Essentiels en python 1

Sabrine BENDIMERAD : sabrine.bendimerad1@gmail.com
Data science&AI Expert

2023/2024

Objectifs et Organisations

- Ce cours prépare à la certification PCEP – Certified Entry-Level Python Programmer
- Documents utilisés :
- Références :
  - ❑ Cours de la Cisco Networking Academy
  - ❑ Cours en ligne : https://openclassrooms.com/fr/courses/4302126-decouvrez-la-programmation-orientee-objet-avec-python et https://www.coursera.org/specializations/python
  - ❑ Programming Python, 4th Edition, Publisher(s): O'Reilly Media, Inc.

Déroulement de la séance

- Les exemples de code requièrent souvent des notions encore inconnues. Pas d'inquiétude si tout n'est pas clair à chaque fois !
- Evitez le mode «audience passive», n'hésitez pas à interagir avec moi et entre vous ☺.
- Suivez les exemples de code et testez sur vos environnements de travail !
- Il n'y a pas de question idiote, alors n'hésitez jamais !
- Ne soyez pas étonnés si je n'ai pas toujours les réponses à vos questions (Je regarderai et reviendrai vers vous)
- Vos retours/remarques/suggestions pour l'amélioration de ce cours sont les bienvenus en fin de séance.

Plan du cours

- **Module 1 :** Introduction
- **Module 2 :** Les types, les variables et les opérateurs basiques
- **Module 3 :** Les booléens, les conditions, les boucles et les listes
- **Module 4 :** Les fonctions, les tuples, les dictionnaires et les exceptions

MODULE 1

Module 1 : Les fondamentaux

- Un programme **informatique** est un ensemble d'opérations destinées à être exécutées par un ordinateur.
- Le programme est écrit par un développeur dans un langage de programmation compilé ou interprété

 Voir suite du cours, partie : langage Interprété vs compilé

Module 1 : Les fondamentaux

- D'une manière similaire à une langue naturelle, un langage de programmation à différents composants :
  - Un alphabet
  - Un dictionnaire/lexique
  - Une syntaxe (phrase correcte et compréhensible)
  - Une sémantique (phrase avec du sens)
- Un langage de programmation est :
  - Un pont entre le langage humain et le langage machine.
    - Le langage machine, ou code machine, est la suite de bits qui est interprétée par le processeur d'un ordinateur exécutant un programme informatique : langage bas niveau.
  - Un langage écrit par des humains et compréhensible par la machine pour exécuter des programmes (langage de programmation haut niveau).
  - Comme un langage normal, mais avec des signes.

Module 1 : Langage interprété vs compilé

- En langage compilé :
  - Le code source (celui que vous écrivez) est compilé, par un logiciel que l'on appelle compilateur, en un code binaire (langage machine) qu'un humain ne peut pas lire mais qui est très facile à lire pour un ordinateur
  - C'est le système d'exploitation qui va utiliser le code binaire et les données d'entrée pour calculer les données de sortie
  - La compilation se fait à chaque modification de code
- En langage interprété :
  - Le code source (celui que vous écrivez) est interprété, par un logiciel que l'on appelle interpréteur
  - L'interpréteur utilise le code source et les données d'entrée pour calculer les données de sortie
  - l'interpréteur va exécuter les lignes du code une par une, en décidant à chaque étape ce qu'il va faire ensuite. Processus « pas à pas »

 Voir suite pour comparaison

Module 1 : Langage interprété vs compilé

	Compilation	Interprétation
Avantages	•Exécution du code plus rapide •Compilateur uniquement pour le développeur •Sauvegarde en langage machine (sécurité)	•Possibilité d'exécuter le code directement après l'avoir écrit •Sauvegarde en langage de programmation, peut être exécuté sur n'importe quel plateforme
Inconvénients	•La compilation peut être longue •Nécessite un compilateur pour chaque plateforme	•Exécution moins rapide •Tout le monde doit avoir l'interpréteur pour exécuter le code

Module 1 : Qu’est-ce que python ?

- Un langage de programmation, très utilisé, orienté objet de haut niveau.
- Créé par **Guido Van Rossum** en 1991
- Baptisé « Python » en hommage à la troupe de comiques les « **Monty Python** »
- Intuitif, puissant, facile à apprendre et à enseigner
- Open source, n’importe qui peut contribuer à son développement
- Compréhensible en Anglais
- Adapté pour n’importe quelle tâche
- 20 après sa création… Top 10 du PYPL

 Les - : Pas le plus performant et peut être parmi les moins aisé à debugger

Module 1 : Utilité et Versions

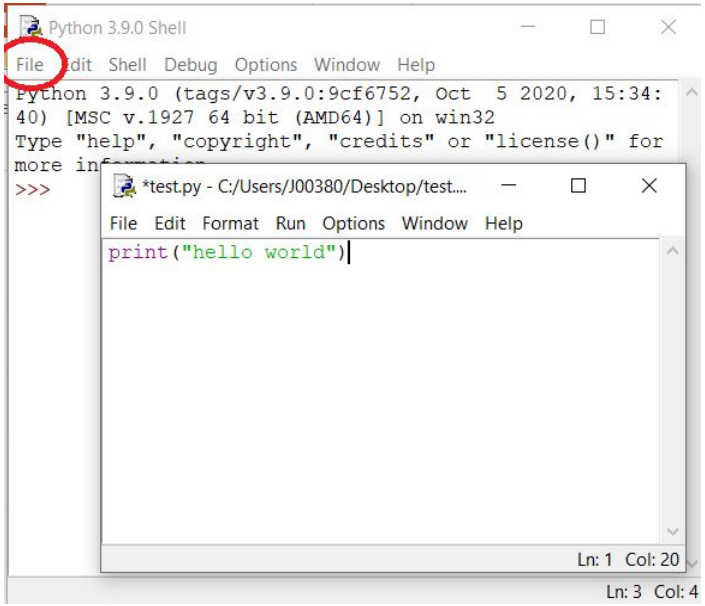
- Utilisé pour implémenter des **services web** complexes comme les moteurs de recherche, les outils de stockage cloud, les réseaux sociaux, etc. Il est aussi largement utilisé au sein de la communauté de Data Science.
- Peu utilisé en programmation bas niveau et en développement d'applications mobiles.
- La version 2 du langage n'est plus mise à jour mais reste utilisée. Python 3 est la version actuelle et celle utilisée dans ce cours.

Module 1 : Installation et Utilisation

- Linux : Installé par défaut. Ouvrir un terminal -> python3
- Windows et MacOs : Lien de téléchargement https://www.python.org/downloads/
- Pour commencer à travailler il faudra :
  - ❑ Un éditeur
  - ❑ Une console
  - ❑ Un débuggeur
- Pour commencer dans ce cours , privilégiez :
  - ❑ L'utilisation de l'IDLE : une application puissante pour écrire et exécuter du code
  - ❑ La sandbox : un environnement de travail vous permettant de tester du code python sans aucune installation sur le poste en local
    - ⇨ Faire le test sur la plateforme

Module 1 : Installation et Utilisation

- Ouvrir IDLE
- New file =>Save as nom_fichier (Sans extension)
- print(“hello world”)
- Run
- Test :
  - ❑ Supprimer une parenthèse + Run
  - ❑ Supprimer une lettre de print + RUN
  - ❑ Message d'erreur différent : La **nature** et le **moment** de découverte des erreurs sont différents.



