Python Essentials

Mohamed SAMB
Ingénieur de Conception en Informatique
mohamedsamb176@gmail.com

SOMMAIRE

I. Introduction

II. Python a été créé au début des années 1990 par Guido van Rossum au Stichting Mathematisch Centrum (CWI, voir https://www.cwi.nl/) aux Pays-Bas en tant que successeur d'un langage appelé ABC. Guido reste le principal auteur de Python, bien qu'il inclue de nombreuses contributions d'autres personnes.

III. En 1995, Guido a continué son travail sur Python à la Corporation for National Research Initiatives (CNRI, voir https://www.cnri.reston.va.us/) à Reston, en Virginie, où il a publié plusieurs versions du logiciel.

IV. En mai 2000, Guido et l'équipe de développement de base de Python ont rejoint BeOpen.com pour former l'équipe BeOpen PythonLabs. En octobre de la même année, l'équipe PythonLabs a rejoint Digital Creations (aujourd'hui Zope Corporation ; voir https://www.zope.org/). En 2001, la Python Software Foundation (PSF, voir https://www.python.org/psf/) a été créée, une organisation à but non lucratif créée spécifiquement pour détenir la propriété intellectuelle liée à Python. Zope Corporation est un membre sponsor de la PSF.

Introduction (2)

II. Toutes les versions de Python sont Open Source (voir https://opensource.org/ pour la définition Open Source).

III. Il est utilisé pour :

   I. le développement Web (côté serveur),

   II. le développement de logiciels,

   III. les mathématiques,

   IV. les scripts système.

Introduction : Que peut faire Python ? (3)

II. Il peut être utilisé sur un serveur pour créer des applications Web.

III. Il peut être utilisé en complément d'un logiciel pour créer des flux de travail.

IV. Il peut se connecter à des systèmes de base de données. Il peut également lire et modifier des fichiers.

V. Il peut être utilisé pour gérer de grandes quantités de données et effectuer des opérations mathématiques complexes.

VI. Il peut être utilisé pour le prototypage rapide ou pour le développement de logiciels prêts à la production.

Introduction : Pourquoi Python ? (4)

II. Le langage fonctionne sur différentes plateformes (Windows, Mac, Linux, Raspberry Pi, etc.).

III. Il a une syntaxe simple similaire à la langue anglaise.

IV. Il a une syntaxe qui permet aux développeurs d'écrire des programmes avec moins de lignes que certains autres langages de programmation.

V. Il s'exécute sur un système d'interprétation, ce qui signifie que le code peut être exécuté dès qu'il est écrit. Cela signifie que le prototypage peut être très rapide.

VI. Il peut être traité de manière ***procédurale***, ***orientée objet*** ou ***fonctionnelle***.

Introduction : bon à savoir (5)

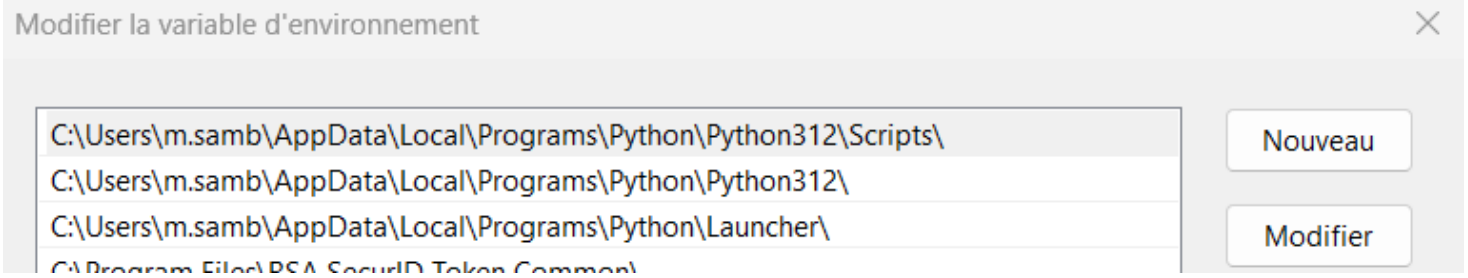
II. La version majeure la plus récente est ***Python 3***, que nous utiliserons dans ce tutoriel. Cependant, *Python 2*, bien qu'il ne soit pas mis à jour avec autre chose que des mises à jour de sécurité, reste assez populaire.

III. Installation :

I. Télécharger l'installeur via son site officiel https://www.python.org/downloads/

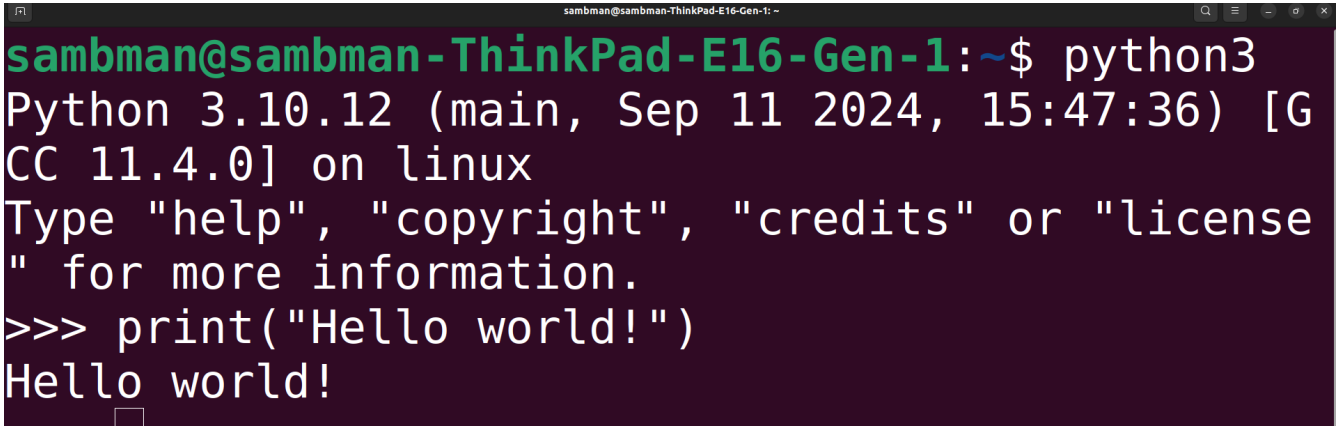
II. Exécuter le fichier (il vous proposera d'ajouter Python sur le ***Path***)

III. Une fois sur le Path, Python s'exécute sur n'importe quel répertoire du cmd ou Terminal.



Introduction : « Hello world ! » (6)

II. L'accès à l'interpréteur Python est effectif juste en tapant son nom (**python** ou **python3**).
    I. Le terme python en ligne de commande est un alias qui pointe sur la version installée.



