Algorithmique et Programmation

Révisions

Elise Bonzon

elise.bonzon@mi.parisdescartes.fr

LIPADE - Université Paris Descartes
http://www.math-info.univ-paris5.fr/∽bonzon/

Révisions

1. Variables, données et entrées/sorties
2. Algèbre de Boole
3. Les instructions conditionnelles
4. Instructions répétitives
5. Les chaînes de caractères
6. Les listes
7. Les ensembles
8. Fonctions et procédures
9. Modules de fonctions
10. Algorithmique
11. Gestion des exceptions
12. Fichiers textuels

Variables, données et entrées/sorties

Variable

Variable

Une variable est une **zone de la mémoire** dans laquelle une **valeur** est stockée. Une variable possède 4 propriétés :

- un nom
- une adresse
- un type
- une valeur

Variable

Variable

Une variable est une **zone de la mémoire** dans laquelle une **valeur** est stockée. Une variable possède 4 propriétés :

- un nom
- une adresse
- un type
- une valeur

- En Python, la **déclaration** d'une variable et son **initialisation** se font en même temps.

Variable

Variable

Une variable est une **zone de la mémoire** dans laquelle une **valeur** est stockée. Une variable possède 4 propriétés :

- un nom
- une adresse
- un type
- une valeur

- En Python, la **déclaration** d'une variable et son **initialisation** se font en même temps.
- L'attribution d'une valeur à une variable s'appelle l'affectation.

Identificateurs

Identificateur

Les noms des variables (et des fonctions) sont appelés des identificateurs.

Règles de formation des identificateurs :

- Peut contenir des lettres (minuscules 'a' ... 'z', majuscules 'A' ... 'Z'), des chiffres ('0' ... '9'), et le caractère de soulignement ('_')
- Le premier caractère doit être une lettre
- Sensible à la casse
  - `test`, `Test` et `TEST` représentent 3 variables différentes
- Ne doit contenir ni espace, ni caractères spéciaux
- Eviter les mots réservés de Python 3

Mots réservés de Python 3

Python 3 contient 33 mots clés :

and	del	from	None	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

Conventions lexicales

- Un bon programmeur doit veiller à ce que ses lignes d'instructions soient faciles à lire : un identificateur doit être aussi explicite que possible !
- Importance d'utiliser une **politique cohérente** de nommage des identificateurs :
  - **Constantes** : tout en majuscules – `MACONSTANTE`
  - Sinon, tout en minuscules, sauf à l'intérieur du nom pour augmenter la lisibilité – `somme, test, tableDesMatieres`

Affectation et égalité

Attention : affecter n'est pas comparer !

Attention de ne pas confondre **affectation** et **égalité mathématique** !

- L'affectation (=) a un **effet** (associe une valeur à une variable), mais n'a pas de **valeur**
- La comparaison (==) a une **valeur** (`True` si la comparaison est vraie, `False` sinon), mais n'a pas d'**effet**

Affectation et égalité

Attention : affecter n'est pas comparer !

Attention de ne pas confondre **affectation** et **égalité mathématique** !

- L'affectation (=) a un **effet** (associe une valeur à une variable), mais n'a pas de **valeur**
- La comparaison (==) a une **valeur** (`True` si la comparaison est vraie, `False` sinon), mais n'a pas d'**effet**

```
>>> x = 3      #la valeur 3 est affectée à x
>>> y = 5      #la valeur 5 est affectée à y
>>> x == y     #est-ce que x est égal à y ?
False
```

Les types de données

Types

Le type d'un objet Python détermine de quelle sorte d'objet il s'agit.

La fonction `type()` fournit le type d'une valeur

Les types de données

Types

Le type d'un objet Python détermine de quelle sorte d'objet il s'agit.

La fonction `type()` fournit le type d'une valeur

Les principaux types de données en Python sont :

- les entiers, type `int`
- les réels (ou flottants), type `float`
- les booléens, type `bool`
- les chaînes de caractères, type `str`
- les listes, type `list`
- les tuples (ou n-uplets), type `tuple`
- les dictionnaires, type `dict`
- les ensembles, type `set`

Les types en Python

Typage des variables

Le typage des variables sous Python est un typage dynamique.

- Inutile de définir manuellement le type des variables en Python
- Il suffit d'assigner une valeur à un nom de variable pour que celle ci soit automatiquement créée avec le type qui correspond le mieux à la valeur fournie

Les entiers, le type `int`

Les opérations sur les entiers

- l'opposé (opération unaire, notée -)
- l'addition (opération binaire, notée +)
- la soustraction (opération binaire, notée -)
- la multiplication (opération binaire, notée *)
- la puissance (opération binaire, notée **)
- la division (opération binaire, notée /)
- la division entière (opération binaire, notée //)
- le reste de la division entière (opération binaire, notée %)

Attention la multiplication n'est pas implicite, le symbole * doit toujours être indiqué explicitement entre les deux opérandes.

Ordre de priorité des opérateurs

PEMDAS

1. P : parenthèses
2. E : exposant
3. MD : multiplication et division
4. AS : additions et soustractions

A priorité égale, les opérations sont évaluées de la gauche vers la droite

Opération	Valeur
$5 + 4 * 2$	13
$(5 + 4) * 2$	18

Les entrées et sorties

La saisie

La fonction `input()` permet d'affecter à une variable une valeur tapée sur le clavier.

La valeur retournée est toujours du type str, mais on peut ensuite changer le type (on dit aussi transtyper).

Les entrées et sorties

La saisie

La fonction `input()` permet d'affecter à une variable une valeur tapée sur le clavier.

La valeur retournée est toujours du type `str`, mais on peut ensuite changer le type (on dit aussi transtyper).

L'écriture

La fonction `print()` permet d'afficher la représentation textuelle des informations qui lui sont données en paramètre.

Algèbre de Boole

Conditions

Condition

Une **condition** est une expression booléenne, ou booléen, de type `bool`, dont la valeur peut être `True` (vraie) ou `False` (faux).

Conditions

Condition

Une **condition** est une expression booléenne, ou booléen, de type `bool`, dont la valeur peut être `True` (vraie) ou `False` (faux).

Opérateurs booléens de base

Le type `bool` admet 3 opérateurs logiques de base :

- La **négation** (non), notée `not` (opérateur unaire)
- La **conjonction** (et), noté `and` (opérateur binaire)
- La **disjonction** (ou), noté `or` (opérateur binaire)

Principe d'évaluation de la négation

`expression`	`not(expression)`
`True`	`False`
`False`	`True`

Principe d'évaluation de la négation

Evaluation de `expression`, et

- Soit la valeur est `True` , dans ce cas retourner `False`
- Soit la valeur est `False` , dans ce cas retourner `True`

Principe d'évaluation de la conjonction

`expression1`	`expression2`	`expression1 and expression2`
`True`	`True`	`True`
`True`	`False`	`False`
`False`	`True`	`False`
`False`	`False`	`False`

Principe d'évaluation de la conjonction

Evaluation de `expression1`, et

- Soit la valeur est `True` , et dans ce cas
  - Evaluation de `expression2`, et
    - Soit la valeur est `True` , et dans ce cas retourner `True`
    - Soit la valeur est `False` , et dans ce cas retourner `False`
- Soit la valeur est `False` , dans ce cas retourner `False`

Attention ! `expression2` peut ne pas être évaluée ! Mode d'évaluation dit paresseux.

Principe d'évaluation de la disjonction

`expression1`	`expression2`	`expression1 or expression2`
`True`	`True`	`True`
`True`	`False`	`True`
`False`	`True`	`True`
`False`	`False`	`False`

Principe d'évaluation de la disjonction

Evaluation de `expression1`, et

- Soit la valeur est `True` , et dans ce cas retourner `True`
- Soit la valeur est `False` , dans ce cas
  - Evaluation de `expression2`, et
    - Soit la valeur est `True` , et dans ce cas retourner `True`
    - Soit la valeur est `False` , et dans ce cas retourner `False`

Attention ! `expression2` peut ne pas être évaluée ! Mode d'évaluation dit paresseux.

Loi de Morgan

Loi de Morgan

Les lois de Morgan permettent de définir la négation des `and` et des `or`.

- `not`(a `and` b) == `not`(a) `or` `not`(b)
- `not`(a `or` b) == `not`(a) `and` `not`(b)

Les instructions conditionnelles

Syntaxe des instructions conditionnelles simples

```
if condition:
    consequent
else:
    alternant
```

- Avec
  - la condition : expression booléenne, de type `bool`
  - le conséquent : instruction ou suite d'instructions
  - l'alternant : instruction ou suite d'instructions
- Remarques :
  - Le bloc `else` peut ne pas être présent
  - Attention à ne pas oublier le `:` et l'indentation
- Attention ! la condition est une expression booléenne !
  - On n'écrit jamais : `condition == True`
  - On n'écrit jamais : `condition == False`

Syntaxe des instructions conditionnelles multiples

```
if condition1:
    conséquent1
elif condition2:
    conséquent2
elif ...

else:
    alternant
```

Evaluer `condition1`.

