L’algorithmie en Python

Introduction

L'algorithmie est une discipline fondamentale en programmation, qui consiste à concevoir des méthodes efficaces pour résoudre des problèmes. Un algorithme est une série d'étapes précises et organisées permettant d'accomplir une tâche spécifique. Ces étapes peuvent inclure des décisions (conditions), des répétitions (boucles), et la gestion d'exceptions (erreurs).

Dans ce cours, nous explorerons les concepts clés de l'algorithmie en Python, un langage réputé pour sa clarté et sa lisibilité. Grâce à sa syntaxe intuitive, Python est un outil idéal pour apprendre à écrire des algorithmes, même si vous débutez en programmation.

Pourquoi apprendre l'algorithmie en Python ?

1. La résolution de problèmes

L'algorithmie vous permet de diviser un problème complexe en sous-problèmes plus simples et de les résoudre étape par étape. Par exemple, un algorithme pourrait répondre à des questions comme :

- Comment trier une liste de nombres dans l'ordre croissant ?
- Comment trouver la valeur maximale d'une liste ?
- Comment automatiser une tâche répétitive ?

2. La prise de décision

Les algorithmes permettent à un programme de "réfléchir" et de prendre des décisions en fonction des données qu'il reçoit. Par exemple, un système peut décider d'accorder ou non un accès en fonction du mot de passe fourni.

3. L'automatisation et les répétitions

Vous pouvez automatiser des tâches répétitives avec des boucles, ce qui économise du temps et réduit les erreurs humaines. Par exemple, calculer la somme de tous les nombres dans une liste ou afficher tous les jours d'un mois.

4. La gestion des erreurs

En algorithmie, tout ne se passe pas toujours comme prévu. Par exemple, si un utilisateur entre une lettre au lieu d'un nombre, cela peut provoquer une erreur. Grâce à Python, nous pouvons gérer ces erreurs de manière élégante à l'aide des blocs try-except.

Les concepts abordés dans ce cours

Ce cours couvre les bases de l'algorithmie en Python à travers des concepts essentiels, notamment :

1. **Les conditions (if, elif, else)**

Les conditions permettent de prendre des décisions en fonction des données. Par exemple, afficher un message différent en fonction de l'âge d'une personne.

2. **Les boucles (for et while)**

Les boucles permettent de répéter un bloc de code plusieurs fois, que ce soit pour parcourir une liste, effectuer un calcul, ou interagir avec un utilisateur.

3. **Les instructions spéciales (break, continue)**

Ces instructions permettent de contrôler précisément le comportement d'une boucle en interrompant ou en sautant certaines itérations.

4. **La gestion des erreurs (try-except)**

Cette structure permet de gérer les cas où le programme rencontre une situation imprévue, comme une division par zéro ou une mauvaise saisie de l'utilisateur.

5. **Combinaison des outils**

Vous apprendrez à combiner ces concepts pour résoudre des problèmes plus complexes, comme créer un menu interactif ou développer un programme de devinette.

Partie 1. Les structures conditionnelles : if, elif, else

Les **conditions** sont essentielles pour la prise de **décisions** dans un programme. Elles permettent d'exécuter **différentes instructions** en fonction des données ou des situations. Par exemple, un programme peut réagir différemment selon l'âge de l'utilisateur, le résultat d'un **calcul** ou la **valeur** d'une variable.

Les structures conditionnelles répondent à des questions du type :
- L'utilisateur a-t-il le droit d'accéder à cette ressource ?
- Faut-il afficher une alerte si une valeur dépasse un certain seuil ?
- Le programme doit-il continuer ou s'arrêter en fonction d'une condition ?

Pourquoi les conditions sont importantes :
- Elles permettent de rendre les programmes interactifs et intelligents.
- Elles introduisent de la logique dans les algorithmes, en fonction des données traitées.
- Elles sont omniprésentes dans tous les langages de programmation et dans tous les domaines (web, jeux vidéo, systèmes embarqués, etc.).

---

1. Syntaxe de base

La syntaxe d'un **if** en Python est la suivante :

```
if condition: # type: ignore
    # Code exécuté si condition1 est vraie
```

Vous pouvez ajouter une condition alternative avec **elif** et une condition par défaut avec **else** :

```
if condition1: # type: ignore
    # Code exécuté si condition1 est vraie
elif condition2: # type: ignore
    # Code exécuté si condition1 est fausse et condition2 est vraie
else: # type: ignore
    # Code exécuté si aucune des conditions précédentes n'est vraie
```

Exemple :

```
age = int(input("Quel est votre âge ? "))

if age < 18:
    print("Vous êtes mineur.")
elif age == 18:
    print("Vous avez exactement 18 ans !")
else:
    print("Vous êtes majeur.")
```

2. Opérateurs de comparaison et logiques

Les conditions sont souvent basées sur des opérateurs de **comparaison** et des opérateurs **logiques**.

