III. Boucles while
- Partie IV. Les nombres premiers
- Partie V. Les nombres flottants
- Partie VI. Exemples
- Partie VII. Pour aller plus loin
- Partie VIII. Table des matières

► **Problème** : on souhaite calculer la somme $S(n)$ des entiers de 1 à n avec $n \geq 1$ entier.

$$S(n) = \sum_{k=1}^n k = 1 + 2 + \dots + n$$

- ▶ **Problème** : on souhaite calculer la somme $S(n)$ des entiers de 1 à n avec $n \geq 1$ entier.

$$S(n) = \sum_{k=1}^n k = 1 + 2 + \dots + n$$

- ▶ Il s'agit de construire un **algorithme** de calcul de $S(n)$: une méthode mécanique, étape par étape qui trouve le résultat correct.

- ▶ **Problème** : on souhaite calculer la somme $S(n)$ des entiers de 1 à n avec $n \geq 1$ entier.

$$S(n) = \sum_{k=1}^n k = 1 + 2 + \dots + n$$

- ▶ Il s'agit de construire un **algorithme** de calcul de $S(n)$: une méthode mécanique, étape par étape qui trouve le résultat correct.
- ▶ À partir de cet algorithme, on va coder la fonction S dans le langage choisi, ici Python.

- ▶ **Problème** : on souhaite calculer la somme $S(n)$ des entiers de 1 à n avec $n \geq 1$ entier.

$$S(n) = \sum_{k=1}^n k = 1 + 2 + \dots + n$$

- ▶ Il s'agit de construire un **algorithme** de calcul de $S(n)$: une méthode mécanique, étape par étape qui trouve le résultat correct.
- ▶ À partir de cet algorithme, on va coder la fonction S dans le langage choisi, ici Python.
- ▶ Ainsi, on obtient un **programme**, dont l'exécution sur ordinateur permet de calculer la fonction S .

- ▶ On veut calculer $S(1000)$. Calculer « à la main » sans utiliser d'astuce est facile, mais laborieux : $1 + 2 + 3 + \dots + 1000 = \dots$

- ▶ On veut calculer $S(1000)$. Calculer « à la main » sans utiliser d'astuce est facile, mais laborieux : $1 + 2 + 3 + \dots + 1000 = \dots$
- ▶ On veut donc le faire faire par un ordinateur.
 - ▶ la méthode doit détailler chaque étape du calcul (pas d'improvisation)
 - ▶ la méthode doit marcher pour tous les entiers n positif.

- ▶ On veut calculer $S(1000)$. Calculer « à la main » sans utiliser d'astuce est facile, mais laborieux : $1 + 2 + 3 + \dots + 1000 = \dots$

- ▶ On veut donc le faire faire par un ordinateur.
 - ▶ la méthode doit détailler chaque étape du calcul (pas d'improvisation)
 - ▶ la méthode doit marcher pour tous les entiers n positif.

- ▶ Trois manières classiques pour calculer $S(n)$:
 - ▶ le calcul direct,
 - ▶ la récurrence,
 - ▶ la boucle ou itération.

C'est la solution idéale, mais rarement possible.

- ▶ Nécessite une idée (un peu) astucieuse.
- ▶ Ici on a la formule :

$$S(n) = \frac{n(n+1)}{2}$$

C'est la solution idéale, mais rarement possible.

- ▶ Nécessite une idée (un peu) astucieuse.
- ▶ Ici on a la formule :

$$S(n) = \frac{n(n+1)}{2}$$

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

SCRIPT

>>>

SHELL

C'est la solution idéale, mais rarement possible.

- ▶ Nécessite une idée (un peu) astucieuse.
- ▶ Ici on a la formule :

$$S(n) = \frac{n(n+1)}{2}$$

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

SCRIPT

```
>>> S(1000)
```

SHELL

C'est la solution idéale, mais rarement possible.

- ▶ Nécessite une idée (un peu) astucieuse.
- ▶ Ici on a la formule :

$$S(n) = \frac{n(n+1)}{2}$$

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

SCRIPT

```
>>> S(1000)  
500500
```

SHELL

C'est la solution idéale, mais rarement possible.

- ▶ Nécessite une idée (un peu) astucieuse.
- ▶ Ici on a la formule :

$$S(n) = \frac{n(n+1)}{2}$$

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

SCRIPT

```
>>> S(1000)  
500500
```

SHELL

- ▶ Ce programme fonctionne pour toutes les valeurs de n.
- ▶ Il est de plus très rapide ; il ne nécessite que 3 opérations.

- ▶ Comme la récurrence mathématique, on résout un problème en le supposant résolu pour des cas plus simples.
- ▶ On veut calculer $S(n)$, on s'autorise à utiliser $S(k)$ pour $k < n$.

- ▶ Comme la récurrence mathématique, on résout un problème en le supposant résolu pour des cas plus simples.
- ▶ On veut calculer $S(n)$, on s'autorise à utiliser $S(k)$ pour $k < n$.

Cas typiques :

▶ $k = n - 1$

▶ $k = n // 2$

- ▶ Comme la récurrence mathématique, on résout un problème en le supposant résolu pour des cas plus simples.
- ▶ On veut calculer $S(n)$, on s'autorise à utiliser $S(k)$ pour $k < n$.

Cas typiques :

▶ $k = n - 1$

▶ $k = n // 2$

- ▶ On a :

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

- ▶ Comme la récurrence mathématique, on résout un problème en le supposant résolu pour des cas plus simples.
- ▶ On veut calculer $S(n)$, on s'autorise à utiliser $S(k)$ pour $k < n$.

Cas typiques :

▶ $k = n - 1$

▶ $k = n // 2$

- ▶ On a :

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

- ▶ Pour calculer $S(n)$, il suffit donc de calculer $S(n-1)$.

- ▶ Comme la récurrence mathématique, on résout un problème en le supposant résolu pour des cas plus simples.
- ▶ On veut calculer $S(n)$, on s'autorise à utiliser $S(k)$ pour $k < n$.

Cas typiques :

▶ $k = n - 1$

▶ $k = n // 2$

- ▶ On a :

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

- ▶ Pour calculer $S(n)$, il suffit donc de calculer $S(n-1)$.
- ▶ Et comment calculer $S(n-1)$? À partir de $S(n-2)$

- ▶ Comme la récurrence mathématique, on résout un problème en le supposant résolu pour des cas plus simples.
- ▶ On veut calculer $S(n)$, on s'autorise à utiliser $S(k)$ pour $k < n$.

Cas typiques :

▶ $k = n - 1$

▶ $k = n // 2$

- ▶ On a :

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

- ▶ Pour calculer $S(n)$, il suffit donc de calculer $S(n-1)$.
- ▶ Et comment calculer $S(n-1)$? À partir de $S(n-2)$
- ▶ Mais il faut s'arrêter : on peut calculer $S(1)$ directement.

$$S(5) = S(5)$$

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
- ▶ $S(4) = S(3) + 4$
- ▶ $S(3) = S(2) + 3$
- ▶ $S(2) = S(1) + 2$
- ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$

- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = S(4) + 5$$

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
- ▶ $S(4) = S(3) + 4$
- ▶ $S(3) = S(2) + 3$
- ▶ $S(2) = S(1) + 2$
- ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$

- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = S(4) + 5$$

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
- ▶ $S(4) = S(3) + 4$
- ▶ $S(3) = S(2) + 3$
- ▶ $S(2) = S(1) + 2$
- ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$

- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = (S(3) + 4) + 5$$

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
- ▶ $S(4) = S(3) + 4$
- ▶ $S(3) = S(2) + 3$
- ▶ $S(2) = S(1) + 2$
- ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$

- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = (S(3) + 4) + 5$$

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
- ▶ $S(4) = S(3) + 4$
- ▶ $S(3) = S(2) + 3$
- ▶ $S(2) = S(1) + 2$
- ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$
- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = ((S(2) + 3) + 4) + 5$$

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
- ▶ $S(4) = S(3) + 4$
- ▶ $S(3) = S(2) + 3$
- ▶ $S(2) = S(1) + 2$
- ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$

- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = ((S(2) + 3) + 4) + 5$$

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
- ▶ $S(4) = S(3) + 4$
- ▶ $S(3) = S(2) + 3$
- ▶ $S(2) = S(1) + 2$
- ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$

- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = (((S(1) + 2) + 3) + 4) + 5$$

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
- ▶ $S(4) = S(3) + 4$
- ▶ $S(3) = S(2) + 3$
- ▶ $S(2) = S(1) + 2$
- ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$

- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = (((S(1) + 2) + 3) + 4) + 5$$

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
- ▶ $S(4) = S(3) + 4$
- ▶ $S(3) = S(2) + 3$
- ▶ $S(2) = S(1) + 2$
- ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$

- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = (((1 + 2) + 3) + 4) + 5$$

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
- ▶ $S(4) = S(3) + 4$
- ▶ $S(3) = S(2) + 3$
- ▶ $S(2) = S(1) + 2$
- ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$

- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = (((1 + 2) + 3) + 4) + 5$$

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
- ▶ $S(4) = S(3) + 4$
- ▶ $S(3) = S(2) + 3$
- ▶ $S(2) = S(1) + 2$
- ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$

- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = ((3 + 3) + 4) + 5$$

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
- ▶ $S(4) = S(3) + 4$
- ▶ $S(3) = S(2) + 3$
- ▶ $S(2) = S(1) + 2$
- ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$
- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = ((3 + 3) + 4) + 5$$

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
 - ▶ $S(4) = S(3) + 4$
 - ▶ $S(3) = S(2) + 3$
 - ▶ $S(2) = S(1) + 2$
 - ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$
- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = (6 + 4) + 5$$

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
- ▶ $S(4) = S(3) + 4$
- ▶ $S(3) = S(2) + 3$
- ▶ $S(2) = S(1) + 2$
- ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$

- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = (6 + 4) + 5$$

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
- ▶ $S(4) = S(3) + 4$
- ▶ $S(3) = S(2) + 3$
- ▶ $S(2) = S(1) + 2$
- ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$

- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = 10 + 5$$

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
- ▶ $S(4) = S(3) + 4$
- ▶ $S(3) = S(2) + 3$
- ▶ $S(2) = S(1) + 2$
- ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$

- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = 10 + 5$$

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
 - ▶ $S(4) = S(3) + 4$
 - ▶ $S(3) = S(2) + 3$
 - ▶ $S(2) = S(1) + 2$
 - ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$
- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = 15$$

Ouf!

On remplace successivement, les valeurs $S(n)$

- ▶ $S(5) = S(4) + 5$
- ▶ $S(4) = S(3) + 4$
- ▶ $S(3) = S(2) + 3$
- ▶ $S(2) = S(1) + 2$
- ▶ Ici on peut s'arrêter car on a directement $S(1) = 1$

- ▶ On obtient alors $S(5) = (((S(1) + 2) + 3) + 4) + 5$

$$S(5) = S(5)$$

$$S(5) = S(4) + 5$$

$$S(5) = S(4) + 5$$

$$S(5) = (S(3) + 4) + 5$$

$$S(5) = (S(3) + 4) + 5$$

$$S(5) = ((S(2) + 3) + 4) + 5$$

$$S(5) = ((S(2) + 3) + 4) + 5$$

$$S(5) = (((S(1) + 2) + 3) + 4) + 5$$

$$S(5) = (((S(1) + 2) + 3) + 4) + 5$$

$$S(5) = (((1 + 2) + 3) + 4) + 5$$

$$S(5) = (((1 + 2) + 3) + 4) + 5$$

$$S(5) = ((3 + 3) + 4) + 5$$

$$S(5) = ((3 + 3) + 4) + 5$$

$$S(5) = (6 + 4) + 5$$

$$S(5) = (6 + 4) + 5$$

$$S(5) = 10 + 5$$

$$S(5) = 10 + 5$$

$$S(5) = 15$$

SCRIPT

```
def S(n):  
    if n == 1: # Cas d'arrêt  
        return 1  
    else:  
        return S(n-1) + n # Formule de récurrence
```

Dans un programme récursif **il faut** :

- ▶ Le cas d'arrêt
- ▶ La formule de récurrence

$$\begin{cases} S(1) &= 1 \\ S(n) &= S(n-1) + n \end{cases}$$

SCRIPT

```
def S(n):  
    if n == 1: # Cas d'arrêt  
        return 1  
    else:  
        return S(n-1) + n # Formule de récurrence
```

Dans un programme récursif **il faut** :

- ▶ Le cas d'arrêt
- ▶ La formule de récurrence

$$\begin{cases} S(1) &= 1 \\ S(n) &= S(n-1) + n \end{cases}$$

- ▶ Un simple **if else** suffit alors.

SCRIPT

```
def S(n):  
    if n == 1: # Cas d'arrêt  
        return 1  
    else:  
        return S(n-1) + n # Formule de récurrence
```

Dans un programme récursif **il faut** :

- ▶ Le cas d'arrêt
- ▶ La formule de récurrence

$$\begin{cases} S(1) &= 1 \\ S(n) &= S(n-1) + n \end{cases}$$

▶ Un simple **if else** suffit alors.

▶ Python est mauvais pour la récursion : à éviter quand il y a trop d'appels récursifs...

On considère la suite u_n arithmético-géométrique définie par récurrence :

$$\begin{cases} u_0 & = 234 \\ u_{n+1} & = -\frac{8}{10}u_n + 3 \end{cases}$$

- ▶ Écrire une fonction récursive pour calculer u_n .
- ▶ Quelle semble être sa limite ? L'écrire sous forme de fraction.