1. Si `condition1` est `True`, exécuter uniquement `conséquent1`
2. Si `condition1` est `False`, évaluer `condition2`
   - 2.1 Si `condition2` est `True`, exécuter uniquement `conséquent2`
   - 2.2 Si `condition2` est `False`, évaluer `condition3`
     - 2.2.1 ...
3. Si aucune des conditions n'est vraie, on évalue `alternant`

Instructions répétitives

Syntaxe de la boucle `while`

```
while condition:
    instruction_1
    instruction_2
    ...
    instruction_n
autre_instruction
```

- `condition` : **expression booléenne**, de type `bool`, appelée condition de boucle
- `instruction_1`, `instruction_2`, `...`, `instruction_n` sont des **instructions**, qui forment le corps de la boucle

Règle pour un `while`

Il faut obligatoirement qu'une des instructions du corps de la boucle modifie *potentiellement* la valeur de la condition de sortie de la boucle.

Evaluation de la boucle `while`

```
while condition:
    instruction_1
    instruction_2
    ...
    instruction_n
autre_instruction
```

1. On évalue la valeur de `condition`

Evaluation de la boucle `while`

```
while condition:
    instruction_1
    instruction_2
    ...
    instruction_n
autre_instruction
```

1. On évalue la valeur de `condition`
2. Si `condition` n'a pas la valeur `False`, on interprète le corps de la boucle

   ```
   instruction_1
   instruction_2
   ...
   instruction_n
   ```

   et on revient à l'étape 1.

Evaluation de la boucle `while`

```
while condition:
    instruction_1
    instruction_2
    ...
    instruction_n
autre_instruction
```

1. On évalue la valeur de `condition`
2. Si `condition` n'a pas la valeur `False`, on interprète le corps de la boucle

   ```
   instruction_1
   instruction_2
   ...
   instruction_n
   ```

   et on revient à l'étape 1.
3. Si `condition` a la valeur `False`, on sort de la boucle et on interprète

   ```
   autre_instruction
   ```

Algorithme d'Euclide

Algorithme d'Euclide

On veut calculer le PGCD (plus grand commun diviseur) de deux entiers a et b, tels que $a \geq b$.

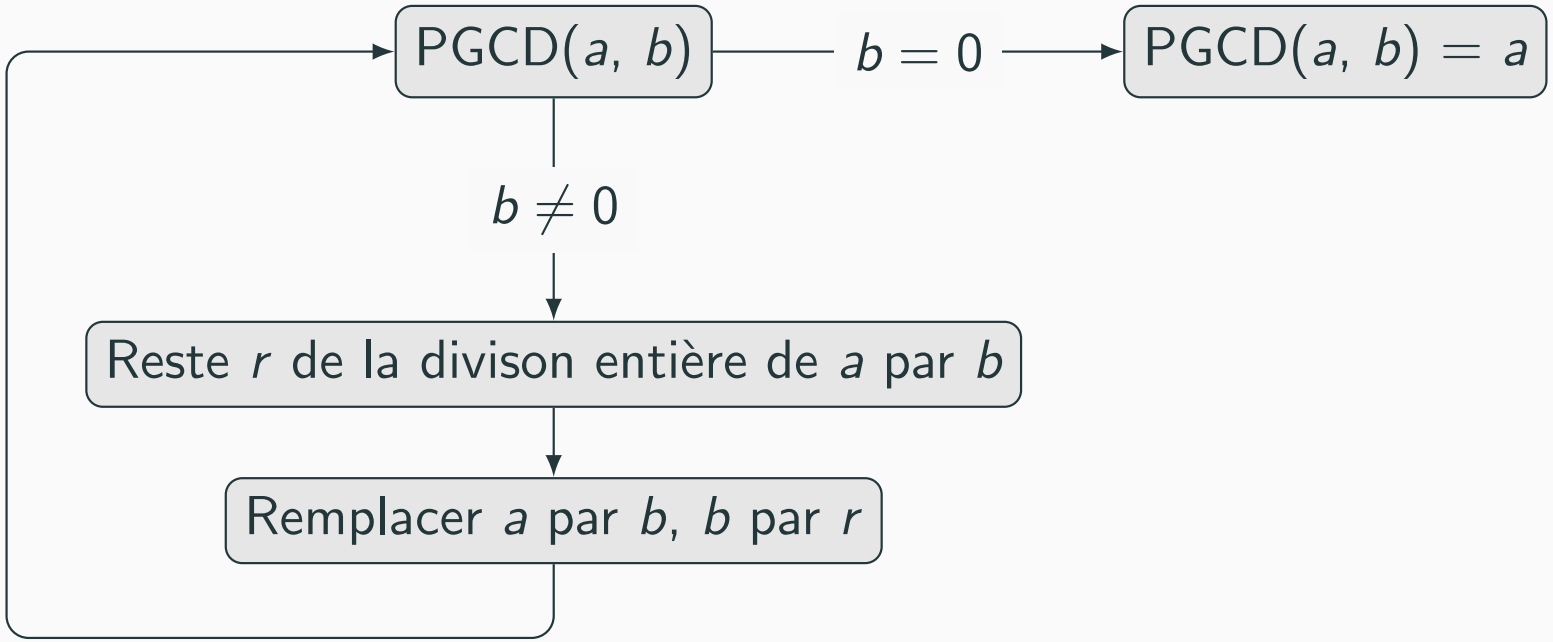
1. Si $b \neq 0$, le PGCD de a et b est le PGCD de b et du reste de la division euclidienne de a par b
2. si $b = 0$, le PGCD de a et b est a

Algorithme d'Euclide

Algorithme d'Euclide

On veut calculer le PGCD (plus grand commun diviseur) de deux entiers a et b, tels que $a \geq b$.

1. Si $b \neq 0$, le PGCD de a et b est le PGCD de b et du reste de la division euclidienne de a par b
2. si $b = 0$, le PGCD de a et b est a



Algorithme d'Euclide : Implémentation

```
# Programme qui calcule le PGCD de deux entiers

a = 147
b = 105

if b > a :  #si b > a, on inverse les valeurs de a et de b
    tmp = a
    a = b
    b = tmp

a_init = a #pour garder la valeur de a pour l'affichage final
b_init = b #pour garder la valeur de b pour l'affichage final

while b != 0 :
    r = a % b #reste de la division euclidienne
    a = b
    b = r

print("Le pgcd de", a_init, "et", b_init, "est", a)
```

Syntaxe de la boucle `for`

```
for var in liste_de_valeurs :
    instruction_1
    instruction_2
    ...
    instruction_n
autre_instruction
```

- La variable `var` prend toutes les valeurs contenues dans `liste_de_valeurs`
- `instruction_1`, `instruction_2`, `...`, `instruction_n` sont des **instructions**, qui forment le corps de la boucle

Intervalle d'entiers : `range`

Intervalle d'entiers : `range`

`range` construit un intervalle d'entiers.

Plusieurs utilisations sont possibles :

- `range`(n) : génère les entiers **de 0 à** $n-1$
- `range`(i, j) : génère les entiers **de** i **à** $j-1$ (si $i > j$, aucun nombre ne sera généré)
- `range`(i, j, k) : génère les entiers **de** i **à** $j-1$**, séparés par un pas de** k

Intervalle d'entiers : `range`

Intervalle d'entiers : `range`

`range` construit un intervalle d'entiers.

Plusieurs utilisations sont possibles :

- `range`(n) : génère les entiers **de 0 à** $n-1$
- `range`(i, j) : génère les entiers **de** i **à** $j-1$ (si $i > j$, aucun nombre ne sera généré)
- `range`(i, j, k) : génère les entiers **de** i **à** $j-1$**, séparés par un pas de** k

- `range`(8) :

Intervalle d'entiers : `range`

Intervalle d'entiers : `range`

`range` construit un intervalle d'entiers.

Plusieurs utilisations sont possibles :

- `range`(n) : génère les entiers **de 0 à** $n-1$
- `range`(i, j) : génère les entiers **de** i **à** $j-1$ (si $i > j$, aucun nombre ne sera généré)
- `range`(i, j, k) : génère les entiers **de** i **à** $j-1$**, séparés par un pas de** k

- `range`(8) : 0, 1, 2, 3, 4, 5, 6, 7

Intervalle d'entiers : `range`

Intervalle d'entiers : `range`

`range` construit un intervalle d'entiers.