MODULE 2

Module 2 : Premier programme (fonctions)

- Une fonction désigne un « sous-programme » permettant d'effectuer des opérations répétitives.
- Au lieu de réécrire le code, on l'encapsule dans une fonction que l'on appellera pour l'exécuter.
- Une fonction reçoit une ou plusieurs informations à partir desquelles elle retourne une ou plusieurs informations.
- Les informations fournies à la fonction sont appelées **arguments** ou **paramètres**. Les informations renvoyées sont appelées **résultats**.
- D'où viennent-elles :
  - ❑ Fonctions de base chargées au lancement de l'interpréteur Disponibles ici
  - ❑ Fonctions encapsulées dans des modules
  - ❑ Fonctions écrites par l'utilisateur
- Une fonction est appelée par son nom, suivi de parenthèse, contenant 0 à n arguments

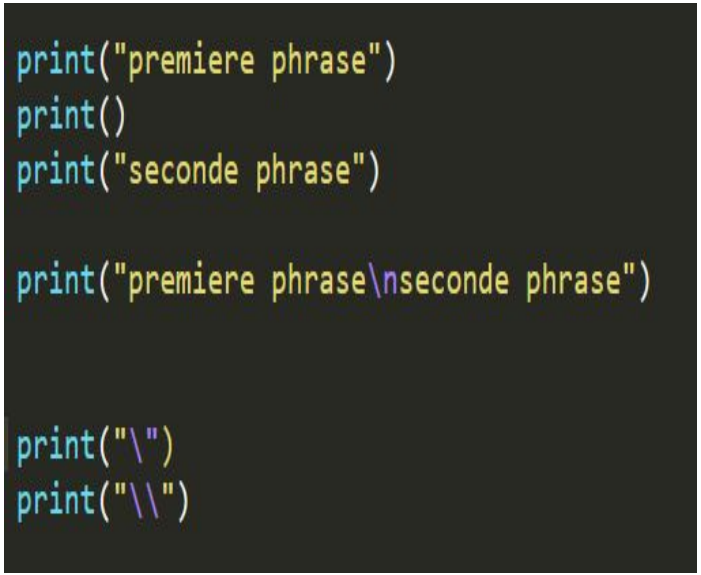
Module 2 : Premier programme (fonctions)

- Procédure d’exécution d’une fonction :

  - ❑ Python vérifie si le nom de la fonction est valide,
  - ❑ Ensuite si le nombre d’arguments entrés est celui attendu par la fonction,
  - ❑ Puis il passe les arguments à la fonction,
  - ❑ Enfin, il exécute le code encapsulé dans la fonction et renvoi le résultat

Module 2 : Premier programme (La fct print)

- La fonction **print** sert à afficher du texte ou à retourner une ligne vide si on l'invoque sans paramètres.
- Print invoquée avec un **\n** produit un saut à la ligne.
- Le \ est appelé escape character. Introduit dans un string, Il annonce que le caractère suivant à un rôle particulier. Ici le n pour newline.
- Cette convention à 2 conséquences :
  - ❑ Pour afficher un vrai \ , il est nécessaire d'utiliser \\
  - ❑ Tous les couples \ + caractère ne sont pas valides



Module 2 : Premier programme (La fct print)

- La fonction **print** peut être invoquée par 0 à n arguments, séparés par une virgule.
- Il existe 2 types d'arguments en python :
  - ❑ Positional arguments : Identifiés par leur position
  - ❑ Keyword arguments : Identifiés par leur nom
- En plus d'afficher les **positional arguments**, **print** contient 2 **keyword arguments :**
  - ❑ **end :** indique quoi afficher à la fin de print (Par défaut retour à la ligne)
  - ❑ **sep :** indique par quoi les arguments affichés sont séparés (Par défaut espace)
- Les keyword arguments doivent s'écrire après les positional arguments

```
print("The itsy bitsy spider" , "climbed up")

print("My name is", "Python.")
print("Monty Python.")
print("My name is", "Python.", end=" ")
print("Monty Python.")


print("My", "name", "is", "Monty", "Python.", sep="-")
print("My", "name", "is", sep="_", end="*")
print("Monty", "Python.", sep="*", end="*\n")
```

Module 2 : Les types (Integer et Float)

- 2 Types numériques sont le plus utilisés en python :
  - **integer** : nombre sans virgule.
  - **float** : nombres à virgule.

 Remarque : La forme de l'input détermine le type dans lequel python va stoker en mémoire

- Quelques conventions sur le type integer :
  - On peut écrire : 11111111 ou 11_111_111, -11111111 ou -11_111_111.
  - Précédé par 0O or 0o préfixe (zero-o), un entier est exprimé en octal (base 8). Par 0x ou 0X, il est exprimé en Hexadécimal (base 16).
- Quelques conventions sur le type float :
  - La virgule est remplacée par un point
  - 4 est un integer, 4.0 est un float
  - On peut écrire 4.0 de cette façon : 4. et 0.4 de celle-ci : .4

Module 2 : Les types (Integer et Float)

- Pour les très grands nombres on peut utiliser la notation scientifique e : 300000000 peut s'écrire en $3 * 10^8$, 3E8 ou 3e8.
- l'exposant (valeur après le E) et la base (valeur avant le E) doivent être de type integer.
- Python choisit parfois des notations différentes que celles de l'utilisateur print(0.000000000000000000001)affiche 1e-22.
- Pour les très petits nombres (Valeur absolue proche de 0), il faudra tiliser 6.62607e-34.

Module 2 : Les types (String)

- Le type String (chaine de caractères) permet de stocker du texte.
- Pour afficher le message I like "Monty Python" :

  - ❑ Précéder les guillemets par un backslash \, ceci changera leur rôle habituel qui consiste à délimiter le début et la fin de print.
  - ❑ Ou alors, délimiter la chaîne de caractères par des apostrophes et utiliser les guillemets pour l'intérieur du texte.

```
print("I like \"Monty Python\"")
print('I like "Monty Python"')
```

Module 2 : Les opérateurs

- Python est utilisé comme calculateur grâce aux opérateurs.
- Les opérateurs sont des symboles permettant de faire des opérations mathématiques.
- Liste des opérateurs : **+, -, *, /, //, %, ****
  - ❑ **** : Pour l'exponentiel
  - ❑ */* : Pour la division. Le résultat de la division est toujours un float
  - ❑ *//* : Pour la division entière. Le résultat toujours entier, arrondi à la valeur inférieure.
  - ❑ **% :** Appelé Modulo , sert à retourner le reste de la division.

```
print(2 ** 3)
print(2 ** 3.)
print(2. ** 3)
print(2. ** 3.)

print(2 * 3)
print(2 * 3.)
print(2. * 3)
print(2. * 3.)

print(6 / 3)
print(6 / 3.)
print(6. / 3)
print(6. / 3.)
```

```
print(6 // 3)
print(6 // 3.)
print(6. // 3)
print(6. // 3.)


print(6 // 4)
print(6. // 4)
print(-6 // 4)
print(6. // -4)

print(14 % 4)
```

Module 2 : Les opérateurs

- Les opérateurs suivent un ordre de priorité, comme les opérations mathématiques.