Opérateurs de comparaison :

Opérateur	Signification	Exemple	Résultat
==	Égal à	5 == 5	True
!=	Différent de	5 != 3	True
<	Inférieur à	3 < 7	True
>	Supérieur à	10 > 5	True
<=	Inférieur ou égal à	5 <= 5	True
>=	Supérieur ou égal à	6 >= 4	True

Opérateurs logiques :

Opérateur	Signification	Exemple	Résultat
and	Les deux conditions doivent être vraies	True and False	False
or	Au moins une des deux conditions est vraie	True or False	True
not	Inverse la valeur logique	not True	False

Exemple combiné :

```
x = 10
y = 20

if x > 5 and y < 30:
    print("Les deux conditions sont vraies.")
if not (x == 15):
    print("x n'est pas égal à 15.")
```

EXERCICES

1. Testez un nombre pair ou impair

- Demandez à l'utilisateur d'entrer un nombre.
- Affichez si ce nombre est pair ou impair.

2. Catégorie d'âge

- Demandez l'âge d'une personne.
- Affichez :
  - "Enfant" si l'âge est inférieur à 12 ans.
  - "Adolescent" si l'âge est entre 12 et 17 ans inclus.
  - "Adulte" si l'âge est supérieur ou égal à 18 ans.

3. Vérifiez une note

- Demandez une note (entre 0 et 20).
- Affichez un message selon la note :
  - ≥ 16 : "Excellent".
  - ≥ 12 : "Bien".
  - ≥ 8 : "Moyen".
  - < 8 : "Insuffisant".

Partie 2. Les boucles : for et while

Les boucles permettent de **répéter** un bloc de code plusieurs fois, ce qui est utile pour traiter des **collections** de données, effectuer des **calculs**, ou **automatiser** des tâches répétitives.

Les boucles répondent à des questions du type :
- Comment parcourir tous les éléments d'une liste ou d'un dictionnaire ?
- Comment effectuer un calcul pour chaque nombre dans une plage donnée ?
- Comment continuer à exécuter un programme tant qu'une condition reste vraie ?

Pourquoi les boucles sont importantes :
- Elles permettent de gagner du temps en automatisant des actions répétitives.
- Elles facilitent le traitement de grandes quantités de données.
- Elles sont utilisées dans presque tous les domaines, que ce soit pour parcourir des fichiers, manipuler des données ou gérer des processus.

Différences principales entre for et while :
- La boucle for est utilisée pour parcourir une séquence (liste, chaîne de caractères, dictionnaire, etc.) ou une plage de nombres définie avec range().
- La boucle while est utilisée lorsqu'on ne connaît pas à l'avance le nombre d'itérations nécessaires. Elle continue tant qu'une condition reste vraie.

---

1. La boucle for

La boucle **for** est utilisée pour **parcourir** une séquence (comme une **liste**, un **dictionnaire** ou une **chaîne de caractères**).

Syntaxe :

```
for variable in sequence:
    # Code à exécuter pour chaque élément de la séquence
```

Exemple : Parcourir une liste

```
fruits = ["pomme", "banane", "cerise"]

for fruit in fruits:
    print(fruit)
```

Sortie :

```
pomme
banane
cerise
```

Exemple : Parcourir une plage de nombres avec **range()**

```
for i in range(1, 6):  # Parcourt les nombres de 1 à 5 inclus
    print(i)
```

---

2. La boucle while

La boucle while est utilisée pour répéter un bloc **tant qu'une condition est vraie**.

Syntaxe :

```
while condition:
    # Code à exécuter tant que la condition est vraie
```

Exemple :

```
x = 0

while x < 5:
    print(x)
    x += 1  # Incrémentation de x
```

Déroulement de l'exécution :

1. La variable x est **initialisée à 0**.
2. La boucle while vérifie si **x < 5**. Tant que cette condition est vraie, le code à l'intérieur de la boucle est exécuté.
3. À chaque itération :
   - La valeur actuelle de x est **affichée avec print(x)**.
   - x est ensuite **incrémenté de 1** avec x += 1.
4. Une fois que x atteint 5, **la condition x < 5 devient fausse**, et la boucle s'arrête.

Étape	Valeur de x	Condition x < 5	Action	Affichage
1	0	Vraie	print(0) et x += 1	0
2	1	Vraie	print(1) et x += 1	1
3	2	Vraie	print(2) et x += 1	2
4	3	Vraie	print(3) et x += 1	3
5	4	Vraie	print(4) et x += 1	4
6	5	Fausse	La boucle s'arrête	-

3. Instructions spéciales : break et continue

Ces instructions permettent de contrôler le comportement d'une boucle.