Indice :

On considère la suite u_n arithmético-géométrique définie par récurrence :

$$\begin{cases} u_0 &= 234 \\ u_{n+1} &= -\frac{8}{10}u_n + 3 \end{cases}$$

- ▶ Écrire une fonction récursive pour calculer u_n .
- ▶ Quelle semble être sa limite ? L'écrire sous forme de fraction.

Essayez de résoudre la question seul avant de regarder l'indice

Indice :

On considère la suite u_n arithmético-géométrique définie par récurrence :

$$\begin{cases} u_0 &= 234 \\ u_{n+1} &= -\frac{8}{10}u_n + 3 \end{cases}$$

- ▶ Écrire une fonction récursive pour calculer u_n .
- ▶ Quelle semble être sa limite ? L'écrire sous forme de fraction.

Essayez de résoudre la question seul avant de regarder l'indice

Je suis sérieux, sortez votre Python et au boulot !

Indice :

On considère la suite u_n arithmético-géométrique définie par récurrence :

$$\begin{cases} u_0 & = 234 \\ u_{n+1} & = -\frac{8}{10}u_n + 3 \end{cases}$$

- ▶ Écrire une fonction récursive pour calculer u_n .
- ▶ Quelle semble être sa limite ? L'écrire sous forme de fraction.

Essayez de résoudre la question seul avant de regarder l'indice

Je suis sérieux, sortez votre Python et au boulot ! Maintenant !

Indice :

On considère la suite u_n arithmético-géométrique définie par récurrence :

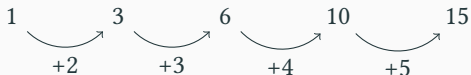
$$\begin{cases} u_0 & = 234 \\ u_{n+1} & = -\frac{8}{10}u_n + 3 \end{cases}$$

- ▶ Écrire une fonction récursive pour calculer u_n .
- ▶ Quelle semble être sa limite ? L'écrire sous forme de fraction.

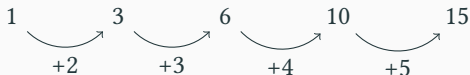
Indice : La récurrence peut s'écrire sous la forme $u_n = -0.8u_{n-1} + 3$

- 🌿 Partie I. Remarques générales
- 🌿 Partie II. Calculs répétitifs
- 🌿 **Partie III. Boucles while**
- 🌿 Partie IV. Les nombres premiers
- 🌿 Partie V. Les nombres flottants
- 🌿 Partie VI. Exemples
- 🌿 Partie VII. Pour aller plus loin
- 🌿 Partie VIII. Table des matières

- ▶ Les boucles forment un mécanisme bien plus répandu pour le calcul.
- ▶ Pour calculer de tête $1 + 2 + \dots + 5$ on passe par les résultats intermédiaires suivants :



- ▶ Les boucles forment un mécanisme bien plus répandu pour le calcul.
- ▶ Pour calculer de tête $1 + 2 + \dots + 5$ on passe par les résultats intermédiaires suivants :

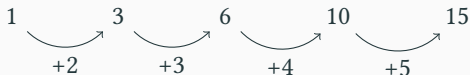


Variable somme

Variable i

- ▶ Il suffit de deux variables pour faire le calcul
 - ▶ Celle correspondant à la somme partielle :
 - ▶ Celle correspondant à ce qu'on ajoute :

- ▶ Les boucles forment un mécanisme bien plus répandu pour le calcul.
- ▶ Pour calculer de tête $1 + 2 + \dots + 5$ on passe par les résultats intermédiaires suivants :



Variable somme

Variable i

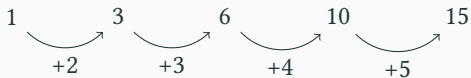
- ▶ Il suffit de deux variables pour faire le calcul
 - ▶ Celle correspondant à la somme partielle :
 - ▶ Celle correspondant à ce qu'on ajoute :

```
somme = somme+i  
i = i+1
```

SCRIPT

Variable somme

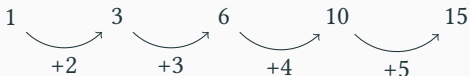
Variable i



somme	1	3	6	10	15
i	2	3	4	5	6

Variable somme

Variable i



somme	1	3	6	10	15
i	2	3	4	5	6

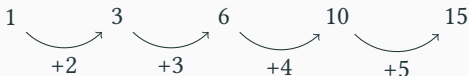
- ▶ Il faut créer les variables : **Initialisation**

```
somme=1
```

```
i=2
```


Variable somme

Variable i



somme	1	3	6	10	15
i	2	3	4	5	6

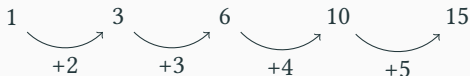
- ▶ Il faut créer les variables : **Initialisation**
- ▶ Préciser le calcul que l'on répétera : **Corps de boucle**

```
somme=1  
i=2
```

```
somme=somme+i  
i=i+1
```

Variable somme

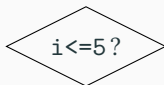
Variable i



somme	1	3	6	10	15
i	2	3	4	5	6

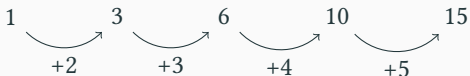
- ▶ Il faut créer les variables : **Initialisation**
- ▶ Préciser le calcul que l'on répétera : **Corps de boucle**
- ▶ Tester si l'on doit répéter le calcul ou non : **Test**

```
somme=1  
i=2
```



```
somme=somme+i  
i=i+1
```

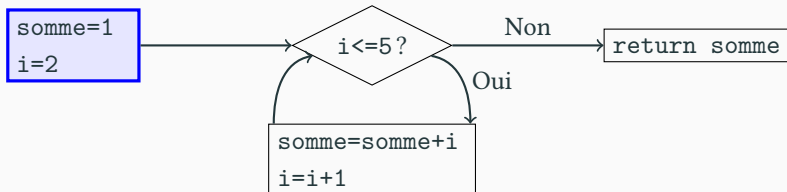
Variable somme



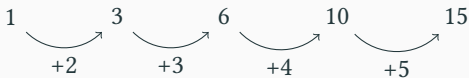
Variable i

somme	1	3	6	10	15
i	2	3	4	5	6

- ▶ Il faut créer les variables : **Initialisation**
- ▶ Préciser le calcul que l'on répétera : **Corps de boucle**
- ▶ Tester si l'on doit répéter le calcul ou non : **Test**



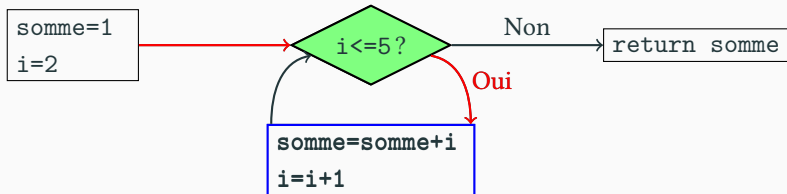
Variable somme



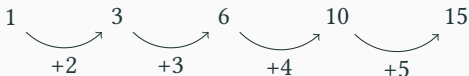
Variable i

somme	1	3	6	10	15
i	2	3	4	5	6

- ▶ Il faut créer les variables : **Initialisation**
- ▶ Préciser le calcul que l'on répétera : **Corps de boucle**
- ▶ Tester si l'on doit répéter le calcul ou non : **Test**



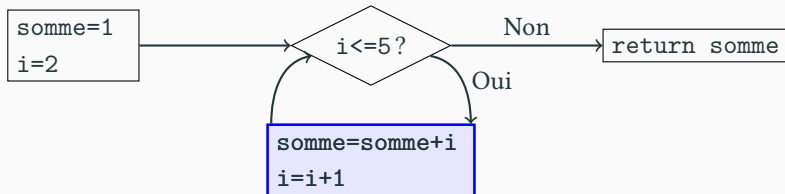
Variable somme



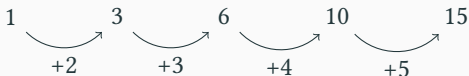
Variable i

somme	1	3	6	10	15
i	2	3	4	5	6

- ▶ Il faut créer les variables : **Initialisation**
- ▶ Préciser le calcul que l'on répétera : **Corps de boucle**
- ▶ Tester si l'on doit répéter le calcul ou non : **Test**



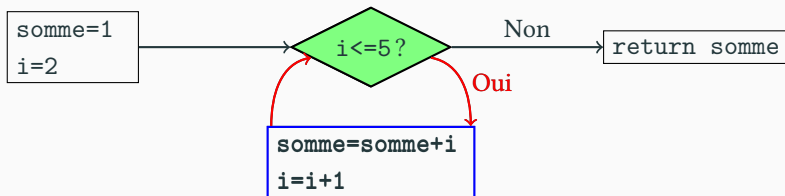
Variable somme



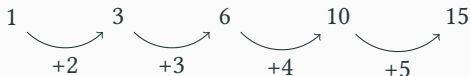
Variable i

somme	1	3	6	10	15
i	2	3	4	5	6

- ▶ Il faut créer les variables : **Initialisation**
- ▶ Préciser le calcul que l'on répétera : **Corps de boucle**
- ▶ Tester si l'on doit répéter le calcul ou non : **Test**



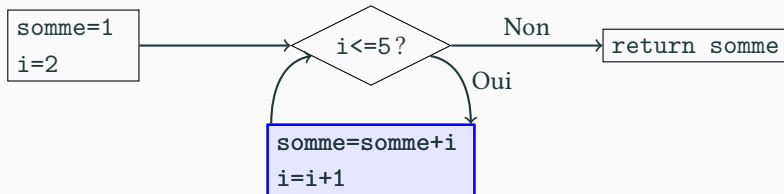
Variable somme



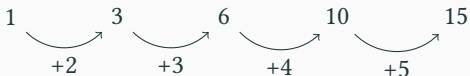
Variable i

somme	1	3	6	10	15
i	2	3	4	5	6

- ▶ Il faut créer les variables : **Initialisation**
- ▶ Préciser le calcul que l'on répétera : **Corps de boucle**
- ▶ Tester si l'on doit répéter le calcul ou non : **Test**



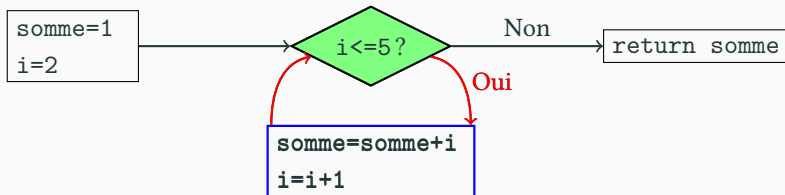
Variable somme



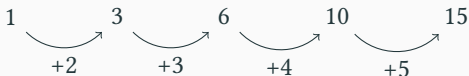
Variable i

somme	1	3	6	10	15
i	2	3	4	5	6

- ▶ Il faut créer les variables : **Initialisation**
- ▶ Préciser le calcul que l'on répétera : **Corps de boucle**
- ▶ Tester si l'on doit répéter le calcul ou non : **Test**



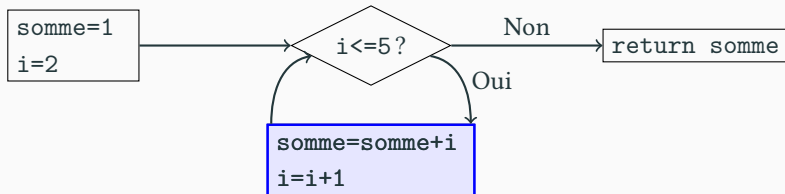
Variable somme



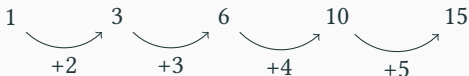
Variable i

somme	1	3	6	10	15
i	2	3	4	5	6

- ▶ Il faut créer les variables : **Initialisation**
- ▶ Préciser le calcul que l'on répétera : **Corps de boucle**
- ▶ Tester si l'on doit répéter le calcul ou non : **Test**



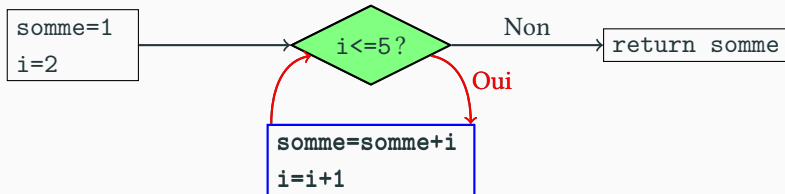
Variable somme



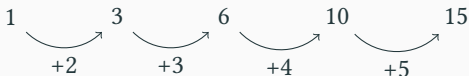
Variable i

somme	1	3	6	10	15
i	2	3	4	5	6

- ▶ Il faut créer les variables : **Initialisation**
- ▶ Préciser le calcul que l'on répétera : **Corps de boucle**
- ▶ Tester si l'on doit répéter le calcul ou non : **Test**



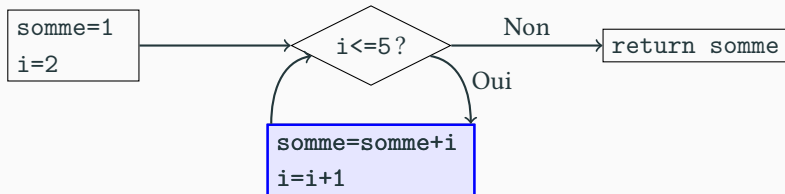
Variable somme



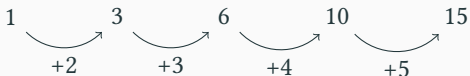
Variable i

somme	1	3	6	10	15
i	2	3	4	5	6

- ▶ Il faut créer les variables : **Initialisation**
- ▶ Préciser le calcul que l'on répétera : **Corps de boucle**
- ▶ Tester si l'on doit répéter le calcul ou non : **Test**