Introduction : « Hello world ! » (7)

I. Une alternative serait de l'enrégister dans un fichier d'extension « .py » et de l'exécuter.

```
sambman@sambman-ThinkPad-E16-Gen-1:~/Documents/
Perso/Cours/ISI/L2/Python/TPs$ touch hello.py
sambman@sambman-ThinkPad-E16-Gen-1:~/Documents/
Perso/Cours/ISI/L2/Python/TPs$ gedit hello.py
sambman@sambman-ThinkPad-E16-Gen-1:~/Documents/
Perso/Cours/ISI/L2/Python/TPs$ python3 hello.py

Hello world!
```

II. **NB** : il existe des options d'exécution dont nous verrons certains dans ce cours.

II. Typage et manipulation des variables

- Le langage embarque des types prédéfinis qui sont les suivants :
  - Texte : str
  - Numériques : int, float, complex
  - Séquences : list, tuple, range
  - Mapping (clé-valeur) : dict
  - Ensembles : set, frozenset
  - Booléen : bool
  - Binaires : bytes, bytearray, memoryview
  - Type None (ou null dans les autres langages) : NoneType
- Dans cette partie, nous allons explorer :
  - La structure des types de base populaires
  - La manipulation des variables
    - Le typage dynamique ou « duck typing » en anglais

II. Typage et … : str

- Les chaînes en Python sont entourées soit de guillemets simples, soit de guillemets doubles.

```
>>> print("Hello")
Hello
>>> print('Hello')
Hello
>>>
```

- Vous pouvez utiliser des guillemets à l'intérieur d'une chaîne, à condition qu'ils ne correspondent pas aux guillemets entourant la chaîne :
  - `print("Il se nomme 'Johnny'")`
  - `print('Il se nomme "Johnny"')`

II. Typage et … : str (2)

- Assignation
  - L'affectation d'une chaîne à une variable se fait avec le nom de la variable suivi d'un signe égal et de la chaîne :
- Chaînes multi-lignes
  - L'affectation d'une chaîne à une variable se fait avec le nom de la variable suivi d'un signe égal et de la chaîne :
    - a = """Lorem ipsum dolor sit amet,

      consectetur adipiscing elit,

      sed do eiusmod tempor incididunt

      ut labore et dolore magna aliqua."""
    - print(a)

II. Typage et … : opérations sur str (3)

- Sous chaînes
  - Vous pouvez renvoyer une plage de caractères en utilisant la syntaxe de découpage.
  - Spécifiez l'indexe de début et l'indexe de fin, séparés par deux points, pour renvoyer une partie de la chaîne.
  - Exemple :
    - Obtenez les caractères de la position 1 à la position 5 (non inclus) :
      - b = "Hello, World!"
      - print(b[1:5]) # sortie ello

II. Typage et … : opérations sur str (4)

- Sous chaînes
  - En omettant l'indexe de départ, la plage commencera au premier caractère :
    - Exemple :
      - b = "Hello, World!"
      - print(b[:5]) # sortie => Hello
  - En omettant l'indexe de fin, la plage ira jusqu'à la fin :
    - Exemple :
      - b = "Hello, World!"
      - print(b[2:]) # sortie => llo, World!

II. Typage et … : opérations sur str (5)

- Sous chaînes
  - Indexation négative :
    - Utilisez des index négatifs pour démarrer la tranche à partir de la fin de la chaîne.
    - Exemple : récupérer les caractères
      - De : « o » dans «Wolrd!» (position -5)
      - À, mais non inclus : «d» dans «World!» (position -2)
      - b = "Hello, World!"
      - print(b[-5:-2]) # sorite => orl

II.Typage et … : opérations sur str (5)

- Quelques méthodes sur les chaînes
  - La méthode upper() renvoie la chaîne en majuscules :
    - a = "Hello, World!"
    - print(a.upper()) # sortie => HELLO, WORLD!
  - La méthode lower() renvoie la chaîne en majuscules :
    - a = "Hello, World!"
    - print(a.lower()) # sortie => hello, world!
  - La méthode strip() supprime tout espace blanc du début ou de la fin :
    - a = "   Hello, World! "
    - print(a.strip()) # sortie => Hello, World!

II. Typage et … : numériques

- Il existe trois types numériques en Python :
  - int
  - float
  - complex
- Les variables de type numérique sont créées lorsque vous leur attribuez une valeur :
  - x = 1 # int
  - y = 2.8 # float
  - z = 1j # complex

II. Typage et … : numériques : int (2)

- Int, ou Integer, est un nombre entier, positif ou négatif, sans décimales, de longueur illimitée.
  - Exemples

```
x = 3
y = 4589622255488805
z = -3255522

print(type(x))
print(type(z))
print(type(z))
```

  - NB : La fonction type() permet de vérifier le type de tout *objet* Python.