Plusieurs utilisations sont possibles :

- `range`(n) : génère les entiers **de 0 à** $n-1$
- `range`(i, j) : génère les entiers **de** i **à** $j-1$ (si $i > j$, aucun nombre ne sera généré)
- `range`(i, j, k) : génère les entiers **de** i **à** $j-1$**, séparés par un pas de** k

- `range`(8) : 0, 1, 2, 3, 4, 5, 6, 7
- `range`(3, 10) :

Intervalle d'entiers : `range`

Intervalle d'entiers : `range`

`range` construit un intervalle d'entiers.

Plusieurs utilisations sont possibles :

- `range`(n) : génère les entiers **de 0 à** $n-1$
- `range`(i, j) : génère les entiers **de** i **à** $j-1$ (si $i > j$, aucun nombre ne sera généré)
- `range`(i, j, k) : génère les entiers **de** i **à** $j-1$**, séparés par un pas de** k

- `range`(8) : 0, 1, 2, 3, 4, 5, 6, 7
- `range`(3, 10) : 3, 4, 5, 6, 7, 8, 9

Intervalle d'entiers : `range`

Intervalle d'entiers : `range`

`range` construit un intervalle d'entiers.

Plusieurs utilisations sont possibles :

- `range`(n) : génère les entiers **de 0 à** $n-1$
- `range`(i, j) : génère les entiers **de** i **à** $j-1$ (si $i > j$, aucun nombre ne sera généré)
- `range`(i, j, k) : génère les entiers **de** i **à** $j-1$**, séparés par un pas de** k

- `range`(8) : 0, 1, 2, 3, 4, 5, 6, 7
- `range`(3, 10) : 3, 4, 5, 6, 7, 8, 9
- `range`(10, 3) :

Intervalle d'entiers : `range`

Intervalle d'entiers : `range`

`range` construit un intervalle d'entiers.

Plusieurs utilisations sont possibles :

- `range`(n) : génère les entiers **de 0 à** $n-1$
- `range`(i, j) : génère les entiers **de** i **à** $j-1$ (si $i > j$, aucun nombre ne sera généré)
- `range`(i, j, k) : génère les entiers **de** i **à** $j-1$**, séparés par un pas de** k

- `range`(8) : 0, 1, 2, 3, 4, 5, 6, 7
- `range`(3, 10) : 3, 4, 5, 6, 7, 8, 9
- `range`(10, 3) : intervalle vide

Intervalle d'entiers : `range`

Intervalle d'entiers : `range`

`range` construit un intervalle d'entiers.

Plusieurs utilisations sont possibles :

- `range`(n) : génère les entiers **de 0 à** $n-1$
- `range`(i, j) : génère les entiers **de** i **à** $j-1$ (si $i > j$, aucun nombre ne sera généré)
- `range`(i, j, k) : génère les entiers **de** i **à** $j-1$**, séparés par un pas de** k

- `range`(8) : 0, 1, 2, 3, 4, 5, 6, 7
- `range`(3, 10) : 3, 4, 5, 6, 7, 8, 9
- `range`(10, 3) : intervalle vide
- `range`(3, 10, 2) :

Intervalle d'entiers : `range`

Intervalle d'entiers : `range`

`range` construit un intervalle d'entiers.

Plusieurs utilisations sont possibles :

- `range`(n) : génère les entiers **de 0 à** $n-1$
- `range`(i, j) : génère les entiers **de** i **à** $j-1$ (si $i > j$, aucun nombre ne sera généré)
- `range`(i, j, k) : génère les entiers **de** i **à** $j-1$**, séparés par un pas de** k

- `range`(8) : 0, 1, 2, 3, 4, 5, 6, 7
- `range`(3, 10) : 3, 4, 5, 6, 7, 8, 9
- `range`(10, 3) : intervalle vide
- `range`(3, 10, 2) : 3, 5, 7, 9

Intervalle d'entiers : `range`

Intervalle d'entiers : `range`

`range` construit un intervalle d'entiers.

Plusieurs utilisations sont possibles :

- `range`(n) : génère les entiers **de 0 à** $n-1$
- `range`(i, j) : génère les entiers **de** i **à** $j-1$ (si $i > j$, aucun nombre ne sera généré)
- `range`(i, j, k) : génère les entiers **de** i **à** $j-1$**, séparés par un pas de** k

- `range`(8) : 0, 1, 2, 3, 4, 5, 6, 7
- `range`(3, 10) : 3, 4, 5, 6, 7, 8, 9
- `range`(10, 3) : intervalle vide
- `range`(3, 10, 2) : 3, 5, 7, 9
- `range`(10, 3, -2) :

Intervalle d'entiers : `range`

Intervalle d'entiers : `range`

`range` construit un intervalle d'entiers.

Plusieurs utilisations sont possibles :

- `range`(n) : génère les entiers **de 0 à** $n-1$
- `range`(i, j) : génère les entiers **de** i **à** $j-1$ (si $i > j$, aucun nombre ne sera généré)
- `range`(i, j, k) : génère les entiers **de** i **à** $j-1$**, séparés par un pas de** k

- `range`(8) : 0, 1, 2, 3, 4, 5, 6, 7
- `range`(3, 10) : 3, 4, 5, 6, 7, 8, 9
- `range`(10, 3) : intervalle vide
- `range`(3, 10, 2) : 3, 5, 7, 9
- `range`(10, 3, -2) : 10, 8, 6, 4

Les chaînes de caractères

Chaîne de caractères

Chaîne de caractères

Une chaîne de caractères est une séquence de caractères. Son type est `str`.

Chaîne de caractères

Chaîne de caractères

Une chaîne de caractères est une séquence de caractères. Son type est `str`.

- La chaîne vide ne contient aucun caractère. Elle est notée `""` ou `''`
  - Attention à ne pas confondre la chaîne vide avec la chaîne contenant un espace ! (`" "` ou `' '`)
- Python ne fait pas de différence entre un caractère et une chaîne de caractères
- Attention de ne pas confondre l'entier `123` et la chaîne de caractères `"123"`
- Attention à la casse : `'Avion'` et `'avion'` sont deux chaînes de caractères différentes

Longueur et concaténation

Longueur d'une chaîne de caractères

L'opérateur `len` retourne la longueur d'une chaîne de caractères, c'est à dire le nombre de caractères qui la compose.

Signature :

$$\texttt{len} :: \texttt{str} \rightarrow \texttt{int}$$

Longueur et concaténation

Longueur d'une chaîne de caractères

L'opérateur `len` retourne la longueur d'une chaîne de caractères, c'est à dire le nombre de caractères qui la compose.

Signature :

$$\texttt{len} :: \texttt{str} \to \texttt{int}$$

Concaténation de chaînes de caractères

L'opérateur `+` permet de concaténer les chaînes de caractères.

Signature :

$$\texttt{+} :: \texttt{str} \times \texttt{str} \to \texttt{str}$$

Duplication et appartenance

Duplication de chaînes de caractères

L'opérateur * permet de dupliquer les chaînes de caractères.

Signature :

$$* :: \texttt{str} \times \texttt{int} \to \texttt{str}$$

Duplication et appartenance

Duplication de chaînes de caractères

L'opérateur `*` permet de dupliquer les chaînes de caractères.

Signature :

$$* :: \texttt{str} \times \texttt{int} \to \texttt{str}$$

Appartenance d'une chaîne de caractères à une autre

L'opérateur `in` permet de déterminer si une chaîne de caractères est incluse dans une autre.

Signature :

$$\texttt{in} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$

Comparaison

Comparaison de chaînes de caractères

Les chaînes de caractères peuvent être comparées au moyen des opérateurs d'égalité ($==$) et d'inégalité ($!=$).

Signatures :

$$\texttt{==} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$
$$\texttt{!=} :: \texttt{str} \times \texttt{str} \to \texttt{bool}$$

Quelques autres fonctions utiles

Fonctions utiles sur les chaînes de caractères

- `str(val)` : converti en `str` la variable `val`

Quelques autres fonctions utiles

Fonctions utiles sur les chaînes de caractères

- `str(val)` : converti en `str` la variable `val`
- `s.lower()` : retourne la chaine `s` où les caractères ont été mis en minuscule

Quelques autres fonctions utiles

Fonctions utiles sur les chaînes de caractères

- `str(val)` : converti en `str` la variable `val`
- `s.lower()` : retourne la chaine `s` où les caractères ont été mis en minuscule
- `s.upper()` : retourne la chaine `s` où les caractères ont été mis en majuscule

Quelques autres fonctions utiles

Fonctions utiles sur les chaînes de caractères

- `str(val)` : converti en `str` la variable `val`
- `s.lower()` : retourne la chaine `s` où les caractères ont été mis en minuscule
- `s.upper()` : retourne la chaine `s` où les caractères ont été mis en majuscule
- `s.capitalize()` : retourne la chaine `s` où la première lettre du premier mot est en majuscule, les autres en minuscule

Quelques autres fonctions utiles

Fonctions utiles sur les chaînes de caractères

- `str(val)` : converti en `str` la variable `val`
- `s.lower()` : retourne la chaine `s` où les caractères ont été mis en minuscule
- `s.upper()` : retourne la chaine `s` où les caractères ont été mis en majuscule
- `s.capitalize()` : retourne la chaine `s` où la première lettre du premier mot est en majuscule, les autres en minuscule
- `s.title()` : retourne la chaine `s` où la première lettre de chaque mot est en majuscule, les autres en minuscule

Quelques autres fonctions utiles

Fonctions utiles sur les chaînes de caractères

- `str(val)` : converti en `str` la variable `val`
- `s.lower()` : retourne la chaine `s` où les caractères ont été mis en minuscule
- `s.upper()` : retourne la chaine `s` où les caractères ont été mis en majuscule
- `s.capitalize()` : retourne la chaine `s` où la première lettre du premier mot est en majuscule, les autres en minuscule
- `s.title()` : retourne la chaine `s` où la première lettre de chaque mot est en majuscule, les autres en minuscule
- `s.swapcase()` : retourne la chaine `s` où les lettres majuscules et minuscules sont inversées.