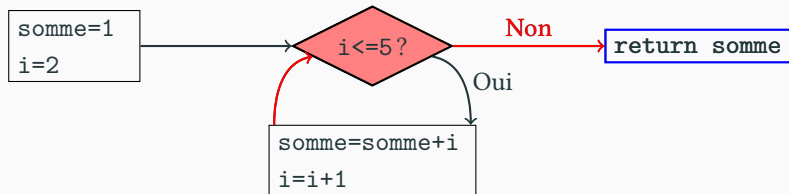
Variable somme



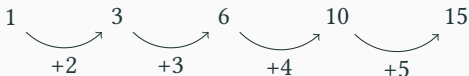
Variable i

somme	1	3	6	10	15
i	2	3	4	5	6

- ▶ Il faut créer les variables : **Initialisation**
- ▶ Préciser le calcul que l'on répétera : **Corps de boucle**
- ▶ Tester si l'on doit répéter le calcul ou non : **Test**



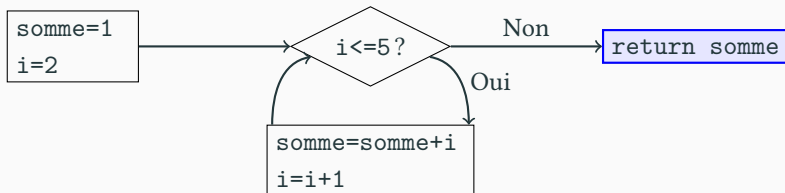
Variable somme

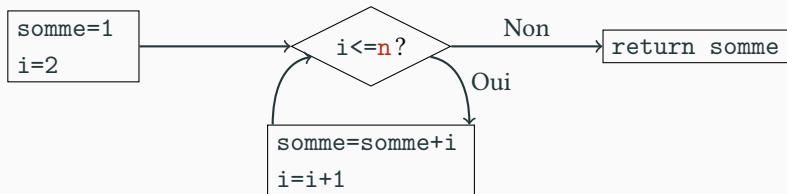


Variable i

somme	1	3	6	10	15
i	2	3	4	5	6

- ▶ Il faut créer les variables : **Initialisation**
- ▶ Préciser le calcul que l'on répétera : **Corps de boucle**
- ▶ Tester si l'on doit répéter le calcul ou non : **Test**





- Pour cela on utilise une boucle while (tant que)

```
def S(n):  
    somme=1           # initialisation  
    i=2              # initialisation  
    while i<=n:     # condition  
        somme=somme+i # corps de boucle  
        i=i+1       # corps de boucle  
    return somme
```

SCRIPT

- ▶ L'ordre des instructions compte ! On commence avec $i=2$ et $somme=1$.

```
somme = somme+i  
i = i+1
```

SCRIPT

```
i = i+1  
somme = somme+i
```

SCRIPT

▶ L'ordre des instructions compte ! On commence avec $i=2$ et $somme=1$.

```
somme = somme+i  
i = i+1
```

SCRIPT

```
i = i+1  
somme = somme+i
```

SCRIPT

▶ $i==2$ et $somme==3$

▶ L'ordre des instructions compte ! On commence avec $i=2$ et $somme=1$.

```
somme = somme+i  
i = i+1
```

SCRIPT

- ▶ $i==2$ et $somme==3$
- ▶ $i==3$ et $somme==3$

```
i = i+1  
somme = somme+i
```

SCRIPT

▶ L'ordre des instructions compte ! On commence avec $i=2$ et $somme=1$.

```
somme = somme+i  
i = i+1
```

SCRIPT

- ▶ $i==2$ et $somme==3$
- ▶ $i==3$ et $somme==3$

```
i = i+1  
somme = somme+i
```

SCRIPT

- ▶ $i==3$ et $somme==1$

▶ L'ordre des instructions compte ! On commence avec $i=2$ et $somme=1$.

```
somme = somme+i  
i = i+1
```

SCRIPT

- ▶ $i==2$ et $somme==3$
- ▶ $i==3$ et $somme==3$

```
i = i+1  
somme = somme+i
```

SCRIPT

- ▶ $i==3$ et $somme==1$
- ▶ $i==3$ et $somme==4$

- ▶ L'ordre des instructions compte ! On commence avec $i=2$ et $somme=1$.

```
somme = somme+i  
i = i+1
```

SCRIPT

- ▶ $i==2$ et $somme==3$
- ▶ $i==3$ et $somme==3$

```
i = i+1  
somme = somme+i
```

SCRIPT

- ▶ $i==3$ et $somme==1$
- ▶ $i==3$ et $somme==4$

- ▶ Pour comprendre ce que fait une boucle, on peut utiliser des `print`.

```
def S(n):  
    somme=1  
    i=2  
    while i<=n:  
        i=i+1  
        somme=somme+i  
        print('i =',i, 'et somme =',somme)  
    return somme
```

SCRIPT

>>>

SHELL

- ▶ L'ordre des instructions compte ! On commence avec $i=2$ et $somme=1$.

```
somme = somme+i  
i = i+1
```

SCRIPT

- ▶ $i==2$ et $somme==3$
- ▶ $i==3$ et $somme==3$

```
i = i+1  
somme = somme+i
```

SCRIPT

- ▶ $i==3$ et $somme==1$
- ▶ $i==3$ et $somme==4$

- ▶ Pour comprendre ce que fait une boucle, on peut utiliser des `print`.

```
def S(n):  
    somme=1  
    i=2  
    while i<=n:  
        i=i+1  
        somme=somme+i  
        print('i =',i, 'et somme =',somme)  
    return somme
```

SCRIPT

```
>>> S(5)
```

SHELL

- ▶ L'ordre des instructions compte ! On commence avec $i=2$ et $somme=1$.

```
somme = somme+i  
i = i+1
```

SCRIPT

- ▶ $i==2$ et $somme==3$
- ▶ $i==3$ et $somme==3$

```
i = i+1  
somme = somme+i
```

SCRIPT

- ▶ $i==3$ et $somme==1$
- ▶ $i==3$ et $somme==4$

- ▶ Pour comprendre ce que fait une boucle, on peut utiliser des `print`.

```
def S(n):  
    somme=1  
    i=2  
    while i<=n:  
        i=i+1  
        somme=somme+i  
        print('i =',i, 'et somme =',somme)  
    return somme
```

SCRIPT

```
>>> S(5)  
i = 3 et somme = 4  
i = 4 et somme = 8  
i = 5 et somme = 13  
i = 6 et somme = 19  
19
```

SHELL

- ▶ La condition est fondamentale ! C'est une grande source de bogue.
- ▶ Si la condition est toujours vérifiée, on ne sort jamais de la boucle.

```
i=3  
while i<=5:  
    i=i-1
```

SCRIPT

- ▶ Une seule solution :

- ▶ La condition est fondamentale! C'est une grande source de bogue.
- ▶ Si la condition est toujours vérifiée, on ne sort jamais de la boucle.

```
i=3  
while i<=5:  
    i=i-1
```

SCRIPT

- ▶ Une seule solution : la manifestation!

- ▶ La condition est fondamentale ! C'est une grande source de bogue.
- ▶ Si la condition est toujours vérifiée, on ne sort jamais de la boucle.

```
i=3
while i<=5:
    i=i-1
```

SCRIPT

- ▶ Une seule solution : ~~la manifestation~~ ! le raccourci **Ctrl** + **C**

- ▶ La condition est fondamentale ! C'est une grande source de bogue.
- ▶ Si la condition est toujours vérifiée, on ne sort jamais de la boucle.

```
i=3
while i<=5:
    i=i-1
```

SCRIPT

- ▶ Une seule solution : ~~la manifestation~~ ! le raccourci **Ctrl** + **C**

Synthèse

```
initialisation # Initialisation des
initialisation # variables de boucle
while conditions:
    instruction # Corps de boucle exécuté
    instruction # tant que le Test est valide
instruction # Lignes exécutées
instruction # dans tous les cas
```

SCRIPT

Écrire une fonction `table_multiplication(n)` qui affiche la table de n.

```
>>>
```

```
SHELL
```

Écrire une fonction `table_multiplication(n)` qui affiche la table de `n`.

```
>>> table_multiplication(11)
```

SHELL

Écrire une fonction `table_multiplication(n)` qui affiche la table de `n`.

```
>>> table_multiplication(11)
0 * 11 = 0
1 * 11 = 11
2 * 11 = 22
3 * 11 = 33
4 * 11 = 44
5 * 11 = 55
6 * 11 = 66
7 * 11 = 77
8 * 11 = 88
9 * 11 = 99
10 * 11 = 110
```

SHELL

- Partie I. Remarques générales
- Partie II. Calculs répétitifs
- Partie III. Boucles while
- Partie IV. Les nombres premiers
- Partie V. Les nombres flottants
- Partie VI. Exemples
- Partie VII. Pour aller plus loin
- Partie VIII. Table des matières

- ▶ On dit qu'un entier naturel n est premier s'il a exactement deux diviseurs positifs : 1 et n .
 - ▶ Exemples : 2, 3, 5, 7, 11, 13, 17, 19, ...
- ▶ Il y a une infinité de nombres premiers (*cf.* Euclide).

- ▶ On dit qu'un entier naturel n est premier s'il a exactement deux diviseurs positifs : 1 et n .
 - ▶ Exemples : 2, 3, 5, 7, 11, 13, 17, 19, ...
- ▶ Il y a une infinité de nombres premiers (*cf.* Euclide).
- ▶ Si n n'est pas premier,
 - ▶ alors n admet une diviseur d , avec $1 < d < n$.
 - ▶ dans ce cas, puisque d divise n on a $n \% d == 0$ ($d \neq 0$).

- ▶ On dit qu'un entier naturel n est premier s'il a exactement deux diviseurs positifs : 1 et n .
 - ▶ Exemples : 2, 3, 5, 7, 11, 13, 17, 19, ...
- ▶ Il y a une infinité de nombres premiers (*cf.* Euclide).
- ▶ Si n n'est pas premier,
 - ▶ alors n admet une diviseur d , avec $1 < d < n$.
 - ▶ dans ce cas, puisque d divise n on a $n \% d == 0$ ($d \neq 0$).
- ▶ Exemple : 2021027 n'est pas premier. En effet,

```
>>>
```

```
SHELL
```

- ▶ On dit qu'un entier naturel n est premier s'il a exactement deux diviseurs positifs : 1 et n .
 - ▶ Exemples : 2, 3, 5, 7, 11, 13, 17, 19, ...
- ▶ Il y a une infinité de nombres premiers (*cf.* Euclide).
- ▶ Si n n'est pas premier,
 - ▶ alors n admet une diviseur d , avec $1 < d < n$.
 - ▶ dans ce cas, puisque d divise n on a $n \% d == 0$ ($d \neq 0$).
- ▶ Exemple : 2021027 n'est pas premier. En effet,

```
>>> 2021027 % 1009
```

```
SHELL
```

- ▶ On dit qu'un entier naturel n est premier s'il a exactement deux diviseurs positifs : 1 et n .
 - ▶ Exemples : 2, 3, 5, 7, 11, 13, 17, 19, ...
- ▶ Il y a une infinité de nombres premiers (*cf.* Euclide).
- ▶ Si n n'est pas premier,
 - ▶ alors n admet une diviseur d , avec $1 < d < n$.
 - ▶ dans ce cas, puisque d divise n on a $n \% d == 0$ ($d \neq 0$).
- ▶ Exemple : 2021027 n'est pas premier. En effet,

```
>>> 2021027 % 1009  
0
```

SHELL

- ▶ Tester si un nombre d divise n pour $1 < d < n$.

```
def est_premier(n):  
    if n < 2:  
        return False  
    d = 2  
    while n % d != 0:  
        d = d + 1  
    # on est sorti de la boucle, si c'est pour d == n,  
    # alors n est premier  
    return d == n
```

SCRIPT

- ▶ Tester si un nombre d divise n pour $1 < d < n$.

SCRIPT

```
def est_premier(n):  
    if n < 2:  
        return False  
    d = 2  
    while n % d != 0:  
        d = d + 1  
    # on est sorti de la boucle, si c'est pour d == n,  
    # alors n est premier  
    return d == n
```

- ▶ Problème : lent car pas malin (on teste trop de diviseurs d).

- ▶ (1) Si n est pair, il est premier seulement si $n = 2$

- ▶ (1) Si n est pair, il est premier seulement si $n = 2$
- ▶ (2) Si n est impair, ses seuls diviseurs sont impairs.

- ▶ (1) Si n est pair, il est premier seulement si $n = 2$
 - ▶ (2) Si n est impair, ses seuls diviseurs sont impairs.
- ▶ si n n'est pas premier,
 - ▶ il possède un plus petit diviseur premier d
 - ▶ n s'écrit alors $n = d \cdot m$ avec $d \leq m$.
 - ▶ $d \cdot d \leq d \cdot m = n$

- ▶ (1) Si n est pair, il est premier seulement si $n = 2$
- ▶ (2) Si n est impair, ses seuls diviseurs sont impairs.