II. Typage et … : numériques : float (3)

- Float, ou « nombre à virgule flottante », est un nombre, positif ou négatif, contenant une ou plusieurs décimales.
  - Exemples

```
x = 2.5
y = 5.10
z = -28.37

print(type(x))
print(type(z))
print(type(z))
```

- Float peut également être des nombres scientifiques avec un « e » pour indiquer la puissance de 10.
  - Exemple : x = 43e7

II. Typage et … : numériques : complex (4)

- Les nombres complexes sont écrits avec la lettre « j » en tant que partie imaginaire.
  - Exemples

```
x = 3 + 2j
y = 7j
z = -3j

print(type(x))
print(type(z))
print(type(z))
```

II. Typage et … : numériques : conversion (4)

- La conversion peut se faire d'un type à un autre avec les methodes int(), float() et complex().
  - Exemples

```
x = 2
y = 1.5
z = 1j

a = float(x) #conversion d'un int en float
b = int(y) #conversion d'un float en int
c = complex(x) #conversion d'un int en complex

print(type(a))
print(type(b))
print(type(c))
```

II. Typage et … : séquences : list

- Les listes sont utilisées pour stocker plusieurs éléments dans une seule variable.
- Les listes sont l'un des 4 types de données intégrés dans Python utilisés pour stocker des collections de données, les 3 autres sont Tuple, Set et Dictionary, tous avec des qualités et des utilisations différentes.
- Les listes sont créées à l'aide de ***crochets*** ou du constructeur list() :
  - thislist = ["apple", "banana", "orange"]
  - print(thislist)
- Les éléments de la liste sont ordonnés, modifiables et autorisent les valeurs en double.
- Les éléments de la liste sont indexés, le premier élément a l'indice [0], le deuxième élément a l'indice [1], etc.

II. Typage et … : séquences : list (2)

- Caractéristiques d'une liste
  - Ordonnée
    - Lorsque nous disons que les listes sont ordonnées, cela signifie que les éléments ont un ordre défini et que cet ordre ne changera pas.
    - Si vous ajoutez de nouveaux éléments à une liste, ceux-ci seront placés à la fin de la liste.
  - Changeable
    - La liste est modifiable, ce qui signifie que nous pouvons modifier, ajouter et supprimer des éléments d'une liste après sa création.
  - Autorise la duplication
    - Étant donné que les listes sont indexées, elles peuvent contenir des éléments ayant la même valeur :
      - thislist = ["apple", "banana", "orange", "apple", "orange"]
      - print(thislist)

II. Typage et … : séquences : list (3)

- Pour déterminer le nombre d'éléments d'une liste, utilisez la fonction len() :
  - thislist = ["apple", "banana", "orange"]
  - print(len(thislist))
- Les éléments de la liste peuvent être de n'importe quel type de données :
  - list1 = ["apple", "banana", "cherry"]
  - list2 = [1, 5, 7, 9, 3]
  - list3 = [True, False, False]
- Une liste peut contenir différents types de données :
  - list1 = ["abc", 34, True, 40, "male"]
- Utilisation du constructeur list() pour créer une liste :
  - thislist = list(("apple", "banana", "orange")) # notez les doubles paranthèses
  - print(thislist)

II. Typage et … : séquences : list (4)

- Accès aux éléments
  - Les éléments de la liste sont indexés et vous pouvez y accéder en vous référant au numéro d'indice :
    - thislist = ["apple", "banana", "orange"]
    - print(thislist[1])
  - L'indexation négative signifie commencer par la fin. -1 fait référence au dernier élément, -2 fait référence à l'avant-dernier élément, etc.
    - Afficher le dernier élément de la liste
      - thislist = ["apple", "banana", "orange"]
      - print(thislist[-1])
  - Sous-liste
    - Vous pouvez spécifier une plage d'index en spécifiant où commencer et où terminer la plage.
    - Lorsque vous spécifiez une plage, la valeur de retour sera une nouvelle liste avec les éléments spécifiés.
      - thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
      - print(thislist[2:5])
  - Pour déterminer si un élément spécifié est présent dans une liste, utilisez le mot-clé in.

II. Typage et … : séquences : list (5)

- Mise à jour des éléments
  - Pour modifier la valeur d'un élément spécifique, reportez-vous au numéro d'indice :
    - Pour changer le premier élément
      - thislist = ["apple", "banana", "orange"]
      - thislist[0] = "pineapple"
      - print(thislist)
  - Pour modifier la valeur des éléments dans une plage spécifique, définissez une liste avec les nouvelles valeurs et faites référence à la plage de numéros d'index dans laquelle vous souhaitez insérer les nouvelles valeurs :
    - thislist = ["apple", "banana", "orange"]
    - thislist[1:2] = ["blackcurrant", "watermelon"]
    - print(thislist)
  - Pour insérer un nouvel élément de liste, sans remplacer aucune des valeurs existantes, nous pouvons utiliser la méthode insert().
    - La méthode insert() insère un élément à l'index spécifié :
      - thislist = ["apple", "banana", "orange"]
      - thislist.insert(2, "watermelon")
      - print(thislist)

II. Typage et … : séquences : list (6)

- Ajout d’éléments
  - Pour ajouter un élément à la fin de la liste, utilisez la méthode append() :
    - thislist = ["apple", "banana", "cherry"]
    - thislist.append("watermelon")
    - print(thislist)
  - Pour ajouter des éléments d'une autre liste à la liste actuelle, utilisez la méthode extend().
    - thislist = ["apple", "banana", "cherry"]
    - tropical = ["mango", "pineapple", "papaya"]
    - thislist.extend(tropical)
    - print(thislist)
  - La méthode extend() n'a pas besoin d'ajouter de listes, vous pouvez ajouter n'importe quel objet itérable (tuples, ensembles, dictionnaires, etc.).

II. Typage et … : séquences : list (7)

- Suppression
  - La méthode remove() supprime l'élément spécifié.
    - thislist = ["apple", "banana", "cherry"]
    - thislist.remove("banana")
    - print(thislist)
  - S'il y a plusieurs éléments avec la valeur spécifiée, la méthode remove() supprime la première occurrence.
  - La méthode pop() supprime l'indice spécifié.
    - thislist = ["apple", "banana", "cherry"]
    - thislist.pop(1)
    - print(thislist)
  - Si vous ne spécifiez pas l'indice, la méthode pop() supprime le dernier élément.

