Initiation : Python pour la SI

Pour les TP systèmes il est nécessaire de savoir :

- *Tracer des courbes issues de données expérimentales ou simulées*
  - *Ouvrir un fichier texte ou csv, mettre les données au bon format, extraire les données*
  - *Tracer les courbes issues de ces données (légendes, titres, etc…)*
- *Comparer ces courbes avec des courbes théoriques*
  - *Être capable de créer une liste correspondant aux données théoriques*
  - *Tracer cette courbe sur la même figure que les courbes précédentes*
- *Maitriser les outils de bases de Python (fonctions, modules, variables, conditions…)*

C'est pourquoi ce cours/TD/TP traite :

- Le Python et la programmation séquentielle
- Les variables et les opérations
- Les conditions et les boucles
- Les fonctions
- Les bibliothèques (modules)
- Tracer des courbes

Remarque : Vous verrez tout cela plus en détail en ITC, mais vous en aurez besoin en TP système.

1. Le Python et la programmation séquentielle

1.1. Le Python

Python a été choisi pour être utilisé en CPGE parce qu'il est **simple à prendre en main** et permet de se concentrer sur **la logique plutôt** que sur **la syntaxe**.
C'est un **langage flexible et polyvalent**. Il est utilisé aussi bien pour des petits scripts que pour des applications complexes. Ce n'est pas qu'un **langage d'apprentissage**, Python est utilisé dans **l'industrie** et dans la **recherche**.
Il existe de très nombreuses bibliothèques adaptées à la modélisation, aux simulations et à l'analyse de données, par exemple.

1.2. La programmation séquentielle

Comme une majorité de langage de programmation, y compris le **Scratch**, **Python** suit une logique de **programmation séquentielle**. Cela signifie que les instructions s'exécutent, ou plutôt s'interprète**, une ligne après l'autre, de haut en bas**.

Les **scripts Python sont une séquence d'instructions** qui se suivent.

Remarque : La programmation séquentielle se distingue de la programmation parallèle, qui, quant à elle, exécute plusieurs instructions en même temps.

1.3. Environnement Python, console, script et notebook

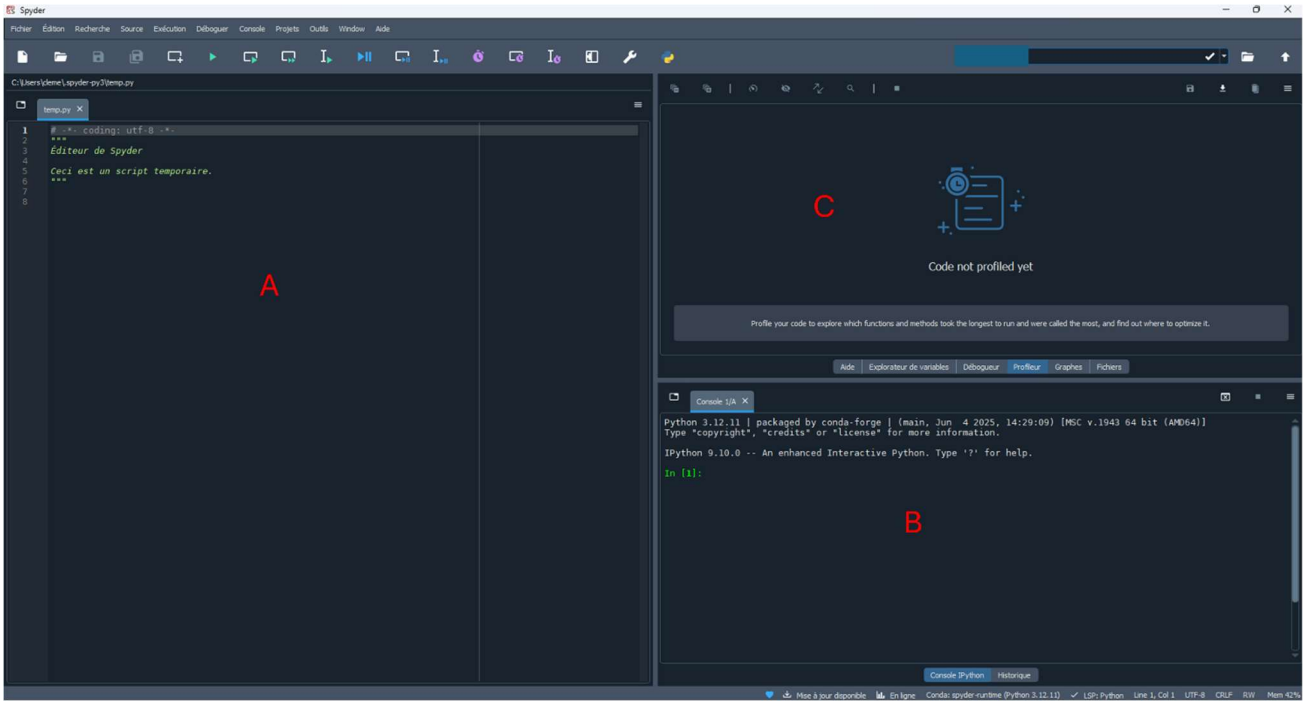
Il existe de nombreux Environnement de Développement Intégré (**IDE**) pour Python, au lycée nous utiliserons **Spyder**, installé sur les ordinateurs de la salle.