Indexation des chaînes de caractères

Indexation simple

L'indice d'un caractère dans une chaîne est sa position dans la chaîne.

Les indices sont numérotés à partir de 0.

Caractère	'R'	'a'	'y'	'o'	'n'	's'	' '	'X'
Indice	0	1	2	3	4	5	6	7

Indexation inverse des chaînes de caractères

Indexation inverse

Le *i*ème caractère de la chaîne `chaine`, en lisant de de droite à gauche se récupère avec `chaine[-i]`

Caractère	'R'	'a'	'y'	'o'	'n'	's'	' '	'X'
Indice	0	1	2	3	4	5	6	7
Indice inverse	-8	-7	-6	-5	-4	-3	-2	-1

Découpage de chaînes de caractères

Découpage de chaînes

Le découpage d'une chaîne de caractères permet d'accéder à une portion ou sous-chaîne de la chaîne.

`chaine[i : j]` retourne la sous chaîne de `chaine` située entre les indices i (inclus) et j (exclus)

Les listes

Listes

Listes

Une liste, de type `list` est une collection ordonnée et modifiable d'éléments éventuellement hétérogènes.

Une liste est formée d'éléments séparés par des virgules, et entourés de crochets.

La liste vide, notée `[]`, est une liste qui ne contient aucun élément

Longueur et comparaison

Longueur d'une liste

Comme pour les chaînes de caractères, l'opérateur `len` retourne la longueur d'une liste, c'est à dire le nombre d'éléments qui la compose.

Signature : `len` :: `list` $\rightarrow$ `int`

Longueur et comparaison

Longueur d'une liste

Comme pour les chaînes de caractères, l'opérateur `len` retourne la longueur d'une liste, c'est à dire le nombre d'éléments qui la compose.
Signature : `len` :: `list` $\rightarrow$ `int`

Comparaison de listes

Les listes peuvent être comparées au moyen des opérateurs d'égalité ($==$) et d'inégalité ($!=$).
Deux listes sont **égales** si elles ont la même longueur, et sont composées des mêmes éléments dans le même ordre. Signatures :

$$\texttt{==} :: \texttt{list} \times \texttt{list} \rightarrow \texttt{bool}$$
$$\texttt{!=} :: \texttt{list} \times \texttt{list} \rightarrow \texttt{bool}$$

Appartenance et concaténation

Appartenance d'un élément à une liste

L'opérateur `in` permet de déterminer si un élément est appartient à une liste.

Signature :

$$\texttt{in} :: \texttt{elem} \times \texttt{list} \rightarrow \texttt{bool}$$

Appartenance et concaténation

Appartenance d'un élément à une liste

L'opérateur `in` permet de déterminer si un élément est appartient à une liste.

Signature :

$$\texttt{in} :: \texttt{elem} \times \texttt{list} \rightarrow \texttt{bool}$$

Concaténation de listes

L'opérateur `+` permet de concaténer les listes.

Signature :

$$\texttt{+} :: \texttt{list} \times \texttt{list} \rightarrow \texttt{list}$$

Mutabilité

Les types que nous avons traités jusque ici (`int`, `bool`, `str`) sont immutables :

- on ne peut pas les modifier
- on peut remplacer la valeur d'une variable (`a = a + b`)
- Pour supprimer un élément d'une chaîne de caractères, on reconstruit la chaîne sans cet élément

Mutabilité

Un objet mutable peut être modifié : remplacement, suppression ou ajout d'une partie de l'objet.

Les **listes** sont mutables.

Ajout en fin de liste

Méthode `append`

La méthode `append` ajoute un élément à la fin de la liste depuis laquelle elle est appelée.

Ajout en fin de liste

Méthode append

La méthode append ajoute un élément à la fin de la liste depuis laquelle elle est appelée.

Syntaxe :

```
<liste>.append(<element>)
```

- `<liste>.append(<element>)` modifie la liste `<liste>` en lui ajoutant `<element>` à la fin
- La méthode append ne s'applique pas aux autres types de séquence
- Cette méthode ne **retourne rien** et modifie directement la liste

Indexation des listes

Indexation simple

L'indice d'un élément dans une liste est sa position dans la liste.
Les indices sont numérotés à partir de 0. Il est possible d'utiliser les indices inverses.

Par exemple, la liste `["le", "petit", "chat", "dort", "dans", "son", "couffin"]`

Elément	"le"	"petit"	"chat"	"dort"	"dans"	"son"	"couffin"
Indice	0	1	2	3	4	5	6
Ind. inverse	-7	-6	-5	-4	-3	-2	-1

Indexation et découpage des listes

Indice d'un élément dans une liste

`liste[i]` est l'élément en position *i* dans la liste `liste`.

On accède à la sous-liste de la liste `liste` qui commence à l'indice `i` **inclus**, et finit à l'indice `j` **exclus** par `liste[i:j]`

Raccourcis :

- `liste[i:]` s'évalue en `liste[i: len(liste)]`
- `liste[:j]` s'évalue en `liste[0:j]`
- `liste[:]` s'évalue en `liste[0: len(liste)]`

Découpage des listes avec pas

Découpage avec pas positif

`liste[i:j:k]` permet d'accéder à tous les éléments de la liste `liste`, compris entre les indices `i` (inclus) et `j` (exclus) avec un pas de `k`.

- Par exemple, `liste[4:11:2]` renvoie la liste `[liste[4], liste[6], liste[8], liste[10]]`
- `liste[i:j:1]` correspond à `liste[i, j]`

Découpage des listes avec pas

Découpage avec pas positif

`liste[i:j:k]` permet d'accéder à tous les éléments de la liste `liste`, compris entre les indices `i` (inclus) et `j` (exclus) avec un pas de `k`.

- Par exemple, `liste[4:11:2]` renvoie la liste `[liste[4], liste[6], liste[8], liste[10]]`
- `liste[i:j:1]` correspond à `liste[i, j]`

Découpage avec pas négatif

`liste[i:j:-k]` permet d'accéder à tous les éléments de la liste `liste`, compris entre les indices `i` (inclus) et `j` (exclus) avec un pas de `k`.

- Par exemple, `liste[11:4:-2]` renvoie la liste `[liste[11], liste[9], liste[7], liste[5]]`

Listes imbriquées

Indices des listes imbriquées

Pour extraire un élément d'une **liste de listes**, il suffit d'enchaîner les indices : d'abord l'indice dans la liste des listes ; puis l'indice dans la liste sélectionnée

```
>>> liste = [[1, 2], [3, 4], [5, 6]]
>>> liste[1]
[3, 4]
>>> liste[1][0]
3
>>> liste2 = [liste, [[7, 8], [9, 10]]]
>>> liste2
[[[1, 2], [3, 4], [5, 6]], [[7, 8], [9, 10]]]
>>> liste2[0]
[[1, 2], [3, 4], [5, 6]]
>>> liste2[0][1]
[3, 4]
>>> liste2[0][1][0]
3
```

Syntaxe d'une construction par compréhension

```
[<expr> for <var> in <seq>]
```

- `<var>` : une variable de compréhension
- `<expr>` : une expression pouvant contenir `<var>`
- `<seq>` : une séquence (`range`, `str` ou `list`)
- Construit la liste composée des éléments :
  1. Le **premier** élément est la valeur de `<expr>` dans laquelle la variable `<var>` a pour valeur le **premier** élément de `<seq>`
  2. Le **deuxième** élément est la valeur de `<expr>` dans laquelle la variable `<var>` a pour valeur le **deuxième** élément de `<seq>`
  3. ...
  4. Le **dernier** élément est la valeur de `<expr>` dans laquelle la variable `<var>` a pour valeur le **dernier** élément de `<seq>`