II. Typage et … : séquences : tuple

- Les tuples sont utilisés pour stocker plusieurs éléments dans une seule variable.
- Tuple est l'un des 4 types de données intégrés dans Python utilisés pour stocker des collections de données.
- Un tuple est une collection ***ordonnée*** et ***immuable***.
- Les tuples sont écrits avec des parenthèses.
  - thistuple = ("apple", "banana", "cherry")
  - print(thistuple)
- Pour déterminer le nombre d'éléments d'un tuple, utilisez la fonction len() :
  - thistuple = ("apple", "banana", "cherry")
  - print(len(thistuple))

II. Typage et … : séquences : set

- Les ensembles sont utilisés pour stocker plusieurs éléments dans une seule variable.

- L'ensemble est l'un des 4 types de données intégrés dans Python utilisés pour stocker des collections de données.

- Un ensemble est une collection qui n'est pas ordonnée, immuable* et non indexée.
  - Les éléments définis ne sont pas modifiables, mais vous pouvez supprimer des éléments et en ajouter de nouveaux.

  - thisset = {"apple", "banana", "cherry"}
  - print(thisset)

II. Typage et … : séquences : set (2)

- Caractéristiques
  - Duplication non autorisée
    - Les ensembles ne peuvent pas avoir deux éléments avec la même valeur.
      - thisset = {"apple", "banana", "cherry", "apple"}
      - print(thisset)
    - True et False sont évalués respectivement en 1 et 0 et seront traités comme des éléments dupliqués dans un ensemble.
      - thisset = {"apple", "banana", "cherry", True, 1, 2}
      - print(thisset)
  - Non ordonné
    - Les éléments d'un ensemble n'ont pas d'ordre défini.
    - Les éléments d'un ensemble peuvent apparaître dans un ordre différent à chaque fois que vous les utilisez et ne peuvent pas être référencés par index ou par clé.

II. Typage et … : séquences : set (3)

- Caractéristiques
  - Les éléments d'un ensemble ne sont pas modifiables, ce qui signifie que nous ne pouvons pas les modifier une fois l'ensemble créé.
    - Une fois qu'un ensemble est créé, vous ne pouvez pas modifier ses éléments, mais vous pouvez supprimer des éléments et ajouter de nouveaux éléments.
- Pour déterminer le nombre d'éléments d'un ensemble, utilisez la fonction len().
  - thisset = {"apple", "banana", "cherry"}
  - print(len(thisset))
- Les éléments d'un ensemble peuvent être de n'importe quel type de données.
- Un ensemble peut contenir différents types de données.

II. Typage et … : séquences : set (4)

- Il est également possible d'utiliser le constructeur set() pour créer un ensemble.
  - thisset = set(("apple", "banana", "cherry")) # noter les doubles parenthèses
  - print(thisset)
- Accès aux éléments
  - Vous ne pouvez pas accéder aux éléments d'un ensemble en faisant référence à un index ou à une clé.
  - Mais vous pouvez parcourir les éléments de l'ensemble à l'aide d'une boucle for ou demander si une valeur spécifiée est présente dans un ensemble à l'aide du mot-clé in.

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
    print(x)
```

II. Typage et … : séquences : set (5)

- Ajout d'éléments
  - Pour ajouter un élément à un ensemble, utilisez la méthode add().

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```

  - Pour ajouter des éléments d'un autre ensemble à l'ensemble actuel, utilisez la méthode update().

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}
thisset.update(tropical)
print(thisset)
```

  - L'objet dans la méthode update() n'a pas besoin d'être un ensemble, il peut s'agir de n'importe quel objet itérable (*tuples*, *listes*, *dictionnaires*, etc.).

II. Typage et … : séquences : set (6)

- Suppression
  - Pour supprimer un élément d'un ensemble, utilisez la méthode remove() ou discard().

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

  - Remarque : si l'élément à supprimer n'existe pas, discard() ne générera pas d'erreur.
  - Vous pouvez également utiliser la méthode pop() pour supprimer un élément, mais cette méthode supprimera *un élément aléatoire*, vous ne pouvez donc pas être sûr de l'élément supprimé.
    - La valeur de retour de la méthode pop() est l'élément supprimé.

II. Typage et … : séquences : dict

- Les dictionnaires sont utilisés pour stocker des valeurs de données dans des paires clé:valeur.
- Un dictionnaire est une collection ***ordonnée****, modifiable et qui n'autorise pas les doublons.
  - À partir de la version 3.7 de Python, les dictionnaires sont ordonnés. Dans Python 3.6 et les versions antérieures, les dictionnaires ne sont pas ordonnés.
- Les dictionnaires sont écrits avec des accolades et ont des clés et des valeurs :

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

II. Typage et … : séquences : dict (2)

- Caractéristiques
  - Les éléments du dictionnaire sont *ordonnés*, *modifiables* et ne permettent pas les doublons.
  - Les éléments du dictionnaire sont présentés sous forme de paires clé:valeur et peuvent être référencés à l'aide du nom de la clé.
    - Exemple : afficher la valeur «brand» du dictionnaire
      - `print(thisdict["brand"])`
  - Les dictionnaires sont modifiables, ce qui signifie que nous pouvons modifier, ajouter ou supprimer des éléments après la création du dictionnaire.
  - Les dictionnaires ne peuvent pas avoir *deux éléments avec la même clé*.
    - Les valeurs en double écraseront les valeurs existantes :

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964,
    "year": 2024
}
print(thisdict)
```

II. Typage et … : séquences : dict (3)

- Pour déterminer le nombre d'éléments d'un dictionnaire, utilisez la fonction len().
  - print(len(thisdict))
- Les valeurs des éléments du dictionnaire peuvent être de n'importe quel type de données.

```
thisdict = {
  "brand": "Ford",
  "electric": False,
  "year": 1970,
  "colors": ["red", "white", "blue"]
}
```

- Il est également possible d'utiliser le constructeur dict() pour créer un dictionnaire.