On distingue trois zones dans Spyder :

A. L'éditeur ou fenêtre de **script** qui permet de rédiger des scripts (programmes). Il s'agit d'un fichier texte dont l'extension est .py.

B. La console qui permet **d'interagir avec le script** (entrer ou **afficher des données** par exemple) **ou de tester des commandes.**

C. Une zone supplémentaire pour l'affichage de graphique par exemple.



En raison de l'utilisation des sessions informatiques, il faudra parfois faire quelques manipulations.

Attention à ne jamais travailler sur clefs, toujours dans vos documents !

2. Manipuler des variables en Python

2.1. Affectation d'une variable

On peut voir une **variable** comme une **boîte** où l'on peut **stocker** un **nombre**, un **texte** ou **d'autres données**, et y **accéder** plus tard **en utilisant** le **nom** de la variable. En fonction de ce qu'elle contient une variable n'aura pas le même **type**.

Une variable possède trois caractéristiques :

1. Un **nom** qui l'identifie, c'est une chaîne de caractère. Il est de bon usage de donner des noms explicites.
2. Une **valeur** qui peut varier lors du déroulement du programme.

3. Un **type** qui désigne la nature des valeurs stockées. Par exemple, des chiffres, des lettres, des booléens (vrai ou faux).

Q.1. **Remplir le tableau suivant en donnant les caractéristiques des variables à partir de leur déclaration.**

Code	Nom	Valeur	Type
`temps = 3`			
`Led_ON = False`			
`Nom_famille = 'Trotobas'`			
`listePair = [2, 4, 6]`			

A noter que vous pouvez utiliser la fonction `type()` sur une variable qui renvoie le type de la variable :

```
>>> type(listePair)
<class 'list'>
```

Attention, une affectation ne fonctionne que dans le sens **`variable = valeur`.**

2.2. Opérations basiques

Q.2. **D'après vous que vont afficher les lignes 8 à 11 ?**

```
1  x = 2000
2  a = x
3  x = x -10
4  c = x
5  x = 42
6  b = a*x
7
8  print(a)
9  print(b)
10 print(c)
11 print(x)
```

Je vous invite à vous renseigner sur la syntaxe des opérations dont vous auriez besoin de moment venu. On utilisera notamment `+,-,*,/,**,//,` et d'autres. Vous pouvez essayer.

En ce qui concerne l'informatique, **internet est extrêmement bien fourni** !

2.3. Les listes

Les listes sont une **structure de données** qui permet de **stocker plusieurs éléments** dans un seul objet, sous **forme ordonnée**. Les **éléments peuvent être de types** différents (nombres, chaînes, etc.). On peut ajouter, retirer ou modifier des éléments dans une liste. Les listes sont très flexibles et permettent de manipuler des ensembles de données de manière simple et efficace.

Pour accéder à un élément d'une liste donc le nom est `maListe`, on utilise la commande `maListe[indice]`.

Q.3. **Qu'affiche dans la console la ligne 2 ? Attention la numérotation Python commence à 0.**

```python
maListe = [1, 2, 3, "texte"]
print(maListe[1])
```

On utilisera notamment `maListe[ :2]`, `maListe[-1]`, `maListe.append(5)`, `len(maListe)`… Vous pouvez essayer.

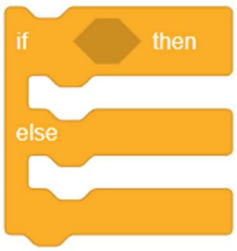
Je vous laisse regarder la syntaxe qui permet de manipuler aisément les listes et leurs éléments. Nous les verrons en ITC.