Syntaxe d'une construction par compréhension conditionnée

```
[<expr> for <var> in <seq> if <condition>]
```

- `<var>` : une variable de compréhension
- `<expr>` : une expression pouvant contenir `<var>`
- `<seq>` : une séquence (`range`, `str` ou `list`)
- `<condition>` : une expression booléenne portant sur `<var>`
- Construit la liste de la même façon, en ne retenant que les éléments pour lesquels la `<condition>` est `True`

Syntaxe d'une construction par compréhension conditionnée

```
[<expr1> if <condition> else <expr2> for <var> in <seq>]
```

- `<var>` : une variable de compréhension
- `<seq>` : une séquence (`range`, `str` ou `list`)
- `<condition>` : une expression booléenne portant sur `<var>`
- `<expr1>` : une expression pouvant contenir `<var>`
- `<expr2>` : une expression pouvant contenir `<var>`
- Si `<condition>` est `True`, `<expr1>` sera appliqué ; autrement `<expr2>` sera appliqué

Syntaxe d'une construction par compréhension multiple

```
[<expr> for <var1> in <seq1> for <var2> in <seq2> ... ]
```

- `<var1>` : une variable de compréhension
- `<seq1>` : une séquence (`range`, `str` ou `list`)
- `<var2>` : une variable de compréhension
- `<seq2>` : une séquence (`range`, `str` ou `list`)
- ...
- `<expr>` : une expression pouvant contenir `<var1>`, `<var2>`, ...

Syntaxe d'une construction par compréhension complètes

```
[<expr> for <var1> in <seq1> if <cond1>
        for <var2> in <seq2> if <cond2>
        ... ]
```

- `<var1>` : une variable de compréhension
- `<seq1>` : une séquence (`range`, `str` ou `list`)
- `<cond1>` : une expression booléenne portant sur `<var1>`
- `<var2>` : une variable de compréhension
- `<seq2>` : une séquence (`range`, `str` ou `list`)
- `<cond2>` : une expression booléenne portant sur `<var2>`
- ...
- `<expr>` : une expression pouvant contenir `<var1>`, `<var2>`, ...

Les ensembles

Ensembles

Ensembles

Un ensemble, de type `set` est une collection non ordonnée d'éléments uniques.

Un ensemble est formé d'éléments séparés par des virgules, et entourés d'accolades.

L'ensemble vide, noté `set()`, est un ensemble qui ne contient aucun élément.

Ensembles

Ensembles

Un ensemble, de type `set` est une collection non ordonnée d'éléments uniques.

Un ensemble est formé d'éléments séparés par des virgules, et entourés d'accolades.

L'ensemble vide, noté `set()`, est un ensemble qui ne contient aucun élément.

Un `set` est une transposition informatique de la notion d'ensemble mathématiques.

Opérations ensemblistes

Soit E et F deux ensembles, et x un élément quelconque

Notation Python	Notation mathématique
`len(E)`	$\lvert E \rvert$: le cardinal de E
`set()`	$\emptyset$: l'ensemble vide
`x in E`	$x \in E$: l'appartenance
`x not in E`	$x \notin E$: la non-appartenance
`E < F`	$E \subset F$: l'inclusion stricte
`E <= F`	$E \subseteq F$: l'inclusion large
`E & F`	$E \cap F$: l'intersection
`E | F`	$E \cup F$: l'union
`E - F`	$E \setminus F$: la différence

Mutabilité

Les **ensembles** sont mutables.

Comme les ensembles ne sont pas ordonnés, la notion d'indice n'a pas de sens.

Soit un ensemble `s` de type `set`

- La méthode `add` ajoute un élément à `s` : `s.add(2)`
- La méthode `update` ajoute plusieurs éléments à `s` : `s.update({4, 5, 6})`
- Les méthodes `remove` et `discard` suppriment un élément de `s` : `s.remove(4)` ou `s.discard(4)`
  - La différence est que si l'élément que l'on souhaite supprimer n'appartient pas à l'ensemble, `discard` ne modifiera pas l'ensemble, tandis que `remove` retournera une erreur
- La méthode `clear` supprime tous les éléments de `s` : `s.clear()`

Fonctions et procédures

Fonctions

Fonction

Un fonction est un bloc d'instructions **nommé** et **paramétré**, réalisant une tâche donnée.
Elle admet zéro, un ou plusieurs **paramètres** et **retourne toujours un résultat**.

Fonctions

Fonction

Un fonction est un bloc d'instructions **nommé** et **paramétré**, réalisant une tâche donnée.
Elle admet zéro, un ou plusieurs **paramètres** et **retourne toujours un résultat**.

- Une fonction est donc une suite ordonnée d'instructions qui *retourne une valeur*
- Une fonction joue le rôle d'une *expression*. Elle enrichit le jeu des expressions possibles.
- Par exemple, `len()` est une fonction prédéfinie

Procédures

Procédure

Un procédure est un bloc d'instructions **nommé** et **paramétré**, réalisant une tâche donnée.
Elle admet zéro, un ou plusieurs **paramètres** et **ne retourne pas de résultat**.

Procédures

Procédure

Un procédure est un bloc d'instructions **nommé** et **paramétré**, réalisant une tâche donnée.
Elle admet zéro, un ou plusieurs **paramètres** et **ne retourne pas de résultat**.

- Une procédure est donc une suite ordonnée d'instructions qui *ne retourne pas de valeur*
- Une procédure joue le rôle d'une *instruction*. Elle enrichit le jeu des instructions possibles.
- Par exemple, `print()` est une procédure prédéfinie

Procédures

Procédure

Un procédure est un bloc d'instructions **nommé** et **paramétré**, réalisant une tâche donnée.
Elle admet zéro, un ou plusieurs **paramètres** et **ne retourne pas de résultat**.

- Une procédure est donc une suite ordonnée d'instructions qui *ne retourne pas de valeur*
- Une procédure joue le rôle d'une *instruction*. Elle enrichit le jeu des instructions possibles.
- Par exemple, `print()` est une procédure prédéfinie

Une **fonction** *vaut* quelque chose (son *retour*), une **procédure** *fait* quelque chose.

Syntaxe d'une fonction ou d'une procédure

```
def nom_fonction(liste_de_parametres) :
    """ chaine de documentation """
    bloc_instructions
```

- `nom_fonction` : doit respecter les règles suivantes :
  - Aucun caractère spécial (hormis "_"), aucun caractère accentué
  - Commence par une minuscule (les majuscules seront utilisées pour les classes)
  - Choisir un nom **suffisamment explicite** !
- `liste_de_parametres` : paramètres de la fonction, séparés par une virgule. **Peut-être vide**.
- `chaine de documentation` : facultative mais **fortement** conseillée. Doit contenir :
  - la **signature** de la fonction
  - une liste d'expressions booléennes qui précisent les **conditions d'application** de la fonction si besoin
  - une phrase qui explique **ce que fait la fonction**
- Attention : l'indentation délimite la fonction ; ne pas oublier les :

Instruction `assert`

Vérification des préconditions

Les **préconditions** que doivent vérifier les paramètres d'entrée d'une fonction ou procédure seront définies grâce à l'instruction `assert`.
A l'exécution du code, cette instruction *lèvera une exception* si la condition testée est fausse.
Les préconditions seront placées juste après l'en-tête de la fonction.

```
assert expr[, message]
```

- `expr` est une **expression booléenne**.
  - Si `expr == True`, on passe à l'instruction suivante
  - Sinon, l'exécution est interrompue, et une exception `AssertionError` est levée
  - Dans ce cas, le `message`, **optionnel**, est affiché

Tests des fonctions

Jeu de tests

Les fonctions doivent être testées grâce à l'instruction `assert`.

```
>>> assert tables(2, 1, 5) == [2, 4, 6, 8, 10]
>>> assert tables(0, 0, 0) == [0]
>>> assert tables(2, 8, 4) == []
```

- Les jeux de tests servent au programmeur pour valider la définition d'une fonction
- Ils doivent couvrir tous les cas possibles
  - les cas de base
  - les cas extrêmes
- Ils n'affichent rien, sauf si un test ne passe pas

Variables locales, variable globales

Variables locales, variables globales

Les variables définies **à l'intérieur** du corps d'une fonction ou d'une procédure ne sont accessibles qu'à la fonction elle-même. Ce sont des variables locales à la fonction.

Le contenu des variables locales est **inaccessible depuis l'extérieur de la fonction**.

Variables locales, variable globales

Variables locales, variables globales

Les variables définies **à l'intérieur** du corps d'une fonction ou d'une procédure ne sont accessibles qu'à la fonction elle-même. Ce sont des variables locales à la fonction.
Le contenu des variables locales est **inaccessible depuis l'extérieur de la fonction**.