```
thisdict = dict(name = "Samba", age = 27, country = "Fouta")
print(thisdict)
```

II. Typage et … : séquences : dict (4)

- Accès aux éléments
  - Vous pouvez accéder aux éléments d'un dictionnaire en vous référant à son nom clé, entre crochets.

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
print(thisdict["model"])
```

  - Il existe également une méthode appelée get() qui vous donnera le même résultat :

```
x = thisdict.get("model")
```

  - La méthode keys() renverra une liste de toutes les clés du dictionnaire.
  - La méthode values() renverra une liste de toutes les valeurs du dictionnaire.
  - La méthode items() renverra chaque élément d'un dictionnaire, sous forme de tuples dans une liste.

II. Typage et … : séquences : dict (5)

- Mise à jour
  - Vous pouvez modifier la valeur d'un élément spécifique en vous référant à son nom clé.
    - Exemple : changer l'attribut « year » à 2018

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
thisdict["year"] = 2018
```

II. Typage et … : séquences : dict (6)

- Mise à jour
  - La méthode update() mettra à jour le dictionnaire avec les éléments de l'argument donné.
    - L'argument doit être un *dictionnaire* ou un objet *itérable* avec des paires clé:valeur.

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
thisdict.update({"year": 2020})
```

II. Typage et … : séquences : dict (7)

- Ajout d’éléments
  - L'ajout d'un élément au dictionnaire se fait en utilisant une nouvelle clé d'index et en lui attribuant une valeur.

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

  - La méthode update() mettra à jour le dictionnaire avec les éléments d'un argument donné. Si l'élément n'existe pas, il sera ajouté.

III. Structures de contrôles et structures répétitives

- La programmation procédurale se base sur ces types de structures sus-mentionnées pour dessiner le squelette d'un programme informatique.
- Ainsi, en faisant une combinaison de celles-ci, on obtient une suite logique d'instructions permettant d'effectuer des tâches spécifiques
- Ces structures sont monitorés par une condition logique évaluée en valeur booléenne (type bool en Python).

III. Structures de contrôles… : if...else

- Python prend en charge les conditions logiques habituelles des mathématiques :
  - Égal à : a == b
  - Pas égal à : a != b
  - Inférieur à : a < b
  - Inférieur ou égal à : a <= b
  - Supérieur à : a > b
  - Supérieur ou égal à : a >= b
- ces conditions peuvent être utilisées de plusieurs manières, le plus souvent dans des « instructions if » et des boucles.
  - Une « instruction if » est écrite en utilisant le mot-clé if.

```
a = 54
b = 200
if b > a:
  - print("b est supérieur à a")
```

III. Structures de contrôles… : if...else (2)

- Indentation :
  - Python s'appuie sur l'indentation (espaces au début d'une ligne) pour définir la portée du code. D'autres langages de programmation utilisent souvent des accolades à cette fin.
    - Instruction If, sans indentation (générera une erreur).

```
a = 54
b = 200
if b > a:
```

    - print("b est supérieur à a") # Cette générera une exception
- Le mot-clé elif est la manière dont Python dit « si les conditions précédentes n'étaient pas vraies, alors essayez cette condition ».

III. Structures de contrôles… : if...else (3)

- Le mot clé else intercepte tout ce qui n'est pas intercepté par les conditions précédentes.

```
a = 200
b = 33
if b > a:
  print("b est supérieur à a")
elif a == b:
  print("a and b sont égaux")
else:
  print("a est supérieur à b")
```

- Dans cet exemple, a est supérieur à b, donc la première condition n'est pas vraie, la condition elif n'est pas non plus vraie, donc nous allons à la condition else et affichons à l'écran que "a est supérieur à b".
- Vous pouvez également avoir un else sans elif.

III. Structures de contrôles… : if...else (4)

- Si vous n'avez qu'une seule instruction à exécuter, vous pouvez la placer sur la même ligne que l'instruction if.
  - if a > b: print("a est supérieur à b")
  - if...else forme contractée :
    - Si vous n'avez qu'une seule instruction à exécuter, une pour if et une pour else, vous pouvez tout mettre sur la même ligne.

```
a = 2
b = 330
print("A") if a > b else print("B")
```

    - Cette technique est connue sous le nom d'opérateurs ternaires ou d'expressions conditionnelles.
    - Vous pouvez également avoir plusieurs instructions else sur la même ligne.

III. Structures de contrôles… : if...else (5)

- Le mot clé and est un opérateur logique et est utilisé pour combiner des instructions conditionnelles.
  - Testez si a est supérieur à b, ET si c est supérieur à a :

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Toutes les conditions sont à True")
```

- Le mot clé or est un opérateur logique et est utilisé pour combiner des instructions conditionnelles.
  - Testez si a est supérieur à b, OU si a est supérieur à c :

```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("Au moins une condition est à True")
```

III. Structures de contrôles… : if...else (6)

- Le mot clé not est un opérateur logique et est utilisé pour inverser le résultat de l'instruction conditionnelle.
  - Testez si a n'est PAS supérieur à b :

```
a = 33
b = 200
if not a > b:
  print("a n'est pas supérieur à b")
```

- Vous pouvez avoir des instructions if à l'intérieur d'instructions if, cela s'appelle des instructions if imbriquées.

```
x = 41
if x > 10:
  print("Au dessus de dix,")
  if x > 20:
    print("et aussi au dessus de 10!")
  else:
    print("mais pas au dessus de 20.")
```

III. Structures de contrôles… : if...else (7)

- les instructions if ne peuvent pas être vides, mais si pour une raison quelconque vous avez une instruction if sans contenu, insérez l'instruction pass pour éviter d'obtenir une erreur.

```
a = 33
b = 200

if b > a:
  pass
```

III. Structures répétitives… : while

- Python possède deux commandes de boucle primitives :
  - boucles while
  - boucles for

- Avec la boucle while, nous pouvons exécuter un ensemble d'instructions tant qu'une condition est vraie.