3. Les conditions et les boucles

Afin de former des scripts complexes - comprendre *utiles dans la majorité des cas* - il faut utiliser des conditions et des boucles sur nos variables, ou pour les faire évoluer.

3.1. Condition *if*

Une **condition `if`** (« si ») permet d'ajuster le comportement du code en fonction de la valeur d'une ou plusieurs variables. Une condition *if* peut contenir une **clause `else`** (« sinon ») qui correspond au code à exécuter si la condition est fausse.



Q.4. **Qu'affiche ce script dans la console ?**

```python
maVar = 12


if maVar == 12:
    maVar = maVar/2
    if maVar > 10:
        print('La valeur est supérieure à 10')
    else:
        print('La valeur est inférieur à 10')
```

Remarques :

- On remarque une indentation (tabulation) du code après le `if`, cela signifie que tout le code indenté fait partie du « bloc `if` ». **L'indentation est au cœur de la syntaxe Python.**
- Des blocs `if` peuvent **s'imbriquer** comme ci-dessus.
- Un `if` réalise un test, **ce n'est pas une affectation** (attention au double '=').
- Ce test se fait sur une **condition logique**, le résultat de cette condition est soit vrai (**True**) et le code continue dans le `if`, soit faux (**False**), dans ce cas le code continue dans le `else` s'il y en a un ou saute le bloc indenté `if` s'il n'y a pas de `else`.

3.2. Les boucles

Les boucles permettent de répéter automatiquement une série d'instructions plusieurs fois.

3.2.1. La boucle `for`

La boucle **`for`** est construire pour se répéter **un certain nombre de fois**.

Q.5. **Qu'affiche ce script dans la console ?**

```
1 maVar = 12
2
3 for k in range(5):
4     maVar = maVar - 1
5 print(maVar)
```

3.2.2. La boucle `while`

La boucle **`while`** est associée à une **condition**, les **instructions se répètent tant que cette condition est vraie**.

Q.6. **Qu'affiche ce script dans la console ?**

```
1 compteur = 0
2 while compteur < 5:
3     print("Ceci est la répétition numéro",
  compteur)
4     compteur += 1
```

4. Les fonctions

Les **fonctions sont des blocs de code réutilisable** qui permettent d'exécuter une tâche spécifique. Elles peuvent prendre des **arguments** (des données en entrée) et peuvent **retourner** une valeur. Les **fonctions permettent** de **structurer le code**, de le **rendre plus lisible** et de ne pas répéter plusieurs fois la même opération ; ce qui **évite les erreurs** et facilite l'évolution du script/programme.

Une fonction se définit avec le mot-clef « `def` », ensuite vient son nom puis des parenthèses. Dans celles-ci il peut y avoir des arguments. Après l'indentation, se trouve le code de la fonction. Celui-ci présente une ligne avec le mot-clef « `return` » qui précise ce que renvoie la fonction.

Pour utiliser une fonction, on l'appelle par son nom suivi des parenthèses avec les éventuels arguments.

Q.7. **Quelle est la valeur de la variable `resultat` ?**

```python
def ajouter(a, b):
    return a + b

resultat = ajouter(3, 4)
```

Remarques :

- Cette fonction est triviale mais on peut en imaginer des bien plus complexes.
- Des fonctions peuvent faire appel à d'autres fonctions.
- Il existe plein de fonction déjà implémentée dans Python.

Les fonctions ont leur propre espace mémoire, il faut se méfier des **variables locales** et des **variables globales**.

Q.8. **A votre avis que se passe t-il si j'exécute ce script ?**

```python
x = 10

def ma_fonction():
    x = 5
    print("Valeur de x à l'intérieur de la fonction:", x)

ma_fonction()
print("Valeur de x à l'extérieur de la fonction:", x)
```

Il existe deux variables *x*, une **locale qui ne vit que dans la fonction** `ma_fonction` et **une globale qui vit partout dans le script**. C'est une **mauvaise pratique** d'avoir des variables locales et globales avec **le même nom**. On remarque de Python favorise la variable locale.

5. Les modules (bibliothèques ou libraries)

Un **module (ou bibliothèque - library) est un fichier contenant** du code réutilisable, comme **des fonctions ou des variables**, que l'on peut **importer** dans **d'autres programmes**. Cela permet d'utiliser du code déjà écrit sans avoir à le réécrire.