Priorité	Opérateur	
1	Parenthèses	
2	+, -	unary
3	**	
4	*, /, %	
5	+, -	binary

- Sinon, priorité de gauche à droite
- Exception : Exposant , priorité de droite à gauche (Voir exemple à gauche en rouge)

```
print(2 + 3 * 5)
print(9 % 6 % 2)
print(2 ** 2 ** 3)
print(2 * 3 % 5)
print((5 * ((2 % 3) + 10) / (2 * 5)) // 2)
```

print(2**2**3)
revient à calculer:
2 ** 3 → 8; 2 ** 8 → 256

Module 2 : Les variables

- Une variable est un concept basique en programmation commun à tous les langages.

```
var = 1
print(var)
```

- Une variable garde en mémoire une information, elle est formée d'un nom et d'une valeur.
- Le nom d'une variable:
  - ❑ Peut être composé de : majuscules, minuscules, chiffres et du caractère _ (underscore).
  - ❑ Doit commencer par une lettre.
  - ❑ Est sensible à la casse (Alice et ALICE, sont 2 variables différentes).
  - ❑ Ne doit pas avoir le même nom que les keywords python.
  - ❑ Les bonnes pratiques de nommage se trouve dans le guide PEP 8 -- Style Guide for Python Code .

Module 2 : Les variables

- Une variable à un nom et une valeur assignée.
- Le contenu d'une variable peut être affiché avec la fonction print.
- Plusieurs variables peuvent être affichés en même temps.
- La fonction print offre la possibilité de concaténer du texte en utilisant l'opérateur **+**.
- On peut assigner une nouvelle valeur à une variable existante.

```
var = 1
print(var)


account_balance = 1000.0
client_name = 'John Doe'
print(var, account_balance, client_name)

var_s = "3.7.1"
print("Python version: " + var_s)

var = var + 1
print(var)
```

Module 2 : Les variables

- Python offre une manière rapide d'écrire un certain type d'opérations de calcul :

```
x = x * 2
x *= 2
```

- Convention :

variable = variable op expression

variable op= expression

- Quelques exemples :

```
i = i + 2 * j ⇒⇒ i += 2 * j

var = var / 2 ⇒⇒ var /= 2

rem = rem % 10 ⇒⇒ rem %= 10

j = j - (i + var + rem) ⇒⇒ j -= (i + var + rem)

x = x ** 2 ⇒ x **= 2
```

Module 2 : Les commentaires

- Les commentaires apportent de l'information au code et sont destinés à l'utilisateur. Ils ne sont pas exécutés par python.
- Un commentaire commence par #. Pour commenter plusieurs lignes :

  Les sélectionner, ensuite : **CTRL+/** (Windows), **CMD+/** (Mac OS)
- Un développeur responsable décrit son code : noms de variables, rôles des variables, rôles des fonctions, etc. (Un nom de variable parlant "self-commenting" est suffisant et n'a pas besoin de commentaire)

```
# This program evaluates the hypotenuse c.
# a and b are the lengths of the legs.
a = 3.0
b = 4.0
c = (a ** 2 + b ** 2) ** 0.5  # We use ** instead of square root.
print("c =", c)
x = x * 2
x *= 2
#z=5
#print(z)
```

Module 2 : La fonction input()

- La fonction input permet à l'utilisateur d'entrer une donnée/valeur.
- Elle peut être invoquée sans argument ou avec un argument de type string.
- La valeur renvoyée par input doit être enregistré pour ne pas être perdue.
- Le résultat de input est de type string.
- Le dernier bout de code à droite doit générer un erreur car on essaye de multiplier un string par u float.

```
anything = input()
print("Hmm...", anything, "... Really?")

anything = input("Tell me anything...")
print("Hmm...", anything, "...Really?")

##Erreur
anything = input("Enter a number: ")
something = anything ** 2.0
print(anything, "to the power of 2 is", something)
```

- Pour y remédier, utiliser le casting, voir slide suivante

Module 2 : La fonction input()

- Le casting est la conversion forcée du type d'une valeur.
- Python offre 3 fonctions pour convertir le type :
  - ❑ int() : pour convertir une valeur en integer
  - ❑ float() : pour convertir une valeur en float
  - ❑ str() : pour convertir une valeur en string
- Dans l'exemple, float introduit avant input permet de convertir le résultat de input afin de l'utiliser dans l'opération mathématique suivante.

```
anything = float(input("Enter a number: "))
something = anything ** 2.0
print(anything, "to the power of 2 is", something)
```

Module 2 : Les opérateurs sur le type string

- Les opérateurs + et * sont aussi utilisés pour des calculs mathématiques mais aussi pour travailler sur le type string :
  - ❑ Concaténation :

    string + string
  - ❑ Réplication :

    string * number

    number*string
- Astuce : Dans le dernier exemple, le résultat hyp a été transformé en string avec la fonction str pour qu'on puisse appliquer la concaténation dans print.

```
print("ab"+"ba")

print("James" * 3)
print(3 * "an")
print(5 * "2")


leg_a = float(input("Input first leg length: "))
leg_b = float(input("Input second leg length: "))

hyp = (leg_a**2 + leg_b**2) ** .5
print("Hypotenuse length is " + str(hyp))
```

MODULE 3

Module 3 : Les opérateurs de comparaison

- En python on peut comparer 2 valeurs via des opérateurs de comparaison :

  == (égale), >, >=, <, >=, != (différent)

- True et False sont les deux réponses possibles à une opération de comparaison.
- True et False sont de type **Booléen** (Oui ! Encore un nouveau type en python).
- La réponse d'une opération de comparaison peut être enregistrée dans une variable.
- Les opérateurs de comparaison sont très utiles pour tester des conditions.

 Explication slide suivante ☺

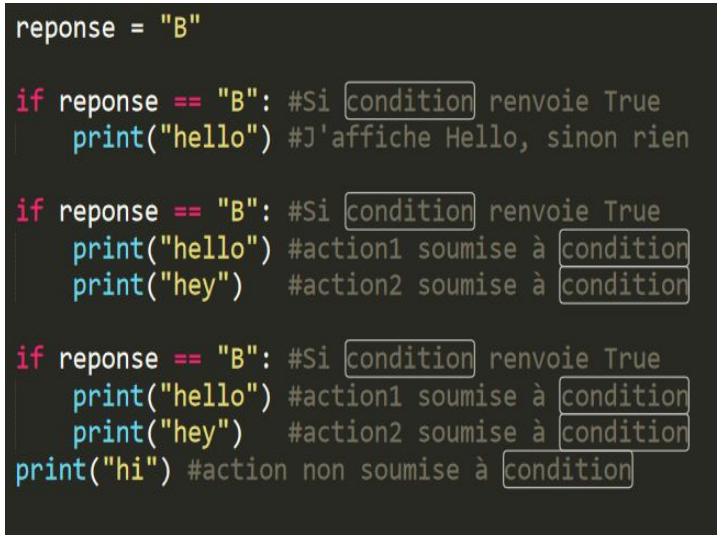
```
2 == 2
2 == 2.
1 == 2

var=2
var == 0

1 > 2
1 >= 2
1 != 2
answer = 5 != 2
```

Module 3 : Les conditions

- Les conditions ont le même sens en informatique que dans le langage courant.
- Si (événement), dans ce cas (action). Exemple : si j'ai 4 euros, je m'achète le dernier Picsou Magazine. Sinon, il faut que j'aille retirer des sous.
- Pour écrire une condition en python :
  - ❑ On commence par if, puis on indique la condition à remplir et on termine la ligne par deux points.
  - ❑ On indique ensuite une/plusieurs actions à effectuer. Afin de différencier du reste de votre programme, on ajoute 4 espaces au-début de la ligne.
  - ⇨ Nous appelons cela l'indentation et Python est très strict sur le sujet !