```
i = 1
while i < 6:
  print(i)
  i += 1
```

  - Remarque : n'oubliez pas d'incrémenter i, sinon la boucle continuera ***indéfiniment***.

III. Structures répétitives… : while (2)

- Avec l'instruction break, nous pouvons arrêter la boucle même si la condition while est vraie.
  - Quitter la boucle si i vaut 3 :

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

III. Structures répétitives… : while (3)

- Avec l'instruction continue, nous pouvons arrêter l'itération en cours et continuer avec la suivante.
  - Continuer à la prochaine itération si i vaut 3 :

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

III. Structures répétitives… : while (4)

- Avec l'instruction else, nous pouvons exécuter un bloc de code une fois lorsque la condition n'est plus vraie.
  - Afficher un message une fois la condition est False :

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

III. Structures répétitives… : loop

- Une boucle for est utilisée pour parcourir une séquence (c'est-à-dire une liste, un tuple, un dictionnaire, un ensemble ou une chaîne).

- Cela ressemble moins au mot-clé for dans d'autres langages de programmation et fonctionne davantage comme une ***méthode d'itération*** comme celle que l'on trouve dans d'autres langages de programmation orientés objet.

- Avec la boucle for, nous pouvons exécuter un ensemble d'instructions, une fois pour chaque élément d'une liste, d'un tuple, d'un ensemble, etc.

III. Structures répétitives… : loop (1)

- Exemple :
  - Afficher chaque fruit de la liste fruits :

```
fruits = ["apple", "banana", "orange"]
for x in fruits:
  print(x)
```

- La boucle for ne nécessite pas de variable d'indexation à définir au préalable.
- Chaîne de caractères
  - Même les chaînes sont des objets itérables, elles contiennent une séquence de caractères :

```
for x in "banana":
  print(x)
```

III. Structures répétitives… : loop (2)

- Avec l'instruction break, nous pouvons arrêter la boucle avant qu'elle n'ait parcouru tous les éléments.
  - Quitter la boucle si x vaut « banana » :

```
fruits = ["apple", "banana", "orange"]
for x in fruits:
  print(x)
  if x == "banana":
    break
```

III. Structures répétitives… : loop (3)

- Avec l'instruction continue, nous pouvons arrêter l'itération actuelle de la boucle et continuer avec la suivante.
  - Ne pas afficher « banana » :

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

III. Structures répétitives… : loop (3)

- La fonction range()
  - Pour parcourir un ensemble de codes un nombre de fois spécifié, nous pouvons utiliser la fonction range().
  - La fonction range() renvoie une séquence de nombres, commençant par 0 *par défaut*, incrémentée de 1 (*par défaut*) et se terminant à un nombre spécifié.
  - Exemple :

```
for x in range(6):
  print(x)
```

    - Notez que range(6) ne correspond pas aux valeurs de 0 à 6, mais aux valeurs de 0 à 5.

III. Structures répétitives… : loop (4)

- La fonction range()
  - La fonction range() prend par défaut 0 comme valeur de départ, mais il est possible de spécifier ***la valeur de départ*** en ajoutant un paramètre : range(2, 6), ce qui signifie des valeurs de 2 à 6 (*mais sans inclure 6*) :

```
for x in range(2, 6):
  print(x)
```

  - La fonction range() incrémente par défaut la séquence de 1, mais il est possible de spécifier ***la valeur d'incrémentation en ajoutant un troisième paramètre*** : range(2, 30, 3) :

```
for x in range(2, 30, 3):
  print(x)
```

III. Structures répétitives… : loop (5)

- Le mot clé else dans une boucle for spécifie un bloc de code à exécuter lorsque la boucle est terminée.
  - Afficher tous les nombres de 0 à 5 et imprimez un message lorsque la boucle est terminée :

```
for x in range(6):
  print(x)
else:
  print("Exécution terminée !")
```

  - Remarque : le bloc else ne sera pas exécuté si la boucle est arrêtée par une *instruction* break.

III. Structures répétitives… : loop (6)

- Boucles imbriquées
  - Une boucle imbriquée est une boucle à l'intérieur d'une boucle.
  - La « boucle interne » sera exécutée une fois pour chaque itération de la « boucle externe ».
  - Exemple :
    - Afficher chaque adjectif pour chaque fruit :

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]
for x in adj:
  for y in fruits:
    print(x, y)
```

III. Structures répétitives… : loop (7)

- les boucles for ne peuvent pas être vides, mais si pour une raison quelconque vous avez une boucle for sans contenu, insérez l'instruction pass pour éviter d'obtenir une erreur.

```
for x in [0, 1, 2, 4, 5]:
  pass
```

IV.Fonctions

- Une fonction est un bloc de code qui ne s'exécute que lorsqu'il est appelé.
- Vous pouvez transmettre des données, appelées paramètres, à une fonction.
- Une fonction peut renvoyer des données en guise de résultat.
- En Python, une fonction est définie à l'aide du mot-clé def :

```
def my_function():
    print("Bonjour depuis une fonction !")
```

IV.Fonctions : (2)

- Appel de fonction
  - Pour appeler une fonction, utilisez le nom de la fonction suivi de parenthèses.

```
def my_function():
    print("Bonjour depuis une fonction !")

my_function()
```

IV.Fonctions : (3)

- Arguments
  - Les informations peuvent être transmises aux fonctions sous forme d'arguments.
  - Les arguments sont spécifiés après le nom de la fonction, entre parenthèses. Vous pouvez ajouter autant d'arguments que vous le souhaitez, séparez-les simplement par une virgule.
  - L'exemple suivant présente une fonction avec un argument (fname). Lorsque la fonction est appelée, nous transmettons un prénom, qui est utilisé à l'intérieur de la fonction pour imprimer le nom complet :

IV.Fonctions : (4)

- Exemple :

```
def my_function(fname):
    print(fname + " est un prénom.")

my_function("Emil")
my_function("Taro")
my_function("Zoro")
```

IV.Fonctions : (5)

- Nombre d’arguments
  - Par défaut, une fonction doit être appelée avec le nombre correct d'arguments. Cela signifie que si votre fonction attend 2 arguments, vous devez appeler la fonction avec 2 arguments, ni plus, ni moins.