Par exemple, Python possède un module standard appelé **`math`**, qui contient des fonctions mathématiques avancées, y compris la constante π (pi). Pour l'utiliser, on importe la bibliothèque en début de script puis on a accès à son contenu en utilisant son nom :

```python
import math

rayon = 5
aire = math.pi * rayon**2
print("L'aire d'un cercle de rayon 5 est :", aire)
```

Remarques :

- **On peut** aussi décider de n'**importer qu'une partie de la bibliothèque**, ici on aurait très bien pu remplacer la première ligne par : `from math import pi` qui n'importe que la valeur de pi. Dans ce cas il n'y a pas besoin d'ajouter `math.` devant `pi`.
- Il est fréquent d'utiliser un **alias**, par exemple *import math as mth*, ainsi la ligne 4 devient `Aire = mth.pi * rayon**2`
- On aurait pu faire une fonction `aire_cercle` qui prend en argument le rayon et renvoie l'aire.

Bilan

Façon d'importer	Effet	Usage
`from math import pi`	**Importe uniquement pi de la library math**	**pi**
`from math import *`	**Importe tout ce qui est contenu dans math**	**pi**
`import math`	**Importe tout ce qui est contenu dans math**	**math.pi**
`import math as mth`	**Importe tout ce qui est contenu dans math**	**mth.pi**

6. Tracer des courbes

Afin de tracer de tracer des courbes nous allons utiliser une bibliothèque très répandue, il s'agit de **`matplotlib.pyplot`** que nous allons **utiliser avec un alias pour gagner en lisibilité** (et en temps). Au début du code nous allons donc ajouter : `import matplotlib.pyplot as plt`.

Q.9. **Ouvrir le script Tracer_courbe.py avec Spyder et l'exécuter.**

Q.10. **Modifier ce code pour tracer la fonction cosinus sur l'intervalle 0 à 20.**

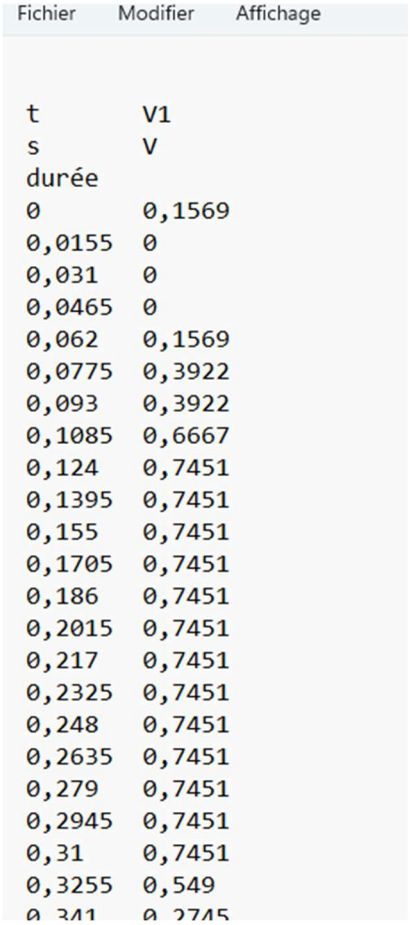
Remarques :

- La ligne 5 nous permet de créer une liste de `x`. Vous pouvez vous renseigner sur la fonction `linspace` de la bibliothèque `numpy`.
- Le **# annonce un commentaire**, il n'est pas pris en compte par le programme mais est utile pour la compréhension du code. **Un bon code est un code bien commenté !**
- Il est **important d'ajouter les titres et labels** pour comprendre à quoi correspond le tracer.
- **N'oublier pas la dernière ligne**, sans elle rien ne s'affiche.

7. Gestion d'un fichier texte ou csv

Un fichier **CSV (Comma-Separated Values) est** un format de **fichier texte** qui stocke des données **sous forme de tableau**, où chaque ligne représente une ligne du tableau, et les valeurs sont séparées par des **séparateurs** comme des virgules ou des points-virgules. Il est **couramment utilisé** pour échanger des données entre différents programmes et donc également pour **l'exportation de données expérimentales**.Nous allons utiliser le fichier texte `mesures_experimentales.txt` pour la suite de ce TP d'initiation. Il s'agit de données expérimentales de la pompe doseuse.

L'objectif est de tracer ces données sur Python.



Q.11. **Ouvrir le fichier `mesures_experimentales.txt` et voir à quoi il ressemble.**

Remarques :

- Le fichier est composé de deux colonnes et de 133 lignes.
- On distingue qu'il y a une première ligne vide suivie d'une ligne avec les noms des colonnes puis une troisième avec les unités. Enfin une quatrième ligne semble être une indication sur la grandeur mesurée.
- Le reste des lignes est composé de deux colonnes de chiffre avec des ',' séparée par une tabulation.