Module 3 : Les conditions

- Comment définir ce qui se passe si la condition n'est pas remplie ? Il suffit d'utiliser else
- En plus de if et else, on peut proposer d'autres choix grâce à elif (sinon si, else+if).
- Exemple : si j'ai 4 euros, je m'achète le dernier Picsou Magazine. Sinon si j'ai 8 euros, je m'achète 2 Picsou Magasines, Sinon je vais retirer des sous.
- Les règles d'indentation de if et else s'appliquent aussi sur elif.
- Remarques :
  - La branche else est toujours la dernière, qu'on utilise elif ou pas.
  - La branche else est optionnelle.

```
if reponse == "B": #Si condition renvoie True
    print("hello") #J'affiche Hello
else:              #Si condition renvoie False
    print("hi")    #J'affiche hi


if reponse == "B":
    print("hello")
elif reponse == "C":
    print("hi")
elif reponse == "D":
    print("hey")
else:
    print("end")
```

Module 3 : Les boucles (While)

- Une boucle (loop en anglais) est une action qui se répète automatiquement un certain nombre de fois.
- Python offre 2 manières principales pour créer des boucles. La première est la boucle while :
  - La syntaxe de while ressemble à if
  - Dans if, le programme exécute l'action si la condition est True. Dans while, Le programme répètera l'action tant que la condition est True.
- Les actions sont appelées : corps de la boucle.
- Si la condition est False (égale à zéro) la première fois, le corps de la boucle ne sera pas exécuté.

```
user_answer = "A"

while user_answer == "A":
    print("Hello")
```

Attention !!!

Boucle infinie Pour terminer :

ctrl+c(KeyboardInterrupt)

While doit être contrôlée pour éviter la boucle infinie

Module 3 : Les boucles (While)

- Décortiquons ensemble ce code qui calcule le nombre de chiffres pairs/impairs que l'utilisateur introduit !!!

Astuces pour Simplification :

while number != 0: => while number:

if number % 2 == 1: => if number % 2:

```
odd_numbers = 0
even_numbers = 0

# read the first number
number = int(input("Enter a number or type 0 to stop: "))

# 0 terminates execution
while number != 0:
    # check if the number is odd
    if number % 2 == 1:
        # increase the odd_numbers counter
        odd_numbers += 1
    else:
        # increase the even_numbers counter
        even_numbers += 1
    # read the next number
    number = int(input("Enter a number or type 0 to stop: "))

# print results
print("Odd numbers count:", odd_numbers)
print("Even numbers count:", even_numbers)
```

Module 3 : Les boucles (While)

- Décortiquons ensemble ce code qui utilise la notion de counter (compteur), souvent utilisée en programmation !!!

```
#Version 1
counter = 5
while counter != 0:
    print("Inside the loop.", counter)
    counter -= 1
print("Outside the loop.", counter)

#Version simplifiée
counter = 5
while counter:
    print("Inside the loop.", counter)
    counter -= 1
print("Outside the loop.", counter)
```

Module 3 : Les boucles (For)

- La boucle while n'est pas utile quand il n'y a pas de condition. La boucle for est plus adaptée.
- for est spécialisée dans le parcours d'une séquence de plusieurs données (Une chaine de caractère est une séquence, nous verrons d'autres type de séquence plus tard).
- Exemple : chaine est une séquence de plusieurs lettres. L'élément lettre prend successivement la valeur de chaque lettre contenue dans la variable chaine. On affiche ces valeurs avec print().
- Les règles d'indentation s'appliquent aussi sur la boucle for.

Syntaxe de la boucle for

```
for element in sequence:
    action1
```

```
chaine = "Bonjour les ZER0S"
for lettre in chaine:
    print(lettre)
```

Module 3 : Les boucles (For)

- La boucle for s'utilise souvent avec range.
- range est un itérateur qui renvoi un objet itérable, c'est-à-dire un objet qu'on peut parcourir avec une boucle for (Exactement comme les chaines de caractères). Pour comprendre, testez !!!
- L'intuition derrière range() :
  - Quand j'écris : `for i in range(5)`
  - Je veux dire en langage humain : `Pour chaque élément dans[0,1,2,3,4]`

 La réalité derrière range() est complexe et ne sera pas abordée dans ce cours. Cela dit, nous recroiseront range pour travailler sur les listes.

```
for i in range(10):
    print("The value of i is currently", i)


for i in range(2, 8):
    print("The value of i is currently", i)


for i in range(2, 8, 3):
    print("The value of i is currently", i)
```

Testez avec : range(1,1) et avec range(4,1) !!!

Module 3 : Les boucles (Le mot clé break)

- Le mot-clé break permet tout simplement d'interrompre une boucle.

- Exemple : La boucle while a pour condition 1, une condition qui sera toujours vraie. C'est une boucle infinie. On demande à l'utilisateur de taper la lettre 'Q' pour quitter. Quand il tape 'Q', le programme affiche **Fin de la boucle** et la boucle s'arrête grâce au mot-clé break. Testez le code sans le mot clé break, puis avec break !!!

- break permet d'arrêter une boucle quelle que soit sa condition. Python sort immédiatement de la boucle et exécute le suite du code, s'il y en a.

Il est possible de faire autrement on mettant la condition lettre != Q dans le while

Module 3 : Les boucles (Le mot clé continue)

- Le mot-clé continue permet de continuer une boucle, en repartant directement à la ligne du while ou du for.
- Exemple : tous les trois tours de boucle, i s'incrémente de 4. Arrivé au mot-clé continue, Python n'exécute pas la fin du bloc mais revient au début de la boucle en testant à nouveau la condition du while.
- Autrement dit, quand Python arrive à la ligne continue, il saute à la ligne while sans exécuter les 2 dernières lignes de code.

```
for i in range(1, 6):
    if i == 3:
        continue
    print("Inside the loop.", i)
print("Outside the loop.")
```

Module 3 : Les boucles (break et continue)

- Décortiquons ensemble ce code qui utilise les mots clés break et continue avec for !!!

```
# break - example

print("The break instruction:")
for i in range(1, 6):
    if i == 3:
        break
    print("Inside the loop.", i)
print("Outside the loop.")


# continue - example

print("\nThe continue instruction:")
for i in range(1, 6):
    if i == 3:
        continue
    print("Inside the loop.", i)
print("Outside the loop.")
```

Module 3 : Les boucles (Avec else)

- Les boucles while et for peuvent être utilisées avec la branche else (Pas très utile mais possible).
- La branche else est exécutée toujours une seule fois, après que le corps de la boucle termine son exécution, et si la boucle ne se termine pas avec le mot break.

```
i = 1
while i < 5:
    print(i)
    i += 1
else:
    print("else:", i)


for i in range(5):
    print(i)
else:
    print("else:", i)
```

Module 3 : Les opérateurs de logique

- Il arrive souvent que nos conditions doivent tester plusieurs prédicats en même temps.