  - Exemple
    - Cette fonction attend 2 arguments et obtient 2 arguments :

```
def my_function(fname, lname):
    print(fname + " " + lname)

my_function("Ala", "Badredine")
```

IV.Fonctions : (6)

- Arguments arbitraires, *args :
  - Si vous ne savez pas combien d'arguments seront passés dans votre fonction, ajoutez un * avant le nom du paramètre dans la définition de la fonction.

  - De cette façon, la fonction recevra un tuple d'arguments et pourra accéder aux éléments en conséquence.
  - Exemple :
    - Si le nombre d'arguments est inconnu, ajoutez un * avant le nom du paramètre :

```
def my_function(*kids):
    print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

IV.Fonctions : (7)

- Arguments clé-valeur
  - Vous pouvez également envoyer des arguments avec la syntaxe clé = valeur.
  - De cette façon, l'ordre des arguments n'a pas d'importance.
  - Exemple :

```
def my_function(child3, child2, child1):
    print("The youngest child is " + child3)

my_function(child1 = "Mamy", child2 = "Marc", child3 = "Aly")
```

IV.Fonctions : (8)

- Arguments clé-valeur arbitraires, **kwargs
  - Si vous ne savez pas combien d'arguments de mots-clés seront passés dans votre fonction, ajoutez deux astérisques : ** avant le nom du paramètre dans la définition de la fonction.
  - De cette façon, la fonction recevra un dictionnaire d'arguments et pourra accéder aux éléments en conséquence.
  - Exemple : si le nombre d'arguments de mots-clés est inconnu, ajoutez un double ** avant le nom du paramètre :

```
def my_function(**kid):
    print("Son nom est " + kid["lname"])

my_function(fname = "Lala", lname = "Diop")
```

IV.Fonctions : (9)

- Valeur de paramètre par défaut
  - L'exemple suivant montre comment utiliser une valeur de paramètre par défaut.

  - Si nous appelons la fonction sans argument, elle utilise la valeur par défaut.

```
def my_function(country = "Sénégal"):
    print("Je viens du " + country)

my_function("Suède")
my_function("Pakistan")
my_function()
my_function("Mali")
```

IV.Fonctions : (10)

- Passer une liste comme argument
  - Vous pouvez envoyer n'importe quel type de données d'argument à une fonction (chaîne, nombre, liste, dictionnaire, etc.), et il sera traité comme le même type de données à l'intérieur de la fonction.
  - Par exemple, si vous envoyez une liste en tant qu'argument, elle sera toujours une liste lorsqu'elle atteindra la fonction :

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

IV.Fonctions : (11)

- Valeurs de retour
  - Pour permettre à une fonction de renvoyer une valeur, utilisez l'instruction return :

```
def my_function(x):
    return 6 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

IV.Fonctions : (11)

- L’instruction pass :
  - les définitions de fonction ne peuvent pas être vides, mais si pour une raison quelconque vous avez une définition de fonction sans contenu, placez-la dans l'instruction pass pour éviter d'obtenir une erreur.

V.Modules

- En Python, un module est un fichier contenant du code Python — généralement des fonctions, des classes ou des variables — qui peut être importé et utilisé dans d'autres programmes Python. Les modules permettent d'organiser le code en sections plus faciles à gérer et favorisent la réutilisabilité. Par exemple, le module intégré math de Python fournit des fonctions et constantes mathématiques comme sqrt() et pi.
- Les modules rendent les programmes plus faciles à maintenir, déboguer et comprendre. Ils permettent aux développeurs de diviser un code volumineux en parties plus logiques. Cette approche modulaire facilite aussi le travail en équipe, car chaque membre peut travailler sur un module différent sans interférence. Elle permet également de réutiliser du code dans plusieurs projets.

V.Modules: (2)

- Création d’un module :
  - Créer un module personnalisé en Python est aussi simple qu’écrire des fonctions et les enregistrer dans un fichier .py. Pour utiliser ce module dans un autre script, on utilise l’instruction import. Par exemple, si vous avez un fichier nommé calculatrice.py, vous pouvez l’importer avec import calculatrice et appeler ses fonctions comme calculatrice.soustraire(50, 19).

V.Modules: (3)

- *Module standard* VS *module personnalisé* :
  - Python inclut une vaste bibliothèque standard de modules pour des tâches courantes comme la lecture/écriture de fichiers, l'accès au web, ou la sérialisation des données. Ces modules peuvent être importés directement, sans installation. En plus des modules intégrés, vous pouvez créer vos propres modules ou en installer d'autres via des outils comme pip.

V.Modules: (4)

- Utilisation d’un module :
  - Soit le module nommé mymodule.py contenant le code suivant :

```
def greeting(name):
    print("Hello, " + name)
```

  - Nous pouvons maintenant utiliser le module que nous venons de créer, en utilisant l'instruction import :

```
import mymodule

mymodule.greeting("Malang")
```

V.Modules: (5)

- Le module peut contenir des fonctions, comme déjà décrit, mais aussi des variables de tous types (tableaux, dictionnaires, objets etc) :
  - Enregistrez ce code dans le fichier mymodule.py

```
a_person = {
    "name": "Mack",
    "age": 32,
    "country": "Dakar"
}
```

  - Importez le module nommé mymodule et accédez au dictionnaire a_person :

```
import mymodule

a = mymodule.a_person["age"]
print(a)
```

V.Modules: (6)

- Renommage de module :
  - Vous pouvez créer un alias lorsque vous importez un module, en utilisant le mot-clé as :
    - Créez un alias pour mymodule appelé mm

```
import mymodule as mm


a = mm.a_person["age"]
print(a)
```

V.Modules: (7)

- Built-in modules :
  - Il existe plusieurs modules intégrés dans Python, que vous pouvez importer quand vous le souhaitez.

```
import platform


x = platform.system()
print(x)
```

V.Modules: (8)

- Contenu d’un module :
  - La fonction dict() permet d’afficher tous les éléments contenu dans un module donné.

```
import platform


x = dir(platform)
print(x)
```