7.1. Ouverture et lecture avec Python

Nous allons maintenant l'ouvrir avec Python.

Pour cela il faut que le script Python soit dans le même dossier sur votre session que le fichier texte. **Attention à ce que ce soit toujours le cas pendant les TP.**

Q.12. **Ouvrir `ouverture_donnees_exp.py` et exécuter le script. Expliquer ce qu'il fait.** ***Si vous avez l'erreur 2, vous référez aux remarques de la partie 7.2.***

Indice :La commande `with open('mesures_experimentales.txt','r') as fichier:` permet de débuter un bloc de code dans lequel le fichier `mesures_experimentales` est ouvert et appelé `fichier`.

7.2. Modification d'un fichier

Le format des données expérimentales n'est pas forcément adapté à notre usage avec Python. Il nous faut souvent modifier les données pour pouvoir les traiter avec Python.

Par exemple pour ce fichier :

- Nous ne voulons que les chiffres dans les données, il faut alors enlever les premières lignes et la toute dernière.
- Nous avons vu que les virgules sont des séparateurs pour les listes et non des virgules pour des chiffres décimaux. Il faut donc remplacer les virgules par des points.

Pour cela nous avons plusieurs options :

- Soit en modifiant directement le fichier texte : supprimer les lignes en trop et changer les séparateurs **avec la fonction chercher-et-remplacer, en aucun cas à la main !** Malheureusement ce ne sera pas toujours possible, parfois il sera obligatoire d'opter pour la seconde option.
- Soit en modifiant le fichier avec Python. C'est ce que nous allons rapidement détailler.

Q.13. **Dans votre dossier, dupliquer (copier-coller) deux fois le fichier texte de sorte à avoir trois fichiers et toujours un dans l'état d'origine. *C'est une bonne pratique pour les TP.***

Q.14. **Ouvrir une des copies avec le bloc note et réaliser les modifications (suppression et la fonction chercher-et-remplacer).**

Q.15. **Ouvrir `modification_format_donnees_exp.py` et exécuter le script pour réaliser les modifications sur une autre copie du fichier.**

Remarques :

- Comme nous ouvrons le fichier en lecture ('`r`' pour `read`) et non en écriture, nous ne modifions pas durablement le fichier, seulement ce que nous sommes en train de lire.
- Vous pouvez rencontrer des erreurs. **Il faut toujours lire et essayer de comprendre ces erreurs : la machine essaie de communiquer avec vous !**
- Une erreur courante est que le fichier n'est pas trouvé. Cela peut venir de l'emplacement de votre script, de votre fichier, du nom du fichier,… et aussi du dossier de travail.
  Voici une check-list si vous rencontrez cette erreur :
  - Vérifier que le nom du fichier que vous voulez ouvrir est le bon.
  - Vérifier que le script (.py) est bien enregistré.
  - Vérifier que le script ET le fichier à ouvrir sont dans le même dossier.
  - Vérifier que ce dossier est bien sur votre session (pas dans l'espace partagé, par sur clef USB…)
  - Si l'erreur persiste, il va falloir débugger, pour cela, les lignes suivantes peuvent être utiles (attention à remplacer `monCheminVersLesFichiers`):

```
import os
# Dossier de travail actuel
print("Dossier de travail actuel :", os.getcwd())

# Définir le dossier de travail
os.chdir('monCheminVersLesFichiers')
```

7.3. Extraire les données vers des listes

Il ne nous reste plus qu'à extraire ces données vers des listes pour pouvoir les tracer.

Q.16. **Créer deux listes vides, `temps` et `voltage` avant d'ouvrir le fichier. La syntaxe est la suivante : `maListeVide=[]`**

7.3.1. Bien comprendre les fichiers csv

Certains caractères dans les fichiers sont codés. Les éditeurs de texte, comme la console Python, traduisent ces codes en caractères que nous pouvons lire. La fonction `repr()` de Python permet d'afficher la chaîne de caractère telle qu'elle est vraiment, sans interprétation de la console.