- Pour cela , il existe le mot clé and.
- Exemple 1 : On cherche à tester à la fois si a est supérieur ou égal à 2 et inférieur ou égal à 8.

- Sur le même mode, il existe le mot clé or.
- Exemple 2 : On cherche à savoir si a n'est pas dans l'intervalle. L'évaluation est égale à True, si la variable a est inférieure à 2 ou supérieure à 8.

- Enfin, il existe le mot clé not qui « inverse » un prédicat. Le prédicat not a==5 équivaut donc à a!=5.

```
a = 5
b= 10

if a>=2 and b<=8:
    print("a et b respectent la condition")
else:
    print("condition non respectée")



if a>0 or b>0:
    print("a ou b ou les 2 sont valide")
else:
    print("condition non respectée")
```

Module 3 : Les listes

- En Python, les listes sont des objets qui peuvent en contenir d'autres.
- On peut créer des listes vides ou non vides.
- Une liste peut contenir des objets de différents types.
- Chaque objet est un élément de la liste et à un index. L'indexation dans python commence à 0.
- On peut accéder à un élément particulier de la liste en l'appelant par son index.
- On peut assigner de nouvelles valeurs à chaque élément de la liste.

```python
#On crée une liste vide V1
ma_liste = list()

#On crée une liste vide V2
ma_liste = []

#Liste avec types différents
ma_liste = [1, 3.5, "une chaine", []]

# On crée une liste avec des entiers
ma_liste = [1, 2, 3, 4, 5]
print(ma_liste)

#On accede au 1er element de la iste
print(ma_liste[0])

#Assigner 111 au 1er element de la liste
ma_liste[0] = 111
print(ma_liste)

#Assigner la valeur de l'element 5 à 2
ma_liste[1] = ma_liste[4]
print(ma_liste)
```

Module 3 : Les listes (Fonctions/Méthodes)

- La fonction len permet de connaitre combien d' éléments contient une liste.
- L'instruction del permet de supprimer un élément de la liste (del est une Instruction et non une fonction).
- Les indexes négatifs sont permis dans les listes.
- On peut accéder au dernier élément d'une liste avec l'indexe -1.
- L'indexation depuis la fin de la liste commence à partir de -1.

```
len(ma_liste)
del ma_liste[1]

numbers = [111, 7, 2, 1]
print(numbers[-1])
print(numbers[-2])
```

Module 3 : Les listes (Fonctions/Méthodes)

- Une méthode est un type de fonction invoquée directement sur un objet.
- Les méthodes de liste modifient l'objet mais ne revoient rien (D'autres méthodes comme celles des string agissent différemment).
- L'invocation d'une méthode se fait de cette manière : objet.method(arg)
- Les méthodes append sert à ajouter un élément à la fin de la liste.
- La méthode insert sert à ajouter un élément dans un index particulier.

 La notion de méthode est abstraite pour l'instant, elle ne sera pas étudiée dans ce cours.

```
numbers = [111, 7, 2, 1]
###
numbers.append(4)
numbers2 = numbers.append(-15)
print(numbers) # modifiée
print(numbers2) # Rien ! Retourne None
###
numbers.insert(0, 222)
print(len(numbers))
print(numbers)
```

Module 3 : Les listes (Itérations)

- Il est possible de créer une liste vide et de l'alimenter en iterant avec for.
- La boucle for offre différentes manières d'itérer des listes :
  - Itération avec range :
    - range(4) OU range (0,4) : permet 4 itérations (0,1,2,3)
    - range(2,5) : permet 3 itérations (2,3,4)
  - Itération avec range + len() : utile quand on ne connait pas le nombre d'éléments d'une liste
- Une autre manière plus simple d'itérer est d'utiliser i comme itérateur sur les valeurs d'une liste (Force de python).

```
#Liste vide
myList = []

#range()
for i in range(5):
    myList.append(i + 1)
print(myList)

#len()
myList = [10, 1, 8, 3, 5]
total = 0
for i in range(len(myList)):
    total += myList[i]
print(total)

#i comme copie de valeurs
for i in myList:
    total += i
print(total)
```

Module 3 : Les listes (Permutations)

- Il est souvent demandé en programmation de permuter 2 valeurs.
- Exemple 1 : Avec ce type de permutation, on perds la valeur de la variable1.
- La bonne pratique en programmation est d'utiliser une variable intermédiaire.
- Python offre une manière simple de faire des permutations, en permutant plusieurs variables à la fois sans passer par la variable intermédiaire.
- Comment permutez à vore avis quand on a 100 éléments dans une liste ?

 Réflexion sur la slide suivante ☺

```
#Exemple 1
variable1 = 1
variable2 = 2
variable2 = variable1
variable1 = variable2

#Variable intermediaure
variable1 = 1
variable2 = 2
auxiliary = variable1
variable1 = variable2
variable2 = auxiliary

#A la manière python
variable1 = 1
variable2 = 2
variable1, variable2 = variable2, variable1
```

Module 3 : Les listes (Permutations)

```python
#Permutation simple avec liste
myList = [10, 1, 8, 3, 5]

myList[0], myList[4] = myList[4], myList[0]
myList[1], myList[3] = myList[3], myList[1]

print(myList)

#Permutation complexe avec liste
myList = [10, 1, 8, 3, 5]
length = len(myList)

for i in range(length // 2):
    myList[i], myList[length - i - 1] = myList[length - i - 1], myList[i]

print(myList)
```

Module 3 : Algorithmes de tri(Tri à bulles)

- Trier une suite de 10 nombres n'est pas très long. Mais qu'en est-il lorsque nous avons 100 000 nombres à trier ? On fait appel aux algorithmes de tri.
- Les algorithmes de tri sont l'essence même de l'algorithmique et nous permettent de réorganiser nos données. Plusieurs algorithmes de tri existent !!! Nous nous intéresserons dans ce cours au « Bubble sort algorithm ».
- Pourquoi le bubble sort algorithm :
  - ❑ Facile à comprendre
  - ❑ Utile pour trier des listes (Reste quand même très consommateur en ressources)

- Pourquoi ce nom 'tri à bulles' ? Lors du tri, la plus grande donnée remonte peu à peu à la fin de la liste, comme des bulles de savon.

Module 3 : Algorithmes de tri(Tri à bulles)

- Cet algorithme avance dans une liste, compare les éléments 2 à 2 et les échange si :

  valeur1 > valeur2

- Il compare toutes les paires d'éléments puis recommence jusqu'à ce que toutes les paires soient dans le bon ordre.
- Essayons de trier cette liste de nombre [8,10,6,2,4] :

```
#bubble sort en pseudo code
Je recommence jusqua trier ma liste :
    pour i de 1 à taille(ma_liste -1)
        si T[i + 1] < T[i]
            echanger_(T[i + 1], T[i])
```

- Prochain slide, bubble sort en python !

Premiere itération

8	10	6	2	4
8	6	10	2	4
8	6	2	10	4
8	6	2	4	10

Seconde itération

8	6	2	4	10
6	8	2	4	10
6	2	8	4	10
6	2	4	8	10

Dernière itération

6	2	4	8	10
2	6	4	8	10
2	4	6	8	10

Module 3 : Algorithmes de tri(Tri à bulles)

- Décortiquons ensemble ce code qui traduit l'algorithme bubble sort en python !!!