- ▶ si n n'est pas premier,
 - ▶ il possède un plus petit diviseur premier d
 - ▶ n s'écrit alors $n = d \cdot m$ avec $d \leq m$.
 - ▶ $d \cdot d \leq d \cdot m = n$

```
def est_premier(n):  
    if n < 2:  
        return False  
    elif n%2 == 0:  
        return n == 2 # Astuce (1)  
    else:  
        d = 3 # on test seulement les d impairs (2)  
        while d*d <= n and n%d != 0:  
            d = d+2 # si d est impair, d+2 est le suivant (2)  
  
        # on est sorti de la boucle, si d*d > n,  
        # alors n est premier  
        return d*d > n
```

SCRIPT

- ▶ (1) Si n est pair, il est premier seulement si $n = 2$
- ▶ (2) Si n est impair, ses seuls diviseurs sont impairs.
- ▶ si n n'est pas premier,
 - ▶ il possède un plus petit diviseur premier d
 - ▶ n s'écrit alors $n = d \cdot m$ avec $d \leq m$.
 - ▶ $d \cdot d \leq d \cdot m = n$

SCRIPT

```
def est_premier(n):  
    if n < 2:  
        return False  
    elif n%2 == 0:  
        return n == 2 # Astuce (1)  
    else:  
        d = 3 # on test seulement les d impairs (2)  
        while d*d <= n and n%d != 0:  
            d = d+2 # si d est impair, d+2 est le suivant (2)  
  
        # on est sorti de la boucle, si d*d > n,  
        # alors n est premier  
        return d*d > n
```

- ▶ Mieux mais loin d'être optimal. Il existe des tests probabilistes ou déterministes efficaces. Le module sympy fournit la fonction `isprime`.

- ▶ Pour un entier $n > 1$, on cherche le plus petit $d > 1$ tel que d divise n .

- Pour un entier $n > 1$, on cherche le plus petit $d > 1$ tel que d divise n .

```
def plus_petit_facteur(n):  
    d = 2  
    while n%d != 0:  
        d = d + 1  
    return d
```

SCRIPT

- Pour un entier $n > 1$, on cherche le plus petit $d > 1$ tel que d divise n .

```
def plus_petit_facteur(n):  
    d = 2  
    while n%d != 0:  
        d = d + 1  
    return d
```

SCRIPT

- Remarque `plus_petit_facteur(n)` renvoie un nombre premier.

- ▶ Pour un entier $n > 1$, on cherche le plus petit $d > 1$ tel que d divise n .

```
def plus_petit_facteur(n):  
    d = 2  
    while n%d != 0:  
        d = d + 1  
    return d
```

SCRIPT

- ▶ Remarque `plus_petit_facteur(n)` renvoie un nombre premier.
- ▶ Comme pour tester si n est premier, cet algorithme est naïf. On peut l'améliorer avec les mêmes techniques que précédemment.

- 🌿 Partie I. Remarques générales
- 🌿 Partie II. Calculs répétitifs
- 🌿 Partie III. Boucles while
- 🌿 Partie IV. Les nombres premiers
- 🌿 **Partie V. Les nombres flottants**
- 🌿 Partie VI. Exemples
- 🌿 Partie VII. Pour aller plus loin
- 🌿 Partie VIII. Table des matières

- ▶ On code les décimaux avec la notation à virgule flottante :
 - ▶ Il est impossible de représenter de façon exacte tous les nombres réels.
 - ▶ On parle de flottants (pour nombre à virgule flottante). En anglais : *float*.

```
>>>
```

```
SHELL
```


- ▶ On code les décimaux avec la notation à virgule flottante :
 - ▶ Il est impossible de représenter de façon exacte tous les nombres réels.
 - ▶ On parle de flottants (pour nombre à virgule flottante). En anglais : *float*.

```
>>> 5/2
```

SHELL

- ▶ On code les décimaux avec la notation à virgule flottante :
 - ▶ Il est impossible de représenter de façon exacte tous les nombres réels.
 - ▶ On parle de flottants (pour nombre à virgule flottante). En anglais : *float*.

```
>>> 5/2  
2.5  
>>>
```

SHELL

- ▶ On code les décimaux avec la notation à virgule flottante :
 - ▶ Il est impossible de représenter de façon exacte tous les nombres réels.
 - ▶ On parle de flottants (pour nombre à virgule flottante). En anglais : *float*.

```
>>> 5/2  
2.5  
>>> 2/3
```

SHELL

- ▶ On code les décimaux avec la notation à virgule flottante :
 - ▶ Il est impossible de représenter de façon exacte tous les nombres réels.
 - ▶ On parle de flottants (pour nombre à virgule flottante). En anglais : *float*.

```
>>> 5/2
2.5
>>> 2/3
0.6666666666666666
>>>
```

SHELL

- ▶ On code les décimaux avec la notation à virgule flottante :
 - ▶ Il est impossible de représenter de façon exacte tous les nombres réels.
 - ▶ On parle de flottants (pour nombre à virgule flottante). En anglais : *float*.

SHELL

```
>>> 5/2
2.5
>>> 2/3
0.6666666666666666
>>> 2.5 * 10**50 # Les puissances de 10 sont notées avec e+
```

- ▶ On code les décimaux avec la notation à virgule flottante :
 - ▶ Il est impossible de représenter de façon exacte tous les nombres réels.
 - ▶ On parle de flottants (pour nombre à virgule flottante). En anglais : *float*.

```
>>> 5/2
2.5
>>> 2/3
0.6666666666666666
>>> 2.5 * 10**50 # Les puissances de 10 sont notées avec e+
2.5e+50
```

SHELL

- ▶ On code les décimaux avec la notation à virgule flottante :
 - ▶ Il est impossible de représenter de façon exacte tous les nombres réels.
 - ▶ On parle de flottants (pour nombre à virgule flottante). En anglais : *float*.

```
>>> 5/2
2.5
>>> 2/3
0.6666666666666666
>>> 2.5 * 10**50 # Les puissances de 10 sont notées avec e+
2.5e+50
```

SHELL

- ▶ On utilise un point au lieu d'une virgule

- ▶ On code les décimaux avec la notation à virgule flottante :
 - ▶ Il est impossible de représenter de façon exacte tous les nombres réels.
 - ▶ On parle de flottants (pour nombre à virgule flottante). En anglais : *float*.

```
>>> 5/2
2.5
>>> 2/3
0.6666666666666666
>>> 2.5 * 10**50 # Les puissances de 10 sont notées avec e+
2.5e+50
```

SHELL

- ▶ On utilise un point au lieu d'une virgule
- ▶ Les flottants n'ont qu'un nombre limité de chiffres après la virgule.

- ▶ On code les décimaux avec la notation à virgule flottante :
 - ▶ Il est impossible de représenter de façon exacte tous les nombres réels.
 - ▶ On parle de flottants (pour nombre à virgule flottante). En anglais : *float*.

```
>>> 5/2
2.5
>>> 2/3
0.6666666666666666
>>> 2.5 * 10**50 # Les puissances de 10 sont notées avec e+
2.5e+50
```

SHELL

- ▶ On utilise un point au lieu d'une virgule
- ▶ Les flottants n'ont qu'un nombre limité de chiffres après la virgule.
- ▶ On n'aura qu'une approximation de π et $\sqrt{2}$. Ce qui largement suffisant pour des calculs pratiques.

```
>>>
```

SHELL

- ▶ On code les décimaux avec la notation à virgule flottante :
 - ▶ Il est impossible de représenter de façon exacte tous les nombres réels.
 - ▶ On parle de flottants (pour nombre à virgule flottante). En anglais : *float*.

```
>>> 5/2
2.5
>>> 2/3
0.6666666666666666
>>> 2.5 * 10**50 # Les puissances de 10 sont notées avec e+
2.5e+50
```

SHELL

- ▶ On utilise un point au lieu d'une virgule
- ▶ Les flottants n'ont qu'un nombre limité de chiffres après la virgule.
- ▶ On n'aura qu'une approximation de π et $\sqrt{2}$. Ce qui largement suffisant pour des calculs pratiques.

```
>>> import math
```

SHELL

- ▶ On code les décimaux avec la notation à virgule flottante :
 - ▶ Il est impossible de représenter de façon exacte tous les nombres réels.
 - ▶ On parle de flottants (pour nombre à virgule flottante). En anglais : *float*.

```
>>> 5/2
2.5
>>> 2/3
0.6666666666666666
>>> 2.5 * 10**50 # Les puissances de 10 sont notées avec e+
2.5e+50
```

SHELL

- ▶ On utilise un point au lieu d'une virgule
- ▶ Les flottants n'ont qu'un nombre limité de chiffres après la virgule.
- ▶ On n'aura qu'une approximation de π et $\sqrt{2}$. Ce qui largement suffisant pour des calculs pratiques.

```
>>> import math
>>>
```

SHELL

- ▶ On code les décimaux avec la notation à virgule flottante :
 - ▶ Il est impossible de représenter de façon exacte tous les nombres réels.
 - ▶ On parle de flottants (pour nombre à virgule flottante). En anglais : *float*.

```
>>> 5/2
2.5
>>> 2/3
0.6666666666666666
>>> 2.5 * 10**50 # Les puissances de 10 sont notées avec e+
2.5e+50
```

SHELL

- ▶ On utilise un point au lieu d'une virgule
- ▶ Les flottants n'ont qu'un nombre limité de chiffres après la virgule.
- ▶ On n'aura qu'une approximation de π et $\sqrt{2}$. Ce qui largement suffisant pour des calculs pratiques.

```
>>> import math
>>> math.pi
```

SHELL

- ▶ On code les décimaux avec la notation à virgule flottante :
 - ▶ Il est impossible de représenter de façon exacte tous les nombres réels.
 - ▶ On parle de flottants (pour nombre à virgule flottante). En anglais : *float*.

```
>>> 5/2
2.5
>>> 2/3
0.6666666666666666
>>> 2.5 * 10**50 # Les puissances de 10 sont notées avec e+
2.5e+50
```

SHELL

- ▶ On utilise un point au lieu d'une virgule
- ▶ Les flottants n'ont qu'un nombre limité de chiffres après la virgule.
- ▶ On n'aura qu'une approximation de π et $\sqrt{2}$. Ce qui largement suffisant pour des calculs pratiques.

```
>>> import math
>>> math.pi
3.141592653589793
>>>
```

SHELL

- ▶ On code les décimaux avec la notation à virgule flottante :
 - ▶ Il est impossible de représenter de façon exacte tous les nombres réels.
 - ▶ On parle de flottants (pour nombre à virgule flottante). En anglais : *float*.

```
>>> 5/2
2.5
>>> 2/3
0.6666666666666666
>>> 2.5 * 10**50 # Les puissances de 10 sont notées avec e+
2.5e+50
```

SHELL

- ▶ On utilise un point au lieu d'une virgule
- ▶ Les flottants n'ont qu'un nombre limité de chiffres après la virgule.
- ▶ On n'aura qu'une approximation de π et $\sqrt{2}$. Ce qui largement suffisant pour des calculs pratiques.

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sqrt(2)
```

SHELL

- ▶ On code les décimaux avec la notation à virgule flottante :
 - ▶ Il est impossible de représenter de façon exacte tous les nombres réels.
 - ▶ On parle de flottants (pour nombre à virgule flottante). En anglais : *float*.

```
>>> 5/2
2.5
>>> 2/3
0.6666666666666666
>>> 2.5 * 10**50 # Les puissances de 10 sont notées avec e+
2.5e+50
```

SHELL

- ▶ On utilise un point au lieu d'une virgule
- ▶ Les flottants n'ont qu'un nombre limité de chiffres après la virgule.
- ▶ On n'aura qu'une approximation de π et $\sqrt{2}$. Ce qui largement suffisant pour des calculs pratiques.

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sqrt(2)
1.4142135623730951
```

SHELL

- ▶ Le calcul sur les nombres flottants est inexact, on évitera de tester l'égalité de tels nombres.

```
>>>
```

```
SHELL
```


- ▶ Le calcul sur les nombres flottants est inexact, on évitera de tester l'égalité de tels nombres.

```
>>> 0.1 + 0.1 + 0.1 + 0.1 == 0.4
```

SHELL

- ▶ Le calcul sur les nombres flottants est inexact, on évitera de tester l'égalité de tels nombres.

```
>>> 0.1 + 0.1 + 0.1 + 0.1 == 0.4  
True  
>>>
```

SHELL

- ▶ Le calcul sur les nombres flottants est inexact, on évitera de tester l'égalité de tels nombres.

```
>>> 0.1 + 0.1 + 0.1 + 0.1 == 0.4
True
>>> 0.1 + 0.1 + 0.1 == 0.3
```

SHELL

- ▶ Le calcul sur les nombres flottants est inexact, on évitera de tester l'égalité de tels nombres.

```
>>> 0.1 + 0.1 + 0.1 + 0.1 == 0.4  
True  
>>> 0.1 + 0.1 + 0.1 == 0.3  
False
```

SHELL

- ▶ Le calcul sur les nombres flottants est inexact, on évitera de tester l'égalité de tels nombres.

```
>>> 0.1 + 0.1 + 0.1 + 0.1 == 0.4  
True  
>>> 0.1 + 0.1 + 0.1 == 0.3  
False
```

SHELL

- ▶ On peut prédire le nombre de décimales exactes, mais c'est difficile (voire très difficile) en général.