Q.17. **Ajouter la fonction `repr()` de sorte à avoir print(repr(lignePoint)).**

Q.18. **A partir de la documentation en fin de document (SI CCINP MP 2025), déterminer à quoi correspondent les séquences d'échappements présents dans le fichier csv.**

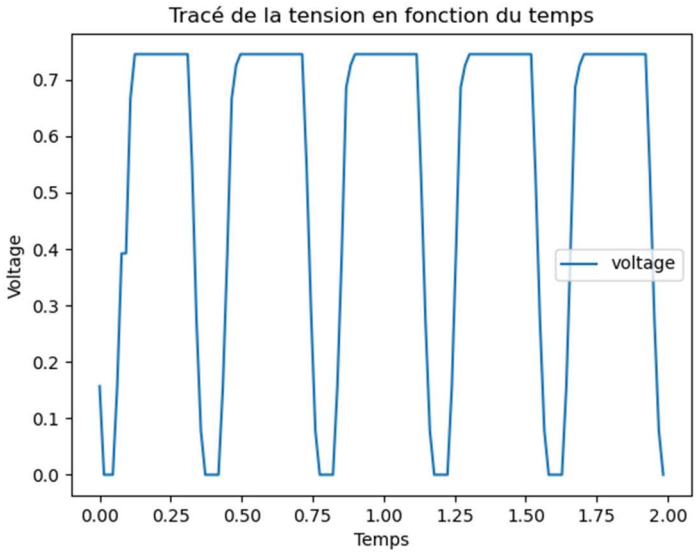
Q.19. **Dans un premier temps, utiliser la méthode (fonction) rstrip afin d'enlever la séquence en fin de ligne. *Comme toutes les méthodes elle s'utilise ainsi : lignePointSansRetour = lignePoint.rstrip(Arg).***

Q.20. **Ensuite, utiliser la méthode split pour récupérer les données t et v.**

Les données récupérées sont des chaines de caractères, nous voulons des flottants, il reste à forcer le type et à ajouter les valeurs aux listes.

Q.21. **Forcer le type et ajouter cette valeur à la liste des temps avec temps.append(float(t)).**

Q.22. **Tracer le voltage en fonction du temps.**



Ressources supplémentaires

Vous pouvez retrouver des cours, tutoriels et résumés en ligne :

- https://fr.wikibooks.org/wiki/Programmation_Python
- http://www.cpge-sii.com/informatique/bases-python/
- https://openclassrooms.com/fr/courses/7168871-apprenez-les-bases-du-langage-python?archived-source=235344
  - Et bien d'autres…

Documentation SI CCINP MP 2025

Une **séquence d'échappement** en Python est une combinaison de caractères commençant par un **antislash (\)**, utilisée pour représenter des caractères spéciaux ou effectuer des actions spécifiques dans une chaîne. Voici une liste des séquences d'échappement les plus courantes accompagnée d'exemples.

Séquence d'échappement	Représente	Exemple	Résultat
`\n`	Retour à la ligne	`text = "CCINP\n2025"`	`'CCINP` `2025'`
`\r`	Retour chariot	`text = "CCINP\r2025"`	`'2025'`
`\t`	Tabulation horizontale	`text = "CCINP\t2025"`	`'CCINP  2025'`
`\v`	Tabulation verticale	`text = "CCINP\v2025"`	`'CCINP` `2025'`

La fonction **`split()`** divise une chaîne en une liste. Vous pouvez spécifier le séparateur ; par défaut, le séparateur est un espace. La fonction **`split()`** prend éventuellement un paramètre :

- **separator(Optionnel)** : spécifie le séparateur à utiliser lors de division de la chaîne. Par défaut, le séparateur est un espace.

Exemple :

Script	Résultat
`text = "CCINP,2025,MP"` `print(text.split(","))`	`['CCINP', '2025', 'MP']`

La fonction **`rstrip()`** supprime tous les derniers caractères (l'espace est le dernier caractère par défaut à supprimer). La fonction **`rstrip()`** prend éventuellement un paramètre :

- **characters(Optionnel)** : une chaîne spécifiant le jeu de caractères à supprimer. Si l'argument characters n'est pas fourni, tous les espaces de la chaîne sont supprimés à la fin.

Exemple :

Script	Résultat

`text = "CCINP\t2025  "`
`print(text)`
`print(text.rstrip("\t"))`
	`'CCINP  2025  '`
	`'CCINP2025  '`

La fonction **`replace()`** remplace une chaîne de caractères spécifiée par une autre chaîne de caractères spécifiée. La fonction **`replace()`** prend deux paramètres :

- **oldStr(Obligatoire)** : la chaîne à rechercher
- **newStr(Obligatoire)** : la chaîne par laquelle remplacer l'ancienne chaîne

Exemple :

Script	Résultat

`text = "CCINP 2024 MP"`
`print(text)`
`print(text.replace("2024","2025"))`
	`'CCINP 2025 MP'`