Attention au swapped (valeur de contrôle) et à son utilité !!!

```python
#bubble sort en python
myList = [8, 10, 6, 2, 4] # list to sort
swapped = True # a little fake, need it to enter the while loop

while swapped:
    swapped = False # no swaps
    for i in range(len(myList) - 1):
        if myList[i] > myList[i + 1]:
            swapped = True # swap occured!
            myList[i], myList[i + 1] = myList[i + 1], myList[i]

print(myList)
```

Module 3 : Algorithmes de tri(Tri à bulles)

- Un code de tri à bulles interactif. Ce code nous permet d'entrer le nombre d'élément de la liste qu'on souhaite construire, de l'alimenter et ensuite de la trier.

```python
#bubble sort interactif en python
myList = []
swapped = True
num = int(input("How many elements do you want to sort: "))

for i in range(num):
    val = float(input("Enter a list element: "))
    myList.append(val)

while swapped:
    swapped = False
    for i in range(len(myList) - 1):
        if myList[i] > myList[i + 1]:
            swapped = True
            myList[i], myList[i + 1] = myList[i + 1], myList[i]

print("Sorted:")
print(myList)
```

Module 3 : Les listes

- Exemple : Nous créons une liste dans la variable ma_liste1. À la ligne 2, nous affectons ma_liste1 à la variable ma_liste2. Quand on ajoute 4 à ma_liste2, ma_liste1 est aussi modifiée.
- ma_liste1 et ma_liste2 contiennent une référence vers le même objet : si on modifie l'objet depuis l'une deux le changement sera visible sur les deux.
- Pour y remédier : Il faudra créer un nouvel objet depuis un autre en utilisant le slice **[:]** : Une expression du langage python pour extraire des elements d'une liste ou d'une chaine.

```
ma_liste1 = [1, 2, 3]
ma_liste2 = ma_liste1
ma_liste2.append(4)
print(ma_liste2)
print(ma_liste1)

#Copier toute la liste
ma_liste1 = [1, 2, 3]
ma_liste2 = ma_liste1[:]
ma_liste2.append(4)
print(ma_liste2)
print(ma_liste1)

#Copier une partie de la listee
ma_liste1 = [10, 8, 6, 4, 2]
ma_liste2 = ma_liste1[1:3]
print(ma_liste1)
print(ma_liste2)
```

Attention : [1:3] prend les éléments des indexs 1 et 2
[debut:fin] : prend les éléments debut jusqu'à fin-1

Module 3 : Les listes

```python
#On peut omettre le 0 au debut du slice
myList = [10, 8, 6, 4, 2]
newList = myList[0:3]
newList = myList[:3]
print(newList)

#On peut omettre la fin du slice
myList[start:]
myList[start:len(myList)]

myList = [10, 8, 6, 4, 2]
newList = myList[3:]
newList = myList[3:len(myList)]
print(newList)
```

Quelques spécificités du slice

```python
#Del sur plusieurs elements
myList = [10, 8, 6, 4, 2]
del myList[1:3]
print(myList)

#Del sur tous les elements
myList = [10, 8, 6, 4, 2]
del myList[:]
print(myList)

#Les operateurs in et not in
myList = [0, 3, 12, 8, 2]
print(5 in myList)
print(5 not in myList)
print(12 in myList)
```

La méthode del et les opérateurs in et not in sur les listes

Module 3 : Les listes (list comprehension)

- Les compréhensions de liste sont un moyen très simple pour modifier une liste. La syntaxe est déconcertante au début mais on s'y habitue ;).
- Lignes de code à droite signifient : « Mettre au carré tous les nombres contenus dans la liste d'origine »
- Quand on utilise une list comprehensions, le résultat est sotcké dans une liste directement.
- Convention d'écriture : valeur de retour suivi de la boucle for.
- Il est également possible d'ajouter des conditions aux list comprehensions.

```python
liste_origine = [0, 1, 2, 3, 4, 5]

#Boucle for classique
resultat=[]
for nb in liste_origine :
    resultat.append(nb**2)
print(resultat)

#List comprehesion
resultat = [nb**2 for nb in liste_origine]
print(resultat)

#List comprehension avec condition
liste_origine = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
resultat = [nb for nb in liste_origine if nb % 2 == 0]
print(resultat)
```

Testez autant de fois que nécessaire pour comprendre l'intuition derrière !!!

Module 3 : Les listes (list comprehension)

- Une liste imbriquée est une liste dont les éléments sont eux-mêmes des listes.
- Exemple : L contient une référence vers une liste à 2 éléments (Liste à 2 dimensions). ces derniers étant tous deux des listes de nombres entiers à trois éléments.
- On peut comparer une telle structure à une matrice caractérisée par un nombre de lignes et de colonnes :$L = \begin{matrix} 1 & 2 & 4 \\ 3 & 4 & 5 \end{matrix}$
- On accède à chaque élément de la liste avec la notation L[i][j] où l'indice i représente celui de la ligne et l'indice j celui de la colonne.
- Il est aussi possible/utile d'utiliser les list compréhensions pour les listes imbriquées

```
L = [[1, 2,4], [3, 4, 5]]
len(L)
L[0][1]

#Afficher les elements avec for
for i in L:
    for j in i:
        print(j)

#Afficher les elements avec list comprehensions
resultat = [[j for j in i] for i in L]
```

MODULE 4

Module 4 : Les fonctions

- Nous avons déjà utilisé dans les modules précédents certaines fonctions:
  - ❑ print(): pour afficher du texte
  - ❑ input(): lire une valeur
- Il est possible d'écrire nos propres fonctions. Mais quand ? À quel moment ?
- Quand un fragment de code commence à apparaître plusieurs fois à différents endroits, on commence à réfléchir à la possibilité de l'isoler dans une fonction pour factoriser le code.
- Une fonction bien écrite doit être visualisée d'un seul coup d'oeil, il faut parfois découper une fonction trop longue en plusieurs fonctions dans le but que le code soit plus lisible, plus simple et plus facile à tester.

Module 4 : Les fonctions

- D'où viennent les fonctions (Rappel : Vu dans le module 1) :

  - ❑ Fonctions de base chargées au lancement de l'interpréteur (built-in functions) Disponibles ici
  - ❑ Fonctions encapsulées dans des modules
  - ❑ Fonctions écrites par l'utilisateur
  - ❑ Fonctions lambda (Ce sont des fonctions extrêmement puissantes et utiles spécifiques à python qu'on ne verra dans ce cours)

- Syntaxe d'une fonction :

```
# Syntaxe d'une fonction
def nom_de_la_fonction(paramètres):
    instructions
```

Attention !!! les noms de fonctions répondent à des règles de nommage, pas de mots clés réservés au langage et aucun caractère spécial ou accentué

Module 4 : Les fonctions

- Imaginons qu'on vous demande de modifier votre phrase dans print ! Vous serez amener à le faire 3 fois !!!!
  - ❑ Sauf si vous créez une fonction ☺ Regardons comment cela fonctionne !
- Décortiquons le code :
  - ❑ Création de la fonction avec le mot clé def, le nom « message », et l'instruction « print ». Cette fonction n'a pas de paramètres.
  - ❑ Invocation de la fonction par son nom suivi de parenthèses.
  - ❑ Attention la fonction doit être définit avant son invocation. Sinon nameError !!!
- Si à présent on souhaite modifier le message de print, on le fait qu'une seule fois dans la fonction.

```
print("Enter a value :")
a = int(input())
print("Enter a value :")
b = int(input())
print("Enter a value :")
c = int(input())
```

```
def message():  # definition de la fonction
    print("Enter a value: ")    #corps

message() #invocation de la fonction
a = int(input())
message() #invocation de la fonction
b = int(input())
message() #invocation de la fonction
c = int(input())
```