- ▶ Le calcul sur les nombres flottants est inexact, on évitera de tester l'égalité de tels nombres.

```
>>> 0.1 + 0.1 + 0.1 + 0.1 == 0.4
True
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

SHELL

- ▶ On peut prédire le nombre de décimales exactes, mais c'est difficile (voire très difficile) en général.
- ▶ Que faire ? Remplacer l'égalité par une précision h

- ▶ Le calcul sur les nombres flottants est inexact, on évitera de tester l'égalité de tels nombres.

```
>>> 0.1 + 0.1 + 0.1 + 0.1 == 0.4
True
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

SHELL

- ▶ On peut prédire le nombre de décimales exactes, mais c'est difficile (voire très difficile) en général.
- ▶ Que faire ? Remplacer l'égalité par une précision h

```
# a et b flottant
a==b # Déconseillé car dangereux

h=.0001
abs(a-b) < h # Oui
abs(a) < h # signifie a est presque nul
```

SCRIPT

- ▶ Quels sont les nombres avec une quantité de chiffres finie après la virgule?
 - ▶ Nombre décimal : nombre ayant un nombre fini de chiffre en base 10.
 - ▶ Nombre dyadique : nombre ayant un nombre fini de chiffre en base 2.

- ▶ Quels sont les nombres avec une quantité de chiffres finie après la virgule?
 - ▶ Nombre décimal : nombre ayant un nombre fini de chiffre en base 10.
 - ▶ Nombre dyadique : nombre ayant un nombre fini de chiffre en base 2.
- ▶ Les flottants sont des nombres dyadiques (un nombre rond pour nous n'est pas forcément rond pour la machine).

$$0,1 = 0,000110011001100110011\dots_2$$

- ▶ Quels sont les nombres avec une quantité de chiffres finie après la virgule?
 - ▶ Nombre décimal : nombre ayant un nombre fini de chiffre en base 10.
 - ▶ Nombre dyadique : nombre ayant un nombre fini de chiffre en base 2.
- ▶ Les flottants sont des nombres dyadiques (un nombre rond pour nous n'est pas forcément rond pour la machine).

$$0,1 = 0,000110011001100110011\dots_2$$

- ▶ Python n'affiche qu'une valeur approchée de la valeur interne.

```
>>>
```

SHELL

- ▶ Quels sont les nombres avec une quantité de chiffres finie après la virgule?
 - ▶ Nombre décimal : nombre ayant un nombre fini de chiffre en base 10.
 - ▶ Nombre dyadique : nombre ayant un nombre fini de chiffre en base 2.
- ▶ Les flottants sont des nombres dyadiques (un nombre rond pour nous n'est pas forcément rond pour la machine).

$$0,1 = 0,000110011001100110011\dots_2$$

- ▶ Python n'affiche qu'une valeur approchée de la valeur interne.

```
>>> 0.1
```

SHELL

- ▶ Quels sont les nombres avec une quantité de chiffres finie après la virgule?
 - ▶ Nombre décimal : nombre ayant un nombre fini de chiffre en base 10.
 - ▶ Nombre dyadique : nombre ayant un nombre fini de chiffre en base 2.
- ▶ Les flottants sont des nombres dyadiques (un nombre rond pour nous n'est pas forcément rond pour la machine).

$$0,1 = 0,000110011001100110011\dots_2$$

- ▶ Python n'affiche qu'une valeur approchée de la valeur interne.

```
>>> 0.1  
0.1  
>>>
```

SHELL

- ▶ Quels sont les nombres avec une quantité de chiffres finie après la virgule?
 - ▶ Nombre décimal : nombre ayant un nombre fini de chiffre en base 10.
 - ▶ Nombre dyadique : nombre ayant un nombre fini de chiffre en base 2.
- ▶ Les flottants sont des nombres dyadiques (un nombre rond pour nous n'est pas forcément rond pour la machine).

$$0,1 = 0,000110011001100110011\dots_2$$

- ▶ Python n'affiche qu'une valeur approchée de la valeur interne.

```
>>> 0.1
0.1
>>> from decimal import Decimal
```

SHELL

- ▶ Quels sont les nombres avec une quantité de chiffres finie après la virgule?
 - ▶ Nombre décimal : nombre ayant un nombre fini de chiffre en base 10.
 - ▶ Nombre dyadique : nombre ayant un nombre fini de chiffre en base 2.
- ▶ Les flottants sont des nombres dyadiques (un nombre rond pour nous n'est pas forcément rond pour la machine).

$$0,1 = 0,000110011001100110011\dots_2$$

- ▶ Python n'affiche qu'une valeur approchée de la valeur interne.

```
>>> 0.1
0.1
>>> from decimal import Decimal
>>>
```

SHELL

- ▶ Quels sont les nombres avec une quantité de chiffres finie après la virgule?
 - ▶ Nombre décimal : nombre ayant un nombre fini de chiffre en base 10.
 - ▶ Nombre dyadique : nombre ayant un nombre fini de chiffre en base 2.
- ▶ Les flottants sont des nombres dyadiques (un nombre rond pour nous n'est pas forcément rond pour la machine).

$$0,1 = 0,000110011001100110011\dots_2$$

- ▶ Python n'affiche qu'une valeur approchée de la valeur interne.

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

SHELL

- ▶ Quels sont les nombres avec une quantité de chiffres finie après la virgule?
 - ▶ Nombre décimal : nombre ayant un nombre fini de chiffre en base 10.
 - ▶ Nombre dyadique : nombre ayant un nombre fini de chiffre en base 2.
- ▶ Les flottants sont des nombres dyadiques (un nombre rond pour nous n'est pas forcément rond pour la machine).

$$0,1 = 0,000110011001100110011\dots_2$$

- ▶ Python n'affiche qu'une valeur approchée de la valeur interne.

```
>>> 0.1
0.1
>>> from decimal import Decimal
>>> Decimal(0.1)
Decimal('0.10000000000000000055511151231257827021181583404541015625')
```

SHELL

- 🍃 Partie I. Remarques générales
- 🍃 Partie II. Calculs répétitifs
- 🍃 Partie III. Boucles while
- 🍃 Partie IV. Les nombres premiers
- 🍃 Partie V. Les nombres flottants
- 🍃 **Partie VI. Exemples**
- 🍃 Partie VII. Pour aller plus loin
- 🍃 Partie VIII. Table des matières

- ▶ On veut calculer la racine carrée approchée d'un nombre réel quelconque, par exemple calculer $\sqrt{2} = 1,4142\dots$ sans utiliser `math.sqrt`.

- ▶ On veut calculer la racine carrée approchée d'un nombre réel quelconque, par exemple calculer $\sqrt{2} = 1,4142\dots$ sans utiliser `math.sqrt`.
- ▶ Méthode de Newton (1669) :

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

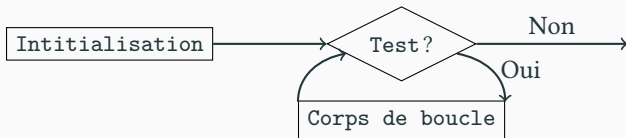
Si a est une approximation de \sqrt{r} alors b est une meilleure approximation. (cf explications en TP).

- ▶ On veut calculer la racine carrée approchée d'un nombre réel quelconque, par exemple calculer $\sqrt{2} = 1,4142\dots$ sans utiliser `math.sqrt`.
- ▶ Méthode de Newton (1669) :

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

Si a est une approximation de \sqrt{r} alors b est une meilleure approximation. (cf explications en TP).

- ▶ **Trois questions pour coder l'algorithme :**



- ▶ Comment améliorer l'approximation courante ? (corps de boucle)
- ▶ Mon approximation est-elle suffisante ? (test)
- ▶ Quelle première approximation choisir ? (initialisation)

- ▶ Comment améliorer l'approximation courante? (corps de boucle)

- ▶ Comment améliorer l'approximation courante? (corps de boucle)
 - ▶ Il suffit d'utiliser la formule de Newton (en remplaçant b par a)

$$a = 0.5 * (a + r/a)$$

SCRIPT

- ▶ Comment améliorer l'approximation courante? (corps de boucle)
 - ▶ Il suffit d'utiliser la formule de Newton (en remplaçant b par a)

$$a = 0.5 * (a + r/a)$$

SCRIPT

- ▶ Mon approximation est-elle suffisante? (test)

► Comment améliorer l'approximation courante? (corps de boucle)

- Il suffit d'utiliser la formule de Newton (en remplaçant b par a)

$$a = 0.5 * (a + r/a)$$

SCRIPT

► Mon approximation est-elle suffisante? (test)

- On s'arrête lorsque l'écart avec \sqrt{r} est suffisamment petit (on prend h petit)

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

► Comment améliorer l'approximation courante? (corps de boucle)

- Il suffit d'utiliser la formule de Newton (en remplaçant b par a)

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

```
SCRIPT
```

► Mon approximation est-elle suffisante? (test)

- On s'arrête lorsque l'écart avec \sqrt{r} est suffisamment petit (on prend h petit)

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

- Cela veut dire que l'on doit continuer tant que :

```
abs(a*a - r) >= h
```

```
SCRIPT
```

► Comment améliorer l'approximation courante? (corps de boucle)

- Il suffit d'utiliser la formule de Newton (en remplaçant b par a)

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

SCRIPT

► Mon approximation est-elle suffisante? (test)

- On s'arrête lorsque l'écart avec \sqrt{r} est suffisamment petit (on prend h petit)

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

- Cela veut dire que l'on doit continuer tant que :

```
abs(a*a - r) >= h
```

SCRIPT

► Quelle première approximation choisir? (initialisation)

► Comment améliorer l'approximation courante? (corps de boucle)

- Il suffit d'utiliser la formule de Newton (en remplaçant b par a)

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

SCRIPT

► Mon approximation est-elle suffisante? (test)

- On s'arrête lorsque l'écart avec \sqrt{r} est suffisamment petit (on prend h petit)

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

- Cela veut dire que l'on doit continuer tant que :

```
abs(a*a - r) >= h
```

SCRIPT

► Quelle première approximation choisir? (initialisation)

- On peut prouver que n'importe quelle valeur **strictement positive** convient.

```
a=1
```

SCRIPT

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

SCRIPT

- ▶ Testons notre programme.

>>>

SHELL

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

SCRIPT

- ▶ Testons notre programme.

```
>>> h=10**-10 # h = 0.00000 00000 1
```

SHELL

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

SCRIPT

- ▶ Testons notre programme.

```
>>> h=10**-10 # h = 0.00000 00000 1
>>>
```

SHELL

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

SCRIPT

- ▶ Testons notre programme.

```
>>> h=10**-10 # h = 0.00000 00000 1
>>> r2 = racine(2,h)
```

SHELL

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

SCRIPT

- ▶ Testons notre programme. (Quatre itérations seulement ont suffi)

```
>>> h=10**-10 # h = 0.00000 00000 1
>>> r2 = racine(2,h)
a = 1
a = 1.5
a = 1.4166666666666665
a = 1.4142156862745097
>>>
```

SHELL


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

SCRIPT

- ▶ Testons notre programme. (Quatre itérations seulement ont suffi)

```
>>> h=10**-10 # h = 0.00000 00000 1
>>> r2 = racine(2,h)
a = 1
a = 1.5
a = 1.4166666666666665
a = 1.4142156862745097
>>> from math import sqrt
```

SHELL

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

SCRIPT

- ▶ Testons notre programme. (Quatre itérations seulement ont suffi)

```
>>> h=10**-10 # h = 0.00000 00000 1
>>> r2 = racine(2,h)
a = 1
a = 1.5
a = 1.4166666666666665
a = 1.4142156862745097
>>> from math import sqrt
>>>
```

SHELL

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

SCRIPT

- ▶ Testons notre programme. (Quatre itérations seulement ont suffi)

```
>>> h=10**-10 # h = 0.00000 00000 1
>>> r2 = racine(2,h)
a = 1
a = 1.5
a = 1.4166666666666665
a = 1.4142156862745097
>>> from math import sqrt
>>> sqrt(2) # La version Python plus précise
```

SHELL

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

SCRIPT

- ▶ Testons notre programme. (Quatre itérations seulement ont suffi)

```
>>> h=10**-10 # h = 0.00000 00000 1
>>> r2 = racine(2,h)
a = 1
a = 1.5
a = 1.4166666666666665
a = 1.4142156862745097
>>> from math import sqrt
>>> sqrt(2) # La version Python plus précise
1.4142135623730951
>>>
```

SHELL

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

SCRIPT

- ▶ Testons notre programme. (Quatre itérations seulement ont suffi)

```
>>> h=10**-10 # h = 0.00000 00000 1
>>> r2 = racine(2,h)
a = 1
a = 1.5
a = 1.4166666666666665
a = 1.4142156862745097
>>> from math import sqrt
>>> sqrt(2) # La version Python plus précise
1.4142135623730951
>>> r2
```

SHELL

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

SCRIPT

- ▶ Testons notre programme. (Quatre itérations seulement ont suffi)

```
>>> h=10**-10 # h = 0.00000 00000 1
>>> r2 = racine(2,h)
a = 1
a = 1.5
a = 1.4166666666666665
a = 1.4142156862745097
>>> from math import sqrt
>>> sqrt(2) # La version Python plus précise
1.4142135623730951
>>> r2
1.4142135623746899
>>>
```

SHELL

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

SCRIPT

- ▶ Testons notre programme. (Quatre itérations seulement ont suffi)

```
>>> h=10**-10 # h = 0.00000 00000 1
>>> r2 = racine(2,h)
a = 1
a = 1.5
a = 1.4166666666666665
a = 1.4142156862745097
>>> from math import sqrt
>>> sqrt(2) # La version Python plus précise
1.4142135623730951
>>> r2
1.4142135623746899
>>> abs(sqrt(2)-r2)
```

SHELL

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

SCRIPT

- ▶ Testons notre programme. (Quatre itérations seulement ont suffi)

```
>>> h=10**-10 # h = 0.00000 00000 1
>>> r2 = racine(2,h)
a = 1
a = 1.5
a = 1.4166666666666665
a = 1.4142156862745097
>>> from math import sqrt
>>> sqrt(2) # La version Python plus précise
1.4142135623730951
>>> r2
1.4142135623746899
>>> abs(sqrt(2)-r2)
1.5947243525715749e-12
```

SHELL

- ▶ On s'intéresse au PGCD (Plus Grand Commun Diviseur) de deux entiers naturels.