VI. Programmation orientée objet (POO)

- Python est un langage de programmation orienté objet.
- En Python, presque tout est un objet, avec ses propriétés et ses méthodes.
- Une classe est comme un constructeur d'objet, ou un « modèle » pour créer des objets.
- Pour créer une classe, on utilise le mot clé class :

```
class MaClasse :
    x = 5
```

VI.POO: création d'un objet (2)

- Nous pouvons maintenant utiliser la classe nommée MaClass pour créer des objets :
  - Exemple :

```
p1 = MaClass()
print(p1.x)
```

VI.POO: création d'un objet (3)

- La fonction __init__():
  - Les exemples ci-dessus représentent des classes et des objets dans leur forme la plus simple, et ne sont pas vraiment utiles dans des applications réelles.
  - Pour comprendre la signification des classes, il est nécessaire de comprendre la fonction intégrée __init__().
  - Toutes les classes possèdent une fonction appelée __init__(), qui est toujours exécutée lors de leur lancement.
  - Utilisez la fonction __init__() pour affecter des valeurs aux propriétés d'un objet ou effectuer d'autres opérations nécessaires à sa création.
  - Cette fonction est appelée ***constructeur*** dans tout langage orienté objet.

VI.POO: création d'un objet (4)

- La fonction __init__():
  - Exemple :
    - Créez une classe nommée Personne, utilisez la fonction __init__() pour attribuer des valeurs pour le nom et l'âge :

```
class Personne:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Personne("John", 36)

print(p1.name)
print(p1.age)
```

VI.POO: fonction __str__() (5)

- La fonction __str__():
  - La fonction __str__() contrôle le contenu renvoyé lorsque l'objet de classe est représenté sous forme de chaîne.
  - Si la fonction __str__() n'est pas définie, la représentation sous forme de chaîne de l'objet est renvoyée :

VI.POO: fonction __str__() (6)

- La fonction __str__():
  - Exemple 1 : représentation d'un objet sans redéfinition de cette fonction

```
class Personne:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Personne("John", 36)

print(p1)
```

VI.POO: fonction __str__() (7)

- La fonction __str__():
- Exemple 1 : représentation d'un objet avec redéfinition de la fonction__str__()

```
class Personne:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def __str__(self):
    return f"nom: {self.name}, age: {self.age}"

p1 = Personne("John", 36)
print(p1)
```

VI.POO: méthodes (8)

- Les objets peuvent également contenir des méthodes. Les méthodes d'un objet sont des fonctions qui lui appartiennent.

- Créons une méthode dans la classe Personne :
  - Exemple :

```
class Personne:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Bonjour, mon non c'est " + self.name)

p1 = Personne("John", 36)
p1.myfunc()
```

VI.POO: le paramètre self (9)

- Le paramètre self est une référence à l'instance courante de la classe et permet d'accéder aux variables qui lui appartiennent.
- Il n'est pas nécessaire de l'appeler self ; vous pouvez l'appeler comme vous le souhaitez, mais il doit être le premier paramètre de n'importe quelle fonction de la classe :

```
class Personne:
  def __init__(my_object, name, age):
    my_object.name = name
    my_object.age = age

  def myfunc(abc):
    print("Bonjour, mon non c'est " + abc.name)

p1 = Personne("John", 36)
p1.myfunc()
```

VI.POO: manipulation d'une propriété d'objet (10)

- Vous pouvez modifier les propriétés des objets comme ceci :

```
p1.age = 35
```

- Vous pouvez supprimer les propriétés des objets comme ceci :

```
del p1.age
```

- les définitions de classe ne peuvent pas être vides, mais si pour une raison quelconque vous avez une définition de classe sans contenu, placez le mot clé pass dans son contenu.

```
class Personne:
  pass
```

VI.POO: héritage (11)

- L'héritage permet de définir une classe qui hérite de toutes les méthodes et propriétés d'une autre classe.

- La classe parente est la classe dont on hérite, également appelée classe de base.

- La classe enfant est la classe qui hérite d'une autre classe, également appelée classe dérivée.

VI.POO: héritage (12)

- N'importe quelle classe peut être une classe parent, donc la syntaxe est la même que pour la création de n'importe quelle autre classe :
  - Exemple : créez une classe nommée Personne, avec les propriétés prenom et nom, et une méthode afficher_nom:

```
class Personne:
  def __init__(self, pr, no):
    self.prenom = pr
    self.nom = no

  def afficher_nom(self):
    print(self.prenom, self.nom)

x = Personne("Mor", "LAM")
x.afficher_nom()
```

VI.POO: héritage (13)

- Création d'une classe enfant
  - Créez une classe nommée Etudiant, qui héritera des propriétés et des méthodes de la classe Personne :

```
class Etudiant(Personne):
  pass
```

    - NB : le mot clé pass est nécessaire lorsque vous n'aimeriez pas enrichir le contenu de la classe Etudiant avec d'autres propriétés.

VI.POO: héritage (14)

- Création d’une classe enfant
  - Lorsque vous ajoutez la fonction __init__(), la classe enfant n'héritera plus de la fonction __init__() du parent.
  - Pour conserver l'héritage de la fonction __init__() du parent, ajoutez un appel à la fonction __init__() du parent :
    - Exemple :

```
class Etudiant(Personne):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

VI.POO: héritage (15)

- Création d'une classe enfant
  - Python dispose également d'une fonction super() qui fera en sorte que la classe enfant hérite de toutes les méthodes et propriétés de son parent :
    - Exemple

```
class Etudiant(Personne):
  def __init__(self, prenom, nom):
    super().__init__(self, prenom, nom)
```

VI.POO: héritage (16)

- Il est possible d'ajouter des propriétés en plus de celles qui existent déjà dans la classe parente :
  - Exemple :

```
class Etudiant(Personne):
    def __init__(self, prenom, nom, annee_graduation):
        super().__init__(self, prenom, nom)
        self.annee_graduation = annee_graduation
```