Les variables définies **à l'extérieur** d'une fonction ou procédure sont des variables globales. Leur contenu est « visible » de l'intérieur d'une fonction, mais la fonction **ne le modifie pas**.

Chaîne de documentation

- La chaîne de documentation placée au début des fonctions et procédures ne joue aucun rôle fonctionnel dans le script
- Elle est traitée comme *un simple commentaire* par Python
- Mais elle est mémorisée à part dans un système de documentation interne automatique

Chaîne de documentation

- La chaîne de documentation placée au début des fonctions et procédures ne joue aucun rôle fonctionnel dans le script
- Elle est traitée comme *un simple commentaire* par Python
- Mais elle est mémorisée à part dans un système de documentation interne automatique

```
>>> def factorielle(n):
...     """Int --> Int
...     Fonction retournant la factorielle de n
...     """
...     assert type(n) is int, "Factorielle d'un entier"
...     fact = 1
...     for i in range(1, n+1) :
...         fact = fact * i
...     return(fact)
...
>>> print(factorielle.__doc__)
Int --> Int
Fonction retournant la factorielle de n
```

Modules de fonctions

Les modules en Python

Modules

Un module de fonctions est un fichier qui regroupe des ensembles de fonctions.

Un module peut également regrouper d'autres outils (classes, données...). On utilise souvent le terme de *bibliothèque*.

Les modules en Python

Modules

Un module de fonctions est un fichier qui regroupe des ensembles de fonctions.

Un module peut également regrouper d'autres outils (classes, données...). On utilise souvent le terme de *bibliothèque*.

- L'utilisation des modules est très fréquente. Elle permet de :
  - ré-utiliser du code
  - isoler, dans un espace identifié, des fonctionnalités particulières
- Il existe un grand nombre de modules fournis d'office avec Python
- Il est également possible de définir ses propres modules

Modules de fonction

- Le **nom d'un fichier** en Python se termine par `.py` et ne contient que des lettres minuscules, des chiffres et des soulignés. Aucun espace !
  - `essai2.py` ; `essai_tortue.py` ; ~~`Essai Tortue.py`~~
- Un module est un fichier `nom_fichier.py` écrit en Python et contenant :
  - des définitions
  - des instructions (par exemple d'affichage)
- Un module peut être destiné :
  - à être directement exécuté. Lorsqu'il est court et effectue une action ré-utilisable, on parle souvent d'un script
  - à être utilisé par un autre module. Il exporte alors un certain nombre de fonctionnalités.

Syntaxes d'import d'un module

3 syntaxes possibles pour importer un module, ou les fonctions d'un module, dans un autre fichier.

```
import nom_module

from nom_module import nom1, nom2...

from nom_module import *
```

Obtenir de l'aide sur les modules importés

```
>>> import tva
>>> help(tva)
Help on module tva:
NAME
tva - #Fichier tva.py
FUNCTIONS
prix_ttc(p)
DATA
COEFF = 1.196
TVA = 19.6
FILE
/PATH/tva.py
```

- Pour vous **déplacer dans l'aide**, utilisez les flèches du haut et du bas, ou les touches page-up et page-down
- Pour **quitter l'aide**, appuyez sur la touche Q.
- Pour **chercher du texte**, tapez / puis le texte que vous cherchez puis la touche Entrée.
- Pour obtenir de l'aide sur une fonction : `help(tva.prix_ttc)`

Obtenir la liste des noms définis

```
>>> dir(tva)
['COEFF', 'TVA', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__',
'__spec__', 'prix_ttc']
>>> dir()
['__annotations__', '__builtins__', '__doc__',
'__loader__', '__name__', '__package__', '__spec__', 'tva']
```

- La fonction interne `dir()` est utilisée pour trouver quels noms sont définis par un module. Elle donne une liste de chaînes classées par ordre lexicographique
- Sans paramètre, `dir()` liste les noms actuellement définis
- La fonction `dir()` ne liste pas les noms des fonctions et variables natives. Si vous en voulez la liste, ils sont définis dans le module standard `__builtin__` : `dir(__builtin__)`

Modules courants

- Il existe de nombreux modules de base en Python
- La liste complète est présente à l'adresse :
  https://docs.python.org/fr/3/py-modindex.html
- N'hésitez pas à la parcourir !

Quelques modules qui vous seront utiles

- `math` : fonctions et constantes mathématiques de base (`sin, cos, exp, pi`...).
- `cmath` : fonctions et constantes mathématiques avec des nombres complexes
- `sys` : interaction avec l'interpréteur Python, passage d'arguments
- `random` : génération de nombres aléatoires.
- `fractions` : fournit un support de l'arithmétique des nombres rationnels.

Et bien d'autres !

Algorithmique

Algorithme

Algorithme

Un algorithme est une suite ordonnée d'instructions qui indique la démarche à suivre pour résoudre une série de problèmes équivalents.

- **Validité** : aptitude à réaliser exactement la tâche pour laquelle il a été conçu
- **Robustesse** : aptitude à se protéger de conditions anormales d'utilisation
- **Réutilisabilité** : aptitude à être réutilisé pour résoudre des tâches équivalentes à celle pour laquelle il a été conçu
- **Complexité** : nombre d'instructions élémentaires à exécuter pour réaliser la tâche pour laquelle il a été conçu
- **Efficacité** : aptitude à utiliser de manière optimale les ressources du matériel qui l'exécute

Programme

Programme

Un programme est une suite d'instructions définies dans un langage donné.
Un programme permet de décrire un algorithme.

- Un algorithme exprime la structure logique d'un programme : il est indépendant du langage de programmation
- La traduction de l'algorithme dans un langage de programmation dépend du langage choisi

Pourquoi l'étude des algorithmes ?

- L'étude des algorithmes est fondamentale en informatique.
- L'analyse rigoureuse des algorithmes proposés permet de les **valider**, d'**évaluer leur complexité** et parfois de **justifier de leur optimalité**
- Il faut être capable
  - de s'assurer qu'un programme se termine toujours
  - d'estimer son temps d'exécution pour des valeurs données
  - de déterminer les conditions d'utilisation, de saturation

Pourquoi étudier la complexité des algorithmes ?

- Pour savoir si un algorithme est "efficace" ou non
- Pour pouvoir comparer deux algorithmes accomplissant la même tâche
- Pour chaque algorithme, on veut déterminer :
  - le temps d'execution
  - la place utilisée en mémoire
  - indépendemment de l'implémentation (langage choisi pour programmer, machine utilisée)
- On ne veut pas :
  "l'algorithme A, implémenté sur la machine M dans le langage L et exécuté sur la donnée D utilise k secondes de calcul et j bits de mémoire"
- On veut :
  "Quels que soient l'ordinateur et le langage utilisés, l'algorithme A_1 est **meilleur** *que l'algorithme A_2, pour des données de grandes tailles"*

Qu'est-ce que la complexité d'un algorithme ?

- Il s'agit de caractériser le comportement d'un algorithme sur l'ensemble D_n des données de taille n
- La complexité dépend en général de la taille n des données
- Plusieurs types de complexité :
  - En temps
  - En espace

Opérations significatives

- Opérations significatives : le temps d'exécution d'un algorithme est toujours proportionnel au nombre de ces opérations
- Dans ce cours, accès à un élément d'une liste :
  - Comparaison entre deux éléments d'une liste
  - Affectation d'un élément à une liste
  - ...
- Si plusieurs opérations significatives différentes sont choisies, elles doivent être décomptées séparément
- En changeant le nombre d'opérations significatives, on varie le degrés de précision de l'analyse

Définitions et notations

- $\text{coût}_A(d)$: complexité de l'algorithme A sur la donnée $d \in D_n$ de taille n
- Complexité au **meilleur des cas** :

$$\text{coût } min_A(n) = min\{\text{coût}_A(d), d \in D_n\}$$

- Complexité au **pire des cas** :

$$\text{coût } max_A(n) = max\{\text{coût}_A(d), d \in D_n\}$$

- Complexité **moyenne** :

$$\text{coût } moy_A(n) = \sum_{d \in D_n} \text{coût}_A(d) \times p(d)$$

Complexité

- Plus la taille des données est grande, plus les écarts en temps se creusent
- Les algorithmes utilisables pour les données de grande taille sont ceux qui s'exécutent en un temps
  - constant
  - logarithmique (Ex : recherche dichotomique)
  - linéaire (Ex : recherche séquentielle)
  - $n \log n$ (Ex : bons algorithmes de tri)
- Les algorithmes qui prennent un temps polynomial ne sont utilisables que pour des données de très petite taille