- ▶ On s'intéresse au PGCD (Plus Grand Commun Diviseur) de deux entiers naturels.

$$\text{PGCD}(4,6) = 2 \quad \text{car} \quad 4 = 2 \times 2 \quad \text{et} \quad 6 = 2 \times 3$$

- ▶ On s'intéresse au PGCD (Plus Grand Commun Diviseur) de deux entiers naturels.

$$\text{PGCD}(4,6) = 2 \quad \text{car} \quad 4 = 2 \times 2 \quad \text{et} \quad 6 = 2 \times 3$$

$$\text{PGCD}(10,15) = 5 \quad \text{car} \quad 10 = 5 \times 2 \quad \text{et} \quad 15 = 5 \times 3$$

- ▶ On s'intéresse au PGCD (Plus Grand Commun Diviseur) de deux entiers naturels.

$$\text{PGCD}(4,6) = 2 \quad \text{car} \quad 4 = 2 \times 2 \quad \text{et} \quad 6 = 2 \times 3$$

$$\text{PGCD}(10,15) = 5 \quad \text{car} \quad 10 = 5 \times 2 \quad \text{et} \quad 15 = 5 \times 3$$

$$\text{PGCD}(0,8) = 8 \quad \text{car} \quad 0 = 8 \times 0 \quad \text{et} \quad 8 = 8 \times 1$$

- ▶ On s'intéresse au PGCD (Plus Grand Commun Diviseur) de deux entiers naturels.

$$\text{PGCD}(4,6) = 2 \quad \text{car} \quad 4 = 2 \times 2 \quad \text{et} \quad 6 = 2 \times 3$$

$$\text{PGCD}(10,15) = 5 \quad \text{car} \quad 10 = 5 \times 2 \quad \text{et} \quad 15 = 5 \times 3$$

$$\text{PGCD}(0,8) = 8 \quad \text{car} \quad 0 = 8 \times 0 \quad \text{et} \quad 8 = 8 \times 1$$

- ▶ L'algorithme d'Euclide calcule le PGCD de deux entiers naturels a et b .
 - ▶ $(d \text{ divise à la fois } a \text{ et } b) \Leftrightarrow (d \text{ divise à la fois } b \text{ et } a\%b)$
 - ▶ En particulier, $\text{PGCD}(a, b) = \text{PGCD}(b, a\%b)$

- ▶ On s'intéresse au PGCD (Plus Grand Commun Diviseur) de deux entiers naturels.

$$\text{PGCD}(4,6) = 2 \quad \text{car} \quad 4 = 2 \times 2 \quad \text{et} \quad 6 = 2 \times 3$$

$$\text{PGCD}(10,15) = 5 \quad \text{car} \quad 10 = 5 \times 2 \quad \text{et} \quad 15 = 5 \times 3$$

$$\text{PGCD}(0,8) = 8 \quad \text{car} \quad 0 = 8 \times 0 \quad \text{et} \quad 8 = 8 \times 1$$

- ▶ L'algorithme d'Euclide calcule le PGCD de deux entiers naturels a et b .
 - ▶ (d divise à la fois a et b) \Leftrightarrow (d divise à la fois b et $a \% b$)
 - ▶ En particulier, $\text{PGCD}(a, b) = \text{PGCD}(b, a \% b)$

- ▶ Calculons le PGCD de 135 et 72.

```
>>>
```

```
SHELL
```

```
PGCD(135,72) =
```

```
=
```

```
=
```

```
=
```

- ▶ On s'intéresse au PGCD (Plus Grand Commun Diviseur) de deux entiers naturels.

$$\text{PGCD}(4,6) = 2 \quad \text{car} \quad 4 = 2 \times 2 \quad \text{et} \quad 6 = 2 \times 3$$

$$\text{PGCD}(10,15) = 5 \quad \text{car} \quad 10 = 5 \times 2 \quad \text{et} \quad 15 = 5 \times 3$$

$$\text{PGCD}(0,8) = 8 \quad \text{car} \quad 0 = 8 \times 0 \quad \text{et} \quad 8 = 8 \times 1$$

- ▶ L'algorithme d'Euclide calcule le PGCD de deux entiers naturels a et b .
 - ▶ (d divise à la fois a et b) \Leftrightarrow (d divise à la fois b et $a \% b$)
 - ▶ En particulier, $\text{PGCD}(a, b) = \text{PGCD}(b, a \% b)$

- ▶ Calculons le PGCD de 135 et 72.

```
>>> 135%72
```

```
SHELL
```

```
PGCD(135,72) =
```

```
=
```

```
=
```

```
=
```

- ▶ On s'intéresse au PGCD (Plus Grand Commun Diviseur) de deux entiers naturels.

$$\text{PGCD}(4,6) = 2 \quad \text{car} \quad 4 = 2 \times 2 \quad \text{et} \quad 6 = 2 \times 3$$

$$\text{PGCD}(10,15) = 5 \quad \text{car} \quad 10 = 5 \times 2 \quad \text{et} \quad 15 = 5 \times 3$$

$$\text{PGCD}(0,8) = 8 \quad \text{car} \quad 0 = 8 \times 0 \quad \text{et} \quad 8 = 8 \times 1$$

- ▶ L'algorithme d'Euclide calcule le PGCD de deux entiers naturels a et b .
 - ▶ (d divise à la fois a et b) \Leftrightarrow (d divise à la fois b et $a \% b$)
 - ▶ En particulier, $\text{PGCD}(a, b) = \text{PGCD}(b, a \% b)$

- ▶ Calculons le PGCD de 135 et 72.

```
>>> 135%72
63
>>>
```

SHELL

$$\begin{aligned} \text{PGCD}(135,72) &= \text{PGCD}(72,63) \\ &= \\ &= \\ &= \end{aligned}$$

- ▶ On s'intéresse au PGCD (Plus Grand Commun Diviseur) de deux entiers naturels.

$$\text{PGCD}(4,6) = 2 \quad \text{car} \quad 4 = 2 \times 2 \quad \text{et} \quad 6 = 2 \times 3$$

$$\text{PGCD}(10,15) = 5 \quad \text{car} \quad 10 = 5 \times 2 \quad \text{et} \quad 15 = 5 \times 3$$

$$\text{PGCD}(0,8) = 8 \quad \text{car} \quad 0 = 8 \times 0 \quad \text{et} \quad 8 = 8 \times 1$$

- ▶ L'algorithme d'Euclide calcule le PGCD de deux entiers naturels a et b .
 - ▶ (d divise à la fois a et b) \Leftrightarrow (d divise à la fois b et $a \% b$)
 - ▶ En particulier, $\text{PGCD}(a, b) = \text{PGCD}(b, a \% b)$

- ▶ Calculons le PGCD de 135 et 72.

```
>>> 135%72
63
>>> 72%63
```

SHELL

$$\begin{aligned} \text{PGCD}(135,72) &= \text{PGCD}(72,63) \\ &= \\ &= \\ &= \end{aligned}$$

- ▶ On s'intéresse au PGCD (Plus Grand Commun Diviseur) de deux entiers naturels.

$$\text{PGCD}(4,6) = 2 \quad \text{car} \quad 4 = 2 \times 2 \quad \text{et} \quad 6 = 2 \times 3$$

$$\text{PGCD}(10,15) = 5 \quad \text{car} \quad 10 = 5 \times 2 \quad \text{et} \quad 15 = 5 \times 3$$

$$\text{PGCD}(0,8) = 8 \quad \text{car} \quad 0 = 8 \times 0 \quad \text{et} \quad 8 = 8 \times 1$$

- ▶ L'algorithme d'Euclide calcule le PGCD de deux entiers naturels a et b .
 - ▶ (d divise à la fois a et b) \Leftrightarrow (d divise à la fois b et $a \% b$)
 - ▶ En particulier, $\text{PGCD}(a, b) = \text{PGCD}(b, a \% b)$

- ▶ Calculons le PGCD de 135 et 72.

```
>>> 135%72
63
>>> 72%63
9
>>>
```

SHELL

$$\begin{aligned} \text{PGCD}(135,72) &= \text{PGCD}(72,63) \\ &= \text{PGCD}(63,9) \\ &= \\ &= \end{aligned}$$

- ▶ On s'intéresse au PGCD (Plus Grand Commun Diviseur) de deux entiers naturels.

$$\text{PGCD}(4,6) = 2 \quad \text{car} \quad 4 = 2 \times 2 \quad \text{et} \quad 6 = 2 \times 3$$

$$\text{PGCD}(10,15) = 5 \quad \text{car} \quad 10 = 5 \times 2 \quad \text{et} \quad 15 = 5 \times 3$$

$$\text{PGCD}(0,8) = 8 \quad \text{car} \quad 0 = 8 \times 0 \quad \text{et} \quad 8 = 8 \times 1$$

- ▶ L'algorithme d'Euclide calcule le PGCD de deux entiers naturels a et b .
 - ▶ (d divise à la fois a et b) \Leftrightarrow (d divise à la fois b et $a \% b$)
 - ▶ En particulier, $\text{PGCD}(a, b) = \text{PGCD}(b, a \% b)$

- ▶ Calculons le PGCD de 135 et 72.

```
>>> 135%72
```

```
63
```

```
>>> 72%63
```

```
9
```

```
>>> 63%9
```

SHELL

$$\text{PGCD}(135,72) = \text{PGCD}(72,63)$$

$$= \text{PGCD}(63,9)$$

$$=$$

$$=$$

- ▶ On s'intéresse au PGCD (Plus Grand Commun Diviseur) de deux entiers naturels.

$$\text{PGCD}(4,6) = 2 \quad \text{car} \quad 4 = 2 \times 2 \quad \text{et} \quad 6 = 2 \times 3$$

$$\text{PGCD}(10,15) = 5 \quad \text{car} \quad 10 = 5 \times 2 \quad \text{et} \quad 15 = 5 \times 3$$

$$\text{PGCD}(0,8) = 8 \quad \text{car} \quad 0 = 8 \times 0 \quad \text{et} \quad 8 = 8 \times 1$$

- ▶ L'algorithme d'Euclide calcule le PGCD de deux entiers naturels a et b .
 - ▶ (d divise à la fois a et b) \Leftrightarrow (d divise à la fois b et $a \% b$)
 - ▶ En particulier, $\text{PGCD}(a, b) = \text{PGCD}(b, a \% b)$

- ▶ Calculons le PGCD de 135 et 72.

```
>>> 135%72
63
>>> 72%63
9
>>> 63%9
0
```

SHELL

$$\begin{aligned} \text{PGCD}(135,72) &= \text{PGCD}(72,63) \\ &= \text{PGCD}(63,9) \\ &= \text{PGCD}(9,0) \\ &= \end{aligned}$$

- ▶ On s'intéresse au PGCD (Plus Grand Commun Diviseur) de deux entiers naturels.

$$\text{PGCD}(4,6) = 2 \quad \text{car} \quad 4 = 2 \times 2 \quad \text{et} \quad 6 = 2 \times 3$$

$$\text{PGCD}(10,15) = 5 \quad \text{car} \quad 10 = 5 \times 2 \quad \text{et} \quad 15 = 5 \times 3$$

$$\text{PGCD}(0,8) = 8 \quad \text{car} \quad 0 = 8 \times 0 \quad \text{et} \quad 8 = 8 \times 1$$

- ▶ L'algorithme d'Euclide calcule le PGCD de deux entiers naturels a et b .
 - ▶ (d divise à la fois a et b) \Leftrightarrow (d divise à la fois b et $a \% b$)
 - ▶ En particulier, $\text{PGCD}(a, b) = \text{PGCD}(b, a \% b)$

- ▶ Calculons le PGCD de 135 et 72.

```
>>> 135%72
```

```
63
```

```
>>> 72%63
```

```
9
```

```
>>> 63%9
```

```
0
```

SHELL

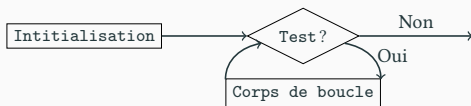
$$\text{PGCD}(135,72) = \text{PGCD}(72,63)$$

$$= \text{PGCD}(63,9)$$

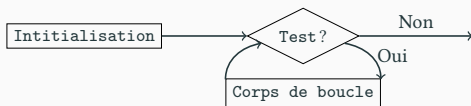
$$= \text{PGCD}(9,0)$$

$$= 9$$

$$\text{PGCD}(135,72) = \text{PGCD}(72,63) = \text{PGCD}(63,9) = \text{PGCD}(9,0) = 9$$



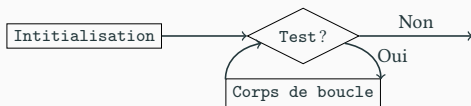
$$\text{PGCD}(135,72) = \text{PGCD}(72,63) = \text{PGCD}(63,9) = \text{PGCD}(9,0) = 9$$