Module 4 : Les fonctions (Paramètres)

- On peut créer des fonctions avec des paramètres.
- Dans l'exemple à droite :
  - name est un paramètre
  - name_user est un argument du paramètre name

```
def hello(name):    #définition de la fonction
    print("Hello,", name)    #corps

name_user = input("Enter your name: ")
hello(name_user)    #invocation
```

- Une fois qu'ils sont déclarés, les paramètres sont obligatoires lors de l'invocation de la fonction.

```
#invocation de hello
hello()
```

- Le code hello() génère une erreur car la fonction est appelée sans arguments. Testez !
- Il est possible de créer une variable du même nom qu'un paramètre. On appelle cela le shadowing.
- Le paramètre number est une entité complètement différente de la variable number.

```
def message(number):
    print("Enter a number:", number)

number = 1234
message(1)
print(number)
```

Module 4 : Les fonctions (Arguments positionnels)

- Une fonction peut avoir plusieurs paramètres (Attention, plus il y'en a , plus vite on oubliera à quoi ils servent).
- Exemple : Quand j'invoque la fonction message avec « price » et « 5 ». L'argument price est positionnée dans le paramètre texte, l'argument 5 est positionnée dans le paramètre value.

```
def message(text, value):
    print("Enter", text, "value", value)


message("telephone", 0102030405)
message("price", 5)
message("number1", "number2")
```

- On dit que la fonction message est invoquée par Passage d'arguments positionnels.

Module 4 : Les fonctions (Arguments mots clés)

- On peut aussi invoquer des fonctions par Passage d'arguments mots clés.
- Dans ce cas là, la position importe peu, l'argument est passé directement par son nom.

```
def introduction(firstName, lastName):
    print("Hello, my name is", firstName, lastName)


introduction(firstName = "James", lastName = "Bond")
introduction(lastName = "Skywalker", firstName = "Luke")
```

- Ici la fonction introduction est invoquée par Passage d'arguments mots clés. Les paramètres mots clés sont *firstName* et *Lastname*.

Module 4 : Les fonctions (Arguments mixtes)

- On peut aussi invoquer des fonctions par en mixant les 2 types de passages d'arguments.

```
def adding(a, b, c):
    print("la somme de",a,b,c,"est", a + b + c)

adding(3, c = 1, b = 2)
adding(3, a = 1, b = 2) #Erreur1
adding(a = 1, b = 2,3)  #Erreur2
```

- Ici la fonction adding est invoquée par Passage d'arguments mixtes.
  - ❑ Erreur1 : Deux valeurs sont passées au même paramètre, a=3 via l'argument de position et a=1 via l'argument mot clé
  - ❑ Erreur2 : On appelle les arguments mots clés, puis l'argument positionnel. Il faudrait appelé d'abord l'argument positionnel, puis les arguments par mots clés

Module 4 : Les fonctions (Paramètres par défaut)

- Il est possible de créer une function avec des paramètres par défaut. Ici *LastName* est un paramètre par défaut.
- L'Erreur à la ligne 3 est due au fait que la function soit invoquée sans le parameter *firstName*.
- L'invocation peut se faire par passage d'argument positionnel ou mot clés (Ligne 4 et 5 du code).
- Il est possible d'écraser la valeur du paramètre par défaut, en donnant une valeur différente à l'argument.

```
def introduction(firstName, lastName="Smith"):
    print("Hello, my name is", firstName, lastName)


introduction() #Erreur

#invocation par argument positionnel ou mot clé
introduction("JOE")
introduction(firstName="Joe")

#possible d'écraser la valeur du parmaètre par défaut
introduction("JOE","Williams")
introduction("JOE",lastName="Williams")
```

Module 4 : Les fonctions (Paramètres par défaut)

- Il est possible d'aller plus loin et de donner une valeur par défaut à tous les paramètres d'une fonction :

```
def introduction(firstName="John", lastName="Smith"):
    print("Hello, my name is", firstName, lastName)

introduction()
introduction(lastName="Hopkins")
```

- Les valeurs des paramètres par défaut peuvent être écrasées en donnant des valeurs différentes au moment d'invoquer la function.

Module 4 : Les fonctions (return)

- Il existe des fonctions comme print qui ne renvoient rien (attention, « renvoyer » et « afficher » sont deux choses différentes) et des fonctions telles que input ou type qui renvoient une valeur. Vous pouvez capturer cette valeur en plaçant une variable devant. Exemple :

```
#Ne renvoie rien
x = print("hello")
print(x)

#Renvoie le type de liste1
liste1 = []
x= type(liste1)
print(x)
```

Module 4 : Les fonctions (return)

- Pour qu'une fonction retourne une valeur il faut utiliser le mot-clé return. L'instruction return signifie qu'on va renvoyer la valeur, pour pouvoir la stocker dans une variable.

- Quand on n'utilise pas return, la fonction renvoie *la valeur None*.

```
#La fonction retourne None
def carre(valeur):
    valeur * valeur

x = carre(5)
print(x)

#La fonction retourne un résultat
def carre(valeur):
    return valeur * valeur

x = carre(5)
print(x)
```

Module 4 : Les fonctions (return)

- L’instruction return arrête le déroulement de la fonction, le code après return ne s'exécutera pas (Si return n’est pas suivi d’une expression, elle ne servira qu’à arrêter l’exécution de la fonction).
- On peut renvoyer dans return plusieurs valeurs séparées par des virgules, et les capturer dans des variables séparées par des virgules.

```
#Return stop l'exécution
def happy_new_year(wishes = True):
    print("Hey...")
    if wishes == False:
        return

    print("Happy New Year!")

happy_new_year()
happy_new_year(False)
```

Module 4 : Les fonctions (Les listes)

- Une liste est une structure importante et très utile, notamment lors de la création de fonction.
- Il est possible d'envoyer une liste en tant qu'argument d'une fonction.
- Il est aussi possible de retourner une liste comme le résultat d'une fonction.

```python
#Liste comme argument
def list_sum(lst):
    s = 0
    for elem in lst:
        s += elem
    return s

x = list_sum([5, 4, 3])
print(x)
#Attention avec y Erreur car objet non itérable
y = list_sum(5)



#Liste comme résultat
def strangeListFunction(n):
    strangeList = []
    for i in range(0, n):
        strangeList.append(i)
    return strangeList

x = strangeListFunction(5)
print(x)
```

Module 4 : Les fonctions (Le scope)

- Exemple 1 : Le scope d'une fonction est la fonction elle-même. Les paramètres sont inaccessibles en dehors de la fonction.
- Exemple 2 : Une variable définit en dehors d'une fonction, existe dans le corps de la fonction (On dit que la variable à un scope dans la fonction).