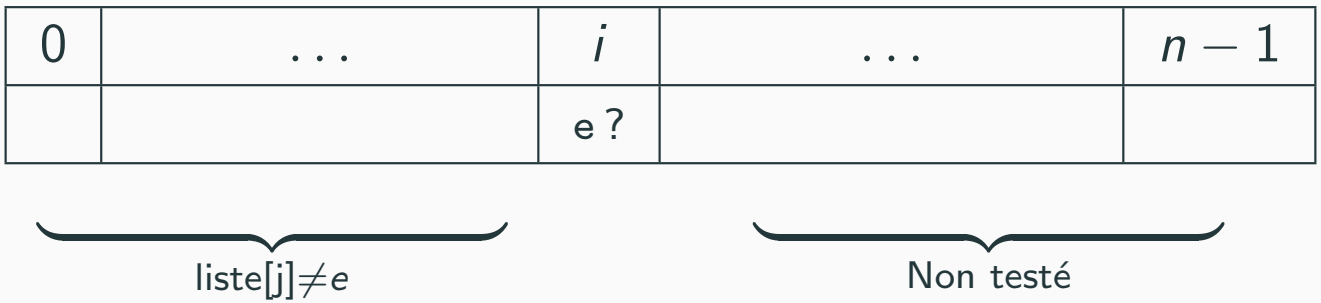
Recherche séquentielle dans une liste non triée

- Soit `liste` une liste non triée de longueur n, de type `list[elem]`, et `e` une élément de type `elem` .
- On cherche s'il existe un indice $i \in [0, n-1]$ tel que `liste[i] == e`

Algorithme de recherche séquentielle dans une liste non triée

Parcourir la liste : pour tout $i \in [0, n-1]$, faire :

- Si `liste[i] == e`, retourner `True`
- Sinon, si $i == n-1$, retourner `False`
- Sinon, $i = i + 1$



Recherche séquentielle dans une liste non triée

```
def recherche_sequentielle_liste_non_triee(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste non triée"""
    appartient = False
    i = 0
    n = len(liste)
    while i < n and not(appartient) :
        if liste[i] == elem :
        appartient = True
        i = i + 1
    return appartient
```

Liste triée

Liste triée

Une liste de longueur n, de type `list[elem]` est triée si et seulement si :

$$\forall i \in [0, n-2],\ \text{liste[i]} \leq \text{liste[i+1]}$$

Si $n = 0$ ou $n = 1$, la liste est triée

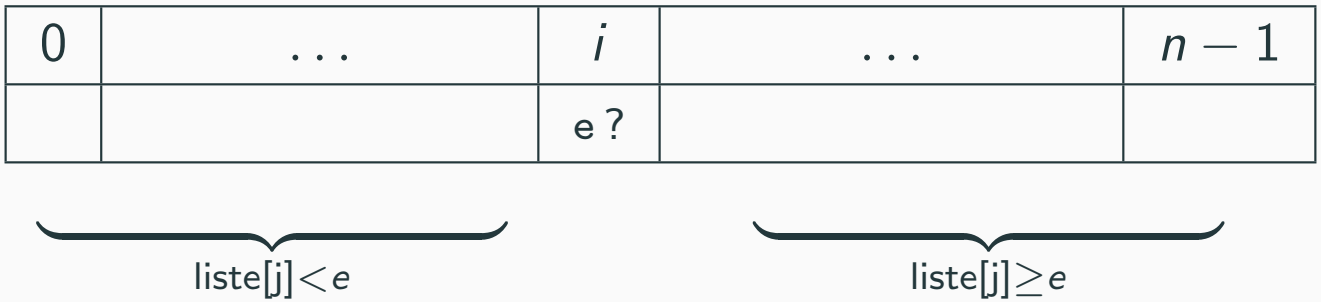
```
def verif_liste_triee(liste):
    """List --> Bool.
    Vérifie si la liste est triée"""
    trie = True
    i = 0
    n = len(liste)
    while i<(n - 1) and trie :
        if liste[i] > liste[i+1]:
        trie = False
        i = i + 1
    return trie
```

Recherche séquentielle dans une liste triée

- Soit `liste` une liste **triée** de longueur n, de type `list[elem]`, et `e` une élément de type `elem` .
- On cherche s'il existe un indice $i \in [0, n-1]$ tel que `liste[i] == e`

Algorithme de recherche séquentielle dans une liste triée

- Si `e > liste[n-1]`, retourner `False`
- Sinon, parcourir la liste : tant que `liste[i] < e`, faire $i = i + 1$
- A la sortie de la boucle,
  - Si `liste[i] == e`, retourner `True`
  - Sinon, retourner `False`



Recherche séquentielle dans une liste triée

```
def recherche_sequentielle_liste_triee(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste triée"""
    if elem > liste[len(liste) - 1] :
        return False
    else :
        i = 0
        while liste[i] < elem :
            i = i + 1
        if liste[i] == elem :
            return True
        else :
            return False
```

Recherche dichotomique

- Il est possible d'être beaucoup plus efficace si la liste est triée
- Méthode dichotomique :
  - Comparer l'élément e à une valeur située au milieu de la liste
  - Si cette valeur est différente de e, continuer la recherche sur la demi-liste susceptible de contenir e

Explication de l'algorithme

Partages successifs sur une liste **triée**

- Situation générale

0	...	inf	...	med	...	sup	...	$n-1$

liste[j]$<e$ — A explorer — liste[j]$\geq e$

$$0 \leq inf \leq sup \leq n-1$$

- 2 cas possibles :
  - $inf \leq sup$: on pose $med = \lfloor \frac{(inf+sup)}{2} \rfloor$
    - `e` == `liste[med]`, retourner `True`
    - `e` $<$ `liste[med]`, $sup = med - 1$
    - `e` $>$ `liste[med]`, $inf = med + 1$
  - $inf > sup$: retourner `False`
- Conditions initiales : $inf = 0$, $sup = n - 1$.

Recherche dichotomique dans une liste triée

```
def recherche_dichotomique(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste triée"""
    appartient = False
    inf, sup = 0, len(liste) - 1

    while inf <= sup and not(appartient) :
        med = (inf + sup)//2
        if liste[med] == elem :
            appartient = True
        elif liste[med] > elem:
            sup = med - 1
        else :
            inf = med + 1
    return appartient
```

Tri par sélection

- Soit `liste` une liste non triée de longueur n, de type `list[elem]`
- On veut **trier** la `liste`
- Parcourir la liste : pour tout $i \in [0, n-2]$
  - On cherche le plus petit élément de `liste` pour $j \in [i, n-1]$
  - On **échange** ce minimum avec `liste[i]`

Echange de deux éléments d'une liste

L'algorithme de tri par sélection va utiliser la procédure `echange(liste, i, j)`

- Prend en entrée une liste et deux indices
- Echange les éléments de la liste correspondant à ces deux indices
- Le type `list` étant mutable, il n'est pas utile de retourner la liste donnée en argument d'appel, elle est directement modifiée par la procédure !

```
def echange(liste, i, j):
    """List x Int x Int --> None
    Echange les élements de liste en position i et j"""

    elem = liste[i]
    liste[i] = liste[j]
    liste[j] = elem
```

Tri par sélection

```
def tri_selection(liste) :
    """List --> None -- Trie la liste donnée en paramètre.
    La liste est directement modifiée par la procédure."""
    n = len(liste)
    for i in range(n-1):
        indice_min = i
        for j in range(i+1, n):
            if liste[j] < liste[indice_min]:
                indice_min = j
        if indice_min != i:
            echange(liste, i, indice_min)
```

Tri par insertion

- Algorithme qu'utilise naturellement l'être humain pour trier des objets, comme par exemple des cartes à jouer
- Soit `liste` une liste non triée de longueur n, de type `list[elem]`
- Soit $i \in [1, n-1]$. A l'étape i
  - On suppose que les éléments d'indice 0 à $i-1$ sont déjà triés
  - On insère l'élément d'indice i à sa place dans la liste `liste[0:i-1]` :
    - $pos = i$, sauvegarde de `elem` = `liste[i]`
    - Tant que `liste[pos-1]` $>$ `elem`, `liste[pos]` = `liste[pos-1]` ; $pos = pos - 1$
    - Si $pos == 0$ ou `liste[pos-1]` $<=$ `elem`, `liste[pos]` = `elem`

Tri par insertion

```
def tri_insertion(liste):
    """List --> None
    Tri la liste donnée en paramètre"""
    for i in range(1, len(liste)):
        elem = liste[i]
        pos = i
        while pos > 0 and liste[pos - 1] > elem :
            liste[pos] = liste[pos-1]
            pos = pos - 1
        liste[pos] = elem
```

Tri par comptage

- Principe : déterminer pour chaque élément de `liste` le nombre d'éléments qui lui sont inférieurs ou égaux
- Pour trouver `ind(i)`, $0 \leq i \leq n-2$, donnant la position de `liste[i]` dans la liste triée, on compare `liste[i]` à tous les `liste[j]`, $j \in [i+1, n-1]$:
  - Soit `liste[j] <= liste[i]` : on incrémente $ind(i)$ de 1
  - Soit `liste[j] > liste[i]` : on incrémente $ind(j)$ de 1