De quelles variables a-t-on besoin ?

a	135	72	63	9
b	72	63	9	0

$$\text{PGCD}(135,72) = \text{PGCD}(72,63) = \text{PGCD}(63,9) = \text{PGCD}(9,0) = 9$$

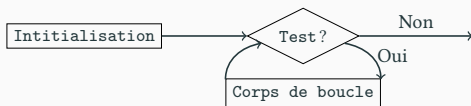


De quelles variables a-t-on besoin ?

a	135	72	63	9
b	72	63	9	0

- Initialisation : a et b sont des données du problèmes

$$\text{PGCD}(135,72) = \text{PGCD}(72,63) = \text{PGCD}(63,9) = \text{PGCD}(9,0) = 9$$



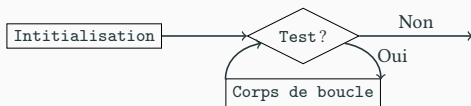
De quelles variables a-t-on besoin ?

a	135	72	63	9
b	72	63	9	0

- ▶ Initialisation : a et b sont des données du problèmes
- ▶ Corps de boucle :

a = b # <i>Faux !</i>	SCRIPT
b = a%b # <i>Pourquoi ?</i>	

$$\text{PGCD}(135,72) = \text{PGCD}(72,63) = \text{PGCD}(63,9) = \text{PGCD}(9,0) = 9$$



De quelles variables a-t-on besoin ?

a	135	72	63	9
b	72	63	9	0

- ▶ Initialisation : a et b sont des données du problèmes
- ▶ Corps de boucle :

```

a = b # Faux !
b = a%b # Pourquoi ?
    
```

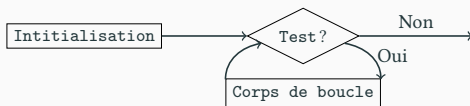
SCRIPT

```

temp=a%b
a = b
b = temp
    
```

SCRIPT

$$\text{PGCD}(135,72) = \text{PGCD}(72,63) = \text{PGCD}(63,9) = \text{PGCD}(9,0) = 9$$



De quelles variables a-t-on besoin ?

a	135	72	63	9
b	72	63	9	0

- ▶ Initialisation : a et b sont des données du problèmes
- ▶ Corps de boucle :

```

a = b    # Faux !
b = a%b  # Pourquoi ?
    
```

SCRIPT

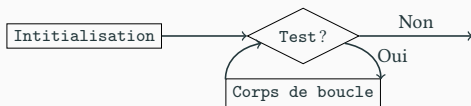
```

temp=a%b
a = b
b = temp
    
```

SCRIPT

- ▶ Test : on continue tant que $b \neq 0$

$$\text{PGCD}(135,72) = \text{PGCD}(72,63) = \text{PGCD}(63,9) = \text{PGCD}(9,0) = 9$$



De quelles variables a-t-on besoin ?

a	135	72	63	9
b	72	63	9	0

- ▶ Initialisation : a et b sont des données du problèmes
- ▶ Corps de boucle :

```

a = b    # Faux !
b = a%b  # Pourquoi ?
    
```

SCRIPT

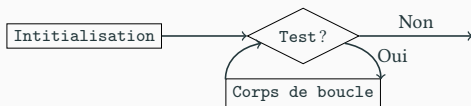
```

temp=a%b
a = b
b = temp
    
```

SCRIPT

- ▶ Test : on continue tant que $b \neq 0$
 - ▶ b est un entier naturel qui décroît strictement

$$\text{PGCD}(135,72) = \text{PGCD}(72,63) = \text{PGCD}(63,9) = \text{PGCD}(9,0) = 9$$



De quelles variables a-t-on besoin ?

a	135	72	63	9
b	72	63	9	0

- ▶ Initialisation : a et b sont des données du problèmes
- ▶ Corps de boucle :

```

a = b   # Faux !
b = a%b # Pourquoi ?
    
```

SCRIPT

```

temp=a%b
a = b
b = temp
    
```

SCRIPT

- ▶ Test : on continue tant que $b \neq 0$
 - ▶ b est un entier naturel qui décroît strictement
 - ▶ b finira toujours par être égale à 0 : **le programme s'arrêtera!**

```
def PGCD(a,b): # on suppose a,b entiers >= 0
    while b != 0:
        temp = a%b
        a = b
        b = temp
        print('a =',a, 'b =', b) # Pour voir !
    return a
```

SCRIPT

>>>

SHELL

```
def PGCD(a,b): # on suppose a,b entiers >= 0
    while b != 0:
        temp = a%b
        a = b
        b = temp
        print('a =',a, 'b =', b) # Pour voir !
    return a
```

SCRIPT

```
>>> PGCD(38160,33000)
```

SHELL

```
def PGCD(a,b): # on suppose a,b entiers >= 0
    while b != 0:
        temp = a%b
        a = b
        b = temp
        print('a =',a, 'b =', b) # Pour voir !
    return a
```

SCRIPT

```
>>> PGCD(38160,33000)
a = 33000 b = 5160
a = 5160 b = 2040
a = 2040 b = 1080
a = 1080 b = 960
a = 960 b = 120
a = 120 b = 0
120
```

SHELL

► Problème récréatif de Leonardo Fibonacci (1202)

*Un homme met un couple de lapins dans un lieu isolé de tous les côtés par un mur.
Combien de couples obtient-on en un an si chaque couple engendre tous les mois un
nouveau couple à compter du troisième mois de son existence ?*

- ▶ Problème récréatif de Leonardo Fibonacci (1202)

Un homme met un couple de lapins dans un lieu isolé de tous les côtés par un mur. Combien de couples obtient-on en un an si chaque couple engendre tous les mois un nouveau couple à compter du troisième mois de son existence ?

- ▶ On note $F(n)$ le nombre de lapins au début du mois n .
- ▶ Pour tout $n > 1$, on a $F(n+1) = F(n) + F(n-1)$

Nouvelle génération = anciens adultes + nouveaux adultes

- ▶ Problème récréatif de Leonardo Fibonacci (1202)

Un homme met un couple de lapins dans un lieu isolé de tous les côtés par un mur. Combien de couples obtient-on en un an si chaque couple engendre tous les mois un nouveau couple à compter du troisième mois de son existence ?

- ▶ On note $F(n)$ le nombre de lapins au début du mois n .
- ▶ Pour tout $n > 1$, on a $F(n+1) = F(n) + F(n-1)$

Nouvelle génération = anciens adultes + nouveaux adultes

- ▶ On voit de manière assez évidente que :

$$F(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

- ▶ Problème récréatif de Leonardo Fibonacci (1202)

Un homme met un couple de lapins dans un lieu isolé de tous les côtés par un mur. Combien de couples obtient-on en un an si chaque couple engendre tous les mois un nouveau couple à compter du troisième mois de son existence ?

- ▶ On note $F(n)$ le nombre de lapins au début du mois n .
- ▶ Pour tout $n > 1$, on a $F(n+1) = F(n) + F(n-1)$

Nouvelle génération = anciens adultes + nouveaux adultes

- ▶ On voit de manière assez évidente que :

$$F(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

- ▶ Mais on souhaite éviter les flottants.

```
def F(n): # Version récursive
    print('Je calcule F(n) pour n =',n) #Voyons...
    if n==1 or n==2:
        return 1
    else:
        return F(n-1) + F(n-2)
```

SCRIPT

>>>

SHELL

```
def F(n): # Version récursive
    print('Je calcule F(n) pour n =', n) #Voyons...
    if n==1 or n==2:
        return 1
    else:
        return F(n-1) + F(n-2)
```

SCRIPT

```
>>> F(6)
```

SHELL

SCRIPT

```
def F(n): # Version récursive
    print('Je calcule F(n) pour n =', n) #Voyons...
    if n==1 or n==2:
        return 1
    else:
        return F(n-1) + F(n-2)
```

SHELL

```
>>> F(6)
Je calcule F(n) pour n = 6
Je calcule F(n) pour n = 5
Je calcule F(n) pour n = 4
Je calcule F(n) pour n = 3
Je calcule F(n) pour n = 2
Je calcule F(n) pour n = 1
Je calcule F(n) pour n = 2
Je calcule F(n) pour n = 3
Je calcule F(n) pour n = 2
Je calcule F(n) pour n = 1
Je calcule F(n) pour n = 4
Je calcule F(n) pour n = 3
Je calcule F(n) pour n = 2
Je calcule F(n) pour n = 1
Je calcule F(n) pour n = 2
8
```

SCRIPT

```
def F(n): # Version récursive
    print('Je calcule F(n) pour n =', n) #Voyons...
    if n==1 or n==2:
        return 1
    else:
        return F(n-1) + F(n-2)
```

SHELL

```
>>> F(6)
Je calcule F(n) pour n = 6
Je calcule F(n) pour n = 5
Je calcule F(n) pour n = 4
Je calcule F(n) pour n = 3
Je calcule F(n) pour n = 2
Je calcule F(n) pour n = 1
Je calcule F(n) pour n = 2
Je calcule F(n) pour n = 3
Je calcule F(n) pour n = 2
Je calcule F(n) pour n = 1
Je calcule F(n) pour n = 4
Je calcule F(n) pour n = 3
Je calcule F(n) pour n = 2
Je calcule F(n) pour n = 1
Je calcule F(n) pour n = 2
8
```

Peu efficace

- ▶ On calcule de nombreuses fois la même chose.
- ▶ Solution possible : voir cours 7 (mémoïsation)

On souhaite calculer $F(N)$.

L'astuce est d'utiliser deux variables

a et b.

$F(n)$	a	1	2	3	5	8
$F(n-1)$	b	1	1	2	3	5
n	n	2	3	4	5	6

On souhaite calculer $F(N)$.

L'astuce est d'utiliser deux variables

a et b.

$F(n)$	a	1	2	3	5	8
$F(n-1)$	b	1	1	2	3	5
n	n	2	3	4	5	6

► Initialisation

```
a = 1  
b = 1  
n = 2
```

SCRIPT

On souhaite calculer $F(N)$.

L'astuce est d'utiliser deux variables

a et b .

$F(n)$	a	1	2	3	5	8
$F(n-1)$	b	1	1	2	3	5
n	n	2	3	4	5	6

► Initialisation

```
a = 1
b = 1
n = 2
```

SCRIPT

► Corps de boucle. On utilise un invariant de boucle $a = F(n)$ et $b = F(n-1)$

```
# a==F(n) et b==F(n-1)
temp = a+b           # temp == F(n+1)
b = a                # b == F(n)
a = temp             # a == F(n+1)
# a==F(n+1) et b==F(n)
n = n+1
# a==F(n) et b==F(n-1)
```

SCRIPT

On souhaite calculer $F(N)$.

L'astuce est d'utiliser deux variables

a et b.

$F(n)$	a	1	2	3	5	8
$F(n-1)$	b	1	1	2	3	5
n	n	2	3	4	5	6

► Initialisation

```
a = 1
b = 1
n = 2
```

SCRIPT

► Corps de boucle. On utilise un invariant de boucle $a = F(n)$ et $b = F(n-1)$

```
# a==F(n) et b==F(n-1)
temp = a+b          # temp == F(n+1)
b = a              # b == F(n)
a = temp           # a == F(n+1)
# a==F(n+1) et b==F(n)
n = n+1
# a==F(n) et b==F(n-1)
```

SCRIPT

► On continue tant que $n < N$.

- ▶ Il suffit d'écrire le code...

```
def F(N):  
    a = 1  
    b = 1  
    n = 2  
    while n < N:  
        temp = a+b  
        b = a  
        a = temp  
        n = n+1  
    return a
```

SCRIPT

- ▶ Il suffit d'écrire le code...

```
def F(N):  
    a = 1  
    b = 1  
    n = 2  
    while n < N:  
        temp = a+b  
        b = a  
        a = temp  
        n = n+1  
    return a
```

SCRIPT

- ▶ ...et apprécier la vitesse et l'efficacité de cette méthode!

```
>>>
```

SHELL

- ▶ Il suffit d'écrire le code...

```
def F(N):  
    a = 1  
    b = 1  
    n = 2  
    while n < N:  
        temp = a+b  
        b = a  
        a = temp  
        n = n+1  
    return a
```

SCRIPT

- ▶ ...et apprécier la vitesse et l'efficacité de cette méthode!

```
>>> F(8)
```

SHELL

- ▶ Il suffit d'écrire le code...

```
def F(N):  
    a = 1  
    b = 1  
    n = 2  
    while n < N:  
        temp = a+b  
        b = a  
        a = temp  
        n = n+1  
    return a
```

SCRIPT

- ▶ ...et apprécier la vitesse et l'efficacité de cette méthode!

```
>>> F(8)  
21  
>>>
```

SHELL

- ▶ Il suffit d'écrire le code...

```
def F(N):  
    a = 1  
    b = 1  
    n = 2  
    while n < N:  
        temp = a+b  
        b = a  
        a = temp  
        n = n+1  
    return a
```

SCRIPT

- ▶ ...et apprécier la vitesse et l'efficacité de cette méthode!

```
>>> F(8)  
21  
>>> F(378)
```

SHELL

- ▶ Il suffit d'écrire le code...

```
def F(N):  
    a = 1  
    b = 1  
    n = 2  
    while n < N:  
        temp = a+b  
        b = a  
        a = temp  
        n = n+1  
    return a
```

SCRIPT

- ▶ ...et apprécier la vitesse et l'efficacité de cette méthode !

```
>>> F(8)  
21  
>>> F(378)  
444470572323423749883397351998290851993343081863640916635139  
7897095281987215864
```

SHELL

On considère la suite u_n arithmético-géométrique définie par récurrence :

$$\begin{cases} u_0 & = 234 \\ u_{n+1} & = -\frac{8}{10}u_n + 3 \end{cases}$$

- ▶ Écrire une fonction avec une boucle while pour calculer u_n .