 Il existe une exception voir Exemple 3

- Exemple 3 : Les valeurs de la variable var à l'intérieur et à l'extérieur de la fonction sont différentes. Mais quand on invoque la fonction, c'est la variable à l'intérieur de la fonction qui prend le dessus sur celle à l'extérieur.

```
#Exemple1
def scopeTest():
    x = 123
scopeTest()
print(x)

#Exemple2
def myFunction():
    print("Do I know that variable?", var)

var = 1
myFunction()
print(var)

#Exemple3
def myFunction():
    var = 2
    print("Do I know that variable?", var)

var = 1
myFunction()
print(var)
```

Module 4 : Les fonctions (Le scope)

- Une fonction peut modifier une variable externe. Comment ?
- Nous pouvons étendre le scope d'une variable grâce au mot clé global.
- En déclarant une variable avec global, la fonction oblige python à utiliser sa variable.

```
def myFunction():
    global var
    var = 2
    print("Do I know that variable?", var)
var = 1
myFunction()
print(var)
```

Module 4 : Les types (Tuple)

- Un tuple est une liste de valeurs non-mutable, c'est à dire non modifiable.
- Un tuple est défini comme une liste avec des parenthèses à la place des crochets.
- Les éléments d'un tuple sont accessibles via des indices débutant à zéro.
- Un tuple est un type ittérable et peut être parcouru par for (C'est même plus rapide que de parcourir une liste).

```
myTuple = (1, 10, 100, 1000)

print(myTuple[0])
print(myTuple[-1])
print(myTuple[1:])
print(myTuple[:-2])

for elem in myTuple:
    print(elem)
```

Module 4 : Les types (Tuple)

- Il est impossible d'ajouter/enlever des éléments dans un tuple.
- La fonction len peut être utilisée pour connaître le nombre d'éléments d'un tuple.
- L'opérateur + sert à joindre des tuples entre eux (Possible sur listes).
- L'opérateur * multiplie les tuples (Possible sur listes).
- Il est possible d'utiliser in ou not in pour vérifier l'existence d'un élément dans un tuple.

```
myTuple = (1, 10, 100)
print(len(myTuple))

t1 = myTuple + (1000, 10000)
t2 = myTuple * 3
print(t1)
print(t2)

print(10 in myTuple)
print(-10 not in myTuple)
```

Module 4 : Les types (Dictionnaire)

- Un dictionnaire est un type mutable.
- Un dictionnaire est un ensemble de clés-valeurs :
  - ❑ Chaque clé doit être unique.
  - ❑ Une clé peut-être de type int, float ou string.
- La fonction len est valide sur un dictionnaire
- Les dictionnaires sont à sens unique (on se sert de la clé pour trouver une valeur mais pas l'inverse).

```
dictionary = {"cat" : "chat", "dog" : "chien"}
phone_numbers = {'boss' : 55512, 'Suzy' : 22657}
empty_dictionary = {}

print(dictionary)
print(phone_numbers)
print(empty_dictionary)

print(dictionary['cat'])
print(phone_numbers['Suzy'])

len(dictionary)
```

Module 4 : Les types (Dictionnaire)

- Décortiquons ensemble ce code qui utilise le mot-clé in pour vérifier si des mot sont des clés(keys) du dictionnaire !!!

```
dictionary = {"cat" : "chat", "dog" : "chien"}
words = ['cat', 'lion', 'horse']
for word in words:
    if word in dictionary:
        print(word, "->", dictionary[word])
    else:
        print(word, "is not in dictionary")
```

Module 4 : Les types (Dictionnaire)

- Un dictionnaire n'est pas de type séquence mais des outils existent pour le parcourir avec une boucle.
- La méthode keys() permet de retourner un objet itérable qui correspond au clés du dictionnaires.
- Il y a aussi la méthode values() qui fonctionne comme keys() mais pour les valeurs.
- Une autre méthode est d'utiliser le mot clé items(). Cette méthode retourne des tuples où chaque tuple est une paire clé-valeur.

```
dictionary = {"cat" : "chat", "dog" : "chien"}
for key in dictionary.keys():
    print(key)


dictionary = {"cat" : "chat", "dog" : "chien"}
for french in dictionary.values():
    print(french)


dictionary = {"cat" : "chat", "dog" : "chien"}
for english, french in dictionary.items():
    print(english, "->", french)
```

Module 4 : Les types (Dictionnaire)

- Il est possible de modifier une valeur d'un dictionnaire en l'appelant par sa clé.
- Il est possible d'ajouter une nouvelle valeur à un dictionnaire de deux manières :
  - ❑ En appelant le dictionnaire avec une nouvelle clé et une nouvelle valeur.
  - ❑ En utilisant la méthode update().

```
#modifier un dictionnaire
dictionary = {"cat" : "chat", "dog" : "chien"}
dictionary['cat'] = 'minou'
print(dictionary)

#ajouter une valeur à un dictionnaire
dictionary = {"cat" : "chat", "dog" : "chien"}
dictionary['swan'] = 'cygne'
print(dictionary)

#ajouter une valeur à la fin du dictionnaire
dictionary = {"cat" : "chat", "dog" : "chien"}
dictionary.update({"duck" : "canard"})
print(dictionary)
```

Module 4 : Les types (Dictionnaire)

- Pour supprimer un élément, il faut utiliser l'instruction del. On supprime toujours une clé, ce qui entraîne la suppression de la valeur associée car une valeur ne peut exister seule.
- Pour supprimer directement le dernier élément d'un dictionnaire, on peut utiliser la méthode popitem().
- On utilise aussi del pour supprimer complètement un dictionnaire.

```
#supprimer une valeur d'un dictionnaire
dictionary = {"cat" : "chat", "dog" : "chien"}
del dictionary['dog']
print(dictionary)

#supprimer la derni_re valeur d'un dictionnaire
dictionary.popitem()
print(dictionary)

#supprimer un dictionnaire
del dictionary
```

Module 4 : Les exceptions

- Lorsque votre programme rencontre des erreurs, il peut s'arrêter brusquement.
- Les exceptions sont des événements qui se produisent lorsque quelque chose d'inattendu arrive.
- Pour éviter les interruptions dues aux exceptions, nous utilisons try et except.
- Regardez cet exemple :

```
value = int(input('Enter a natural number: '))
print('The reciprocal of', value, 'is', 1/value)
```

- Si l'utilisateur entre autre chose qu'un nombre naturel, une exception se produit et arrête le programme

Module 4 : Les exceptions

- L'instruction try/except permet de gérer les erreurs. Si une exception se produit, except exécute un autre bloc code.
- L'exception ValueError est levée lorsque la conversion en entier échoue.

```
try:
    value = int(input('Enter a natural number: '))
    print('The reciprocal of', value, 'is', 1/value)
except:
    print('I do not know what to do.')
```

- Vous pouvez ajouter plusieurs blocs except pour gérer différents types d'exceptions.

```
try:
    value = int(input('Enter a natural number: '))
    print('The reciprocal of', value, 'is', 1/value)
except ValueError:
    print('I do not know what to do.')
except ZeroDivisionError:
    print('Division by zero is not allowed in our Universe.')
```

Module 4 : Les exceptions (Exemple complet)

```
try:
    value = int(input('Enter a natural number: '))
    print('The reciprocal of', value, 'is', 1/value)
except ValueError:
    print('I do not know what to do.')
except ZeroDivisionError:
    print('Division by zero is not allowed in our Universe.')
except:
    print('Something strange has happened here... Sorry!')
```