Tri par comptage

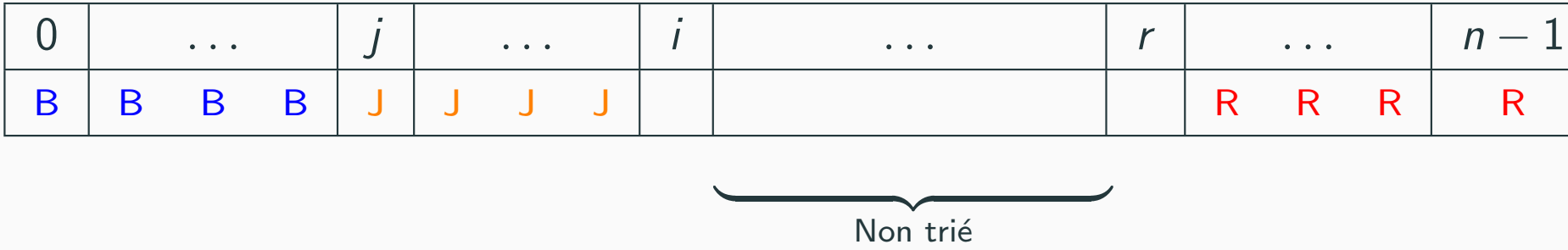
```
def tri_comptage(liste):
    """List --> List
    Tri la liste donnée en paramètre"""
    ind = []
    result = []
    n = len(liste)
    # initialisation des listes indices et résultat
    for i in range(n) :
        ind.append(0)
        result.append(0)
    for i in range(n - 1): #comptage
        for j in range(i+1, n) :
            if liste[j] > liste[i] :
                ind[j] = ind[j] + 1
            else :
                ind[i] = ind[i] + 1

    for i in range(n) : #liste triée résultat
        result[ind[i]] = liste[i]

    return result
```

Algorithme du drapeau à 3 couleurs

- Soit `liste` une liste non triée de longueur n contenant des données de 3 types : *R*ouge, *B*leu et *J*aune. On veut trier la liste de façon à ce que les premiers éléments soient bleus, les suivants jaunes, puis enfin les derniers rouges.



- Deux cas possibles
  - $i = r + 1$. C'est fini, la liste est triée
  - $i \leq r$, 3 cas possibles
    - `liste[i]` $== J$, $i = i + 1$
    - `liste[i]` $== B$, `echange(liste, j, i)`; $i = i + 1$; $j = j + 1$
    - `liste[i]` $== R$, `echange(liste, r, i)`; $r = r - 1$

Algorithme du drapeau

```
def drapeau(liste):
    """List --> None
    Tri la liste, contenant 3 couleurs R, J, B,
    donnée en paramètre"""
    i = 0
    j = 0
    r = len(liste) - 1
    while i <= r:
        if liste[i] == "J":
            i = i + 1
        elif liste[i] == "B" :
            echange(liste, j, i)
            i = i + 1
            j = j + 1
        else :
            echange(liste, r, i)
            r = r - 1
```

Insertion d'un élément dans une liste triée

- Insérer un élément à la fin d'une liste : méthode `append`
- Mais comment insérer un élément *à sa place* dans une liste triée ?
- Nous voulons modifier la liste et ne pas en créer une nouvelle contenant cet élément
- Et ne pas avoir besoin de trier de nouveau toute la liste...
- **Idée :** Utiliser la méthode d'insertion déjà vu dans l'algorithme de **tri par insertion** :
  - Ajouter l'élément en fin de la liste (méthode `append`)
  - Décaler les valeurs de la liste vers la droite, jusqu'à trouver la place de l'élément à insérer

Insertion d'un élément dans une liste triée

```
def insertion(liste, elem):
    """List x Elem --> None
    Insère l'élément elem à sa position dans la liste triée"""
    liste.append(elem)
    n = len(liste)
    indice = n - 1
    while indice > 0 and liste[indice - 1] > elem :
        liste[indice] = liste[indice - 1]
        indice = indice - 1
    liste[indice] = elem
```

Suppression d'un élément dans une liste : algorithmique

- D'un point de vue algorithmique :
  - Rechercher l'élément (recherche dichotomique dans une liste triée, recherche séquentielle dans une liste triée, recherche séquentielle dans une liste non triée...)
  - Suppression de l'élément
  - La méthode de suppression dépend du langage de programmation !

Suppression d'un élément dans une liste : en Python

- Deux fonctions prédéfinies :
  - La **méthode** `remove` supprime la première occurence de l'élément donné en paramètre

```
>>> liste = ['a', 'b', 'c', 'a', 'd', 'b']
>>> liste.remove('b')
>>> liste
['a', 'c', 'a', 'd', 'b']
```

  - La **procédure** `del` supprime l'élément positionné à l'indice donné en paramètre

```
>>> liste = ['a', 'b', 'c', 'a', 'd', 'b']
>>> del liste[1]
>>> liste
['a', 'c', 'a', 'd', 'b']
>>> del liste[2:4]
>>> liste
['a', 'c', 'b']
```

Gestion des exceptions

Gestion des des exceptions : syntaxe

```
try :
    <sequence_instructions1>
except :
    <sequence_instructions2>
```

- Si au cours de l'exécution de la `sequence_instructions1` une exception se produit, l'exécution du bloc est abandonnée, et la `sequence_instructions2` est exécutée
- Si l'exécution de `sequence_instructions1` s'est déroulée normalement, on ne rentre pas dans le bloc `except`

Gestion des des exceptions particulières

- Il y a différents types d'exceptions : `IndexError`, `NameError`, `ZeroDivisionError`...
- Il est possible de ne vouloir gérer que certaines exceptions, ou de faire des traitements différents en fonction de du type d'erreur rencontrée

```
try :
    <sequence_instructions1> #séquence normale d'exécution
except <type_exception1> :
    <sequence_instructions2> #traitement de l'exception 1
except <type_exception2> :
    <sequence_instructions3> #traitement de l'exception 2
...
else :
    #bloc d'instruction exécuté en l'absence d'erreurs
    <sequence_instruction4>
```

Fichiers textuels

Ouverture d'un fichier en écriture – append

```
mon_fichier = open('Monfichier.txt', 'a')
```

- La **fonction intégrée** `open()` permet de créer un *objet-fichier*
- `open()` attend 2 arguments, sous forme de **chaînes de caractères**
  1. Le nom du fichier à ouvrir
  2. Le mode d'ouverture
- L'option `'a'` permet d'ouvrir le fichier en mode ajout (*append*)
  - **S'il existe un fichier du nom indiqué**, les données enregistrées sont ajoutées à la fin du fichier
  - **S'il n'existe pas de fichier de ce nom**, un nouveau fichier est créé

Ouverture d'un fichier en écriture – write

```
mon_fichier = open('Monfichier.txt', 'w')
```

- L'option `'w'` permet d'ouvrir le fichier en mode écriture (*write*)
  - **S'il existe un fichier du nom indiqué**, ce fichier est écrasé, et l'écriture des données commence à partir du début d'un nouveau fichier, vide.
  - **S'il n'existe pas de fichier de ce nom**, un nouveau fichier est créé

Ecriture séquentielle dans un fichier

```
mon_fichier = open('Monfichier.txt', 'a')
mon_fichier.write('Bonjour !\n')
mon_fichier.write('Saperlipopette ! ')
mon_fichier.write('Dit le poète ! ')
mon_fichier.close()
```

- La méthode `write()` écrit les données voulues dans le fichier
  - Les données sont enregistrées les unes à la suite des autres
  - Si le caractère de retour à la ligne `\n` n'est pas indiqué, le prochain `write()` se fera sur la même ligne
- La méthode `close()` ferme le fichier
  - Les écritures sont mises en tampon, elles ne prennent pas forcément effet immédiatement. Elles peuvent ne pas être enregistrées tant que le fichier n'est pas fermé !
  - Ne pas oublier le `close()` donc !

Ouverture d'un fichier en lecture

```
mon_fichier = open('Monfichier.txt', 'r')
```

- L'option `'r'` permet d'ouvrir le fichier en mode lecture (*read*)
- **S'il n'existe pas de fichier de ce nom**, une exception est levée : `FileNotFoundError`
- La méthode `read()` lit la totalité du fichier dans **une seule chaîne de caractères**
- Cette méthode peut également être utilisée avec **un argument** qui indique le nombre de caractères qui doivent être lus, **à partir** de la position déjà atteinte dans le fichier
- La méthode `readline()` permet de lire le fichier ligne à ligne