On considère la suite u_n arithmético-géométrique définie par récurrence :

$$\begin{cases} u_0 & = 234 \\ u_{n+1} & = -\frac{8}{10}u_n + 3 \end{cases}$$

- ▶ Écrire une fonction avec une boucle while pour calculer u_n .
- ▶ Sa limite est $\ell = \frac{5}{3}$. Faire une boucle qui s'arrête lorsque $|u_n - \ell| < 10^{-5}$

- 🌿 Partie I. Remarques générales
- 🌿 Partie II. Calculs répétitifs
- 🌿 Partie III. Boucles while
- 🌿 Partie IV. Les nombres premiers
- 🌿 Partie V. Les nombres flottants
- 🌿 Partie VI. Exemples
- 🌿 **Partie VII. Pour aller plus loin**
- 🌿 Partie VIII. Table des matières

- ▶ La commande `return` provoque la sortie immédiate d'une fonction
 - ▶ même à l'intérieur d'une boucle.

```
def div4(n):  
    i = n  
    while i < n+4:  
        if i%4 == 0:  
            return i  
        i = i+1  
    print('Inutile')
```

SCRIPT

```
>>>
```

SHELL

- ▶ La commande `return` provoque la sortie immédiate d'une fonction
 - ▶ même à l'intérieur d'une boucle.

```
def div4(n):  
    i = n  
    while i < n+4:  
        if i%4 == 0:  
            return i  
        i = i+1  
    print('Inutile')
```

SCRIPT

```
>>> div4(13)
```

SHELL

- ▶ La commande `return` provoque la sortie immédiate d'une fonction
 - ▶ même à l'intérieur d'une boucle.

```
def div4(n):  
    i = n  
    while i < n+4:  
        if i%4 == 0:  
            return i  
        i = i+1  
    print('Inutile')
```

SCRIPT

```
>>> div4(13)  
16  
>>>
```

SHELL

- ▶ La commande `return` provoque la sortie immédiate d'une fonction
 - ▶ même à l'intérieur d'une boucle.

```
def div4(n):  
    i = n  
    while i < n+4:  
        if i%4 == 0:  
            return i  
        i = i+1  
    print('Inutile')
```

SCRIPT

```
>>> div4(13)  
16  
>>> div4(144)
```

SHELL

- ▶ La commande `return` provoque la sortie immédiate d'une fonction
 - ▶ même à l'intérieur d'une boucle.

```
def div4(n):  
    i = n  
    while i < n+4:  
        if i%4 == 0:  
            return i  
        i = i+1  
    print('Inutile')
```

SCRIPT

```
>>> div4(13)  
16  
>>> div4(144)  
144
```

SHELL

- ▶ La commande `return` provoque la sortie immédiate d'une fonction
 - ▶ même à l'intérieur d'une boucle.

```
def div4(n):  
    i = n  
    while i < n+4:  
        if i%4 == 0:  
            return i  
        i = i+1  
    print('Inutile')
```

SCRIPT

```
>>> div4(13)  
16  
>>> div4(144)  
144
```

SHELL

- ▶ La commande `break` provoque la sortie de la boucle, l'exécution se poursuit.

```
def div5(n):  
    i = n  
    while i < n+5:  
        if i%5 == 0:  
            break  
        i = i+1  
    print('Sortie')  
    return i
```

SCRIPT

```
>>>
```

SHELL

- ▶ La commande `return` provoque la sortie immédiate d'une fonction
 - ▶ même à l'intérieur d'une boucle.

```
def div4(n):  
    i = n  
    while i < n+4:  
        if i%4 == 0:  
            return i  
        i = i+1  
    print('Inutile')
```

SCRIPT

```
>>> div4(13)  
16  
>>> div4(144)  
144
```

SHELL

- ▶ La commande `break` provoque la sortie de la boucle, l'exécution se poursuit.

```
def div5(n):  
    i = n  
    while i < n+5:  
        if i%5 == 0:  
            break  
        i = i+1  
    print('Sortie')  
    return i
```

SCRIPT

```
>>> div5(13)
```

SHELL

- ▶ La commande `return` provoque la sortie immédiate d'une fonction
 - ▶ même à l'intérieur d'une boucle.

```
def div4(n):  
    i = n  
    while i < n+4:  
        if i%4 == 0:  
            return i  
        i = i+1  
    print('Inutile')
```

SCRIPT

```
>>> div4(13)  
16  
>>> div4(144)  
144
```

SHELL

- ▶ La commande `break` provoque la sortie de la boucle, l'exécution se poursuit.

```
def div5(n):  
    i = n  
    while i < n+5:  
        if i%5 == 0:  
            break  
        i = i+1  
    print('Sortie')  
    return i
```

SCRIPT

```
>>> div5(13)  
Sortie  
15  
>>>
```

SHELL

- ▶ La commande `return` provoque la sortie immédiate d'une fonction
 - ▶ même à l'intérieur d'une boucle.

```
def div4(n):  
    i = n  
    while i < n+4:  
        if i%4 == 0:  
            return i  
        i = i+1  
    print('Inutile')
```

SCRIPT

```
>>> div4(13)  
16  
>>> div4(144)  
144
```

SHELL

- ▶ La commande `break` provoque la sortie de la boucle, l'exécution se poursuit.

```
def div5(n):  
    i = n  
    while i < n+5:  
        if i%5 == 0:  
            break  
        i = i+1  
    print('Sortie')  
    return i
```

SCRIPT

```
>>> div5(13)  
Sortie  
15  
>>> div5(145)
```

SHELL

- ▶ La commande `return` provoque la sortie immédiate d'une fonction
 - ▶ même à l'intérieur d'une boucle.

```
def div4(n):  
    i = n  
    while i < n+4:  
        if i%4 == 0:  
            return i  
        i = i+1  
    print('Inutile')
```

SCRIPT

```
>>> div4(13)  
16  
>>> div4(144)  
144
```

SHELL

- ▶ La commande `break` provoque la sortie de la boucle, l'exécution se poursuit.

```
def div5(n):  
    i = n  
    while i < n+5:  
        if i%5 == 0:  
            break  
        i = i+1  
    print('Sortie')  
    return i
```

SCRIPT

```
>>> div5(13)  
Sortie  
15  
>>> div5(145)  
Sortie  
145
```

SHELL



Déconseillé au moins de 18 ans.





Déconseillé au moins de 18 ans.



- En utilisant le `break`, on peut se passer de test dans la boucle `while` !

```
n = 40
while True:
    if n%7 == 0:
        break
    n = n+1
    print('n=',n)
```

SCRIPT



Déconseillé au moins de 18 ans.



- ▶ En utilisant le `break`, on peut se passer de test dans la boucle `while` !

```
n = 40
while True:
    if n%7 == 0:
        break
    n = n+1
    print('n=',n)
```

SCRIPT

- ▶ Pratique dangereuse, à ne pas utiliser sans bonnes raisons.

- ▶ Pour deux entiers naturels a et n on veut calculer a^n , sans utiliser `**`.

- ▶ Pour deux entiers naturels a et n on veut calculer a^n , sans utiliser `**`.
- ▶ Version récursive Idée : $a^n = a \times a^{n-1}$ et $a^0 = 1$

```
def puissance(a,n):  
    if n == 0:  
        return 1  
    else:  
        return a * puissance(a, n-1)
```

SCRIPT

- ▶ Pour deux entiers naturels a et n on veut calculer a^n , sans utiliser `**`.
- ▶ Version récursive Idée : $a^n = a \times a^{n-1}$ et $a^0 = 1$

```
def puissance(a,n):  
    if n == 0:  
        return 1  
    else:  
        return a * puissance(a, n-1)
```

SCRIPT

- ▶ Version itérative.

```
def puissance(a,n):  
    i = 0  
    p = 1 # p initialisé à a0  
    while i < n:  
        p = a*p  
        i = i+1 # p == ai  
    return p # i == n et p == ai donc p == an
```

SCRIPT

$$a^{21} = a^{2 \times 10 + 1} = a \times (a^{10})^2$$

$$\begin{aligned} a^{21} &= a^{2 \times 10 + 1} = a \times (a^{10})^2 \\ a^{10} &= a^{2 \times 5 + 0} = (a^5)^2 \end{aligned}$$

$$a^{21} = a^{2 \times 10 + 1} = a \times (a^{10})^2$$

$$a^{10} = a^{2 \times 5 + 0} = (a^5)^2$$

$$a^5 = a^{2 \times 2 + 1} = a \times (a^2)^2$$

$$a^{21} = a^{2 \times 10 + 1} = a \times (a^{10})^2$$

$$a^{10} = a^{2 \times 5 + 0} = (a^5)^2$$

$$a^5 = a^{2 \times 2 + 1} = a \times (a^2)^2$$

$$a^2 = a^{2 \times 1 + 0} = (a^1)^2$$

$$a^{21} = a^{2 \times 10 + 1} = a \times (a^{10})^2$$

$$a^{10} = a^{2 \times 5 + 0} = (a^5)^2$$

$$a^5 = a^{2 \times 2 + 1} = a \times (a^2)^2$$

$$a^2 = a^{2 \times 1 + 0} = (a^1)^2$$

$$a^1 = a^{2 \times 0 + 1} = a \times (a^0)^2$$

$$a^{21} = a^{2 \times 10 + 1} = a \times (a^{10})^2$$

$$a^{10} = a^{2 \times 5 + 0} = (a^5)^2$$

$$a^5 = a^{2 \times 2 + 1} = a \times (a^2)^2$$

$$a^2 = a^{2 \times 1 + 0} = (a^1)^2$$

$$a^1 = a^{2 \times 0 + 1} = a \times (a^0)^2$$

▶ $a^0 = 1$

▶ n pair : $a^n = (a^{n//2})^2$

▶ n impair : $a^n = a \times (a^{n//2})^2$

$$a^{21} = a^{2 \times 10 + 1} = a \times (a^{10})^2$$

$$a^{10} = a^{2 \times 5 + 0} = (a^5)^2$$

$$a^5 = a^{2 \times 2 + 1} = a \times (a^2)^2$$

$$a^2 = a^{2 \times 1 + 0} = (a^1)^2$$

$$a^1 = a^{2 \times 0 + 1} = a \times (a^0)^2$$

▶ $a^0 = 1$

▶ n pair : $a^n = (a^{n//2})^2$

▶ n impair : $a^n = a \times (a^{n//2})^2$

▶ Beaucoup moins d'opérations !

$$a^{21} \rightarrow a^{10} \rightarrow a^5 \rightarrow a^2 \rightarrow a^1 \rightarrow a^0$$

$$\begin{aligned}a^{21} &= a^{2 \times 10 + 1} = a \times (a^{10})^2 \\a^{10} &= a^{2 \times 5 + 0} = (a^5)^2 \\a^5 &= a^{2 \times 2 + 1} = a \times (a^2)^2 \\a^2 &= a^{2 \times 1 + 0} = (a^1)^2 \\a^1 &= a^{2 \times 0 + 1} = a \times (a^0)^2\end{aligned}$$

▶ $a^0 = 1$

▶ n pair : $a^n = (a^{n//2})^2$

▶ n impair : $a^n = a \times (a^{n//2})^2$

▶ Beaucoup moins d'opérations!

$$a^{21} \rightarrow a^{10} \rightarrow a^5 \rightarrow a^2 \rightarrow a^1 \rightarrow a^0$$

```
def puissance_rapide(a,n):  
    if n == 0:  
        return 1  
    elif n%2 == 0:  
        return puissance_rapide(a, n//2) ** 2  
    else:  
        return a * puissance_rapide(a, n//2) ** 2
```

SCRIPT

Merci pour votre attention

Questions



Cours 2 — Boucles while et récursions

Partie I. Remarques générales

Pour ceux qui ont déjà fait du Python

Comparer des nombres

Partie II. Calculs répétitifs

Du problème au programme

Différentes approches

Formule astucieuse

La récurrence

La récurrence : comment calculer ?

Détail du calcul

Programme récursif

 Exercice récursion

Partie III. Boucles while

Les boucles

Début, corps de boucle et fin

Boucle While

Corps de boucle : se tromper et comprendre

Vers l'infini ou au delà !

 Exercice boucle while

Partie IV. Les nombres premiers

Définition

Un algorithme naïf

Un algorithme moins naïf

 Trouver le plus petit facteur

Partie V. Les nombres flottants

Les flottants

De l'inexactitude du calcul flottant

Approximation des nombres décimaux

Partie VI. Exemples

Méthode de Newton : principe

Méthode de Newton : analyse

Méthode de Newton : programme

Algorithme d'Euclide : principe

Algorithme d'Euclide : analyse

Algorithme d'Euclide : programme

Fibonacci : la suite de Fibonacci

Fibonacci : méthode récursive et naïve

Fibonacci : analyse

Fibonacci : méthode itérative et efficace

 Exercice suite convergente

Partie VII. Pour aller plus loin

Sortie de boucle

Maîtriser les boucles infinies

Calcul de puissance

Exponentiation rapide

Partie VIII. Table des matières