- **break :** Stoppe immédiatement l'exécution de la boucle, quel que soit l'état de l'itération.
- **continue :** Saute immédiatement à l'itération suivante, en ignorant le reste du code de la boucle pour l'itération actuelle.

Pourquoi utiliser break et continue ?

- Elles permettent de gérer des cas spécifiques dans une boucle.
- Elles améliorent la lisibilité du code en simplifiant certaines conditions complexes.

1. **break :** Interrompt la boucle immédiatement

Exemple :

```
for i in range(10):
    if i == 5:
        break
    print(i)
# Affiche : 0, 1, 2, 3, 4
```

2. **continue :** Passe à l'itération suivante sans exécuter le reste du code dans la boucle

Exemple :

```
for i in range(5):
    if i == 3:
        continue
    print(i)
# Affiche : 0, 1, 2, 4
```

EXERCICES

1. **Comptez de 1 à 10**
   - Utilisez une boucle **for** pour afficher les nombres de **1 à 10**.

2. **Somme des nombres**
   - Demandez à l'utilisateur d'entrer un nombre n.
   - Utilisez une boucle pour calculer la somme des nombres de 1 à n.

3. **Trouver un élément**
   - Créez une liste contenant plusieurs noms.
   - Demandez à l'utilisateur un nom à rechercher.
   - Parcourez la liste avec une boucle pour vérifier si le nom est présent.

Partie 3. La gestion des erreurs : try-except

Les **erreurs** sont inévitables en programmation. Elles peuvent survenir pour plusieurs raisons, comme une **saisie** incorrecte de l'utilisateur, un fichier **introuvable**, ou une **division par zéro**. Sans **gestion** appropriée, ces erreurs provoquent **l'arrêt immédiat** du programme.

Le bloc **try-except** permet de **gérer** ces erreurs de manière élégante en les **anticipant**. Vous pouvez **définir** un comportement spécifique si une erreur survient, au lieu de laisser le programme se **bloquer**.

Les blocs try-except répondent à des questions du type :

- Que faire si l'utilisateur entre une lettre au lieu d'un nombre ?
- Comment gérer un fichier manquant sans planter le programme ?
- Comment afficher un message d'erreur clair pour guider l'utilisateur ?

Pourquoi gérer les erreurs est important :

- Cela rend votre programme plus robuste et fiable.
- Cela améliore l'expérience utilisateur en évitant des plantages brutaux.
- C'est une bonne pratique de programmation, utilisée dans tous les projets professionnels.

---

1. Syntaxe de base

```
try:
    # Code susceptible de générer une erreur
except:
    # Code exécuté si une erreur survient
```

Exemple :

```
try:
    x = int(input("Entrez un nombre : "))
    print(f"Le double de votre nombre est {x * 2}")
except ValueError:
    print("Erreur : Vous devez entrer un nombre valide.")
```

2. Le bloc finally

Le bloc **finally** permet d'exécuter un code, qu'une erreur ait eu lieu ou non.

```
try:
    x = int(input("Entrez un nombre : "))
except ValueError:
    print("Erreur : Ce n'est pas un nombre.")
finally:
    print("Fin du programme.")
```

EXERCICES

1. **Saisie d'un entier**
   - Demandez à l'utilisateur de saisir un nombre entier.
   - Si l'utilisateur entre autre chose qu'un entier, affichez un message d'erreur.

2. **Division sécurisée**
   - Demandez deux nombres à l'utilisateur.
   - Calculez leur division, en gérant l'erreur si le dénominateur est zéro.

Partie 4. Combiner les outils pour des algorithmes plus complexes

Les structures **conditionnelles**, les **boucles**, et la **gestion des erreurs** sont souvent **combinées** dans un programme pour résoudre des **problèmes complexes**. En les maîtrisant, vous pourrez créer des programmes **interactifs** et **efficaces**, capables de répondre à des besoins réels.

Exemple :

```
while True:
    print("\nMenu principal :")
    print("1. Ajouter un nombre")
    print("2. Afficher les nombres")
    print("3. Quitter")

    choix = input("Choisissez une option : ")

    if choix == "1":
        try:
            nombre = int(input("Entrez un nombre : "))
            print(f"Nombre ajouté : {nombre}")
        except ValueError:
            print("Erreur : Vous devez entrer un entier.")
    elif choix == "2":
        print("Affichage des nombres...")
    elif choix == "3":
        print("Au revoir !")
        break
    else:
        print("Option invalide, essayez encore.")
```

EXERCICE : JEU DE DEVINETTE AVEC MENU INTERACTIF

Dans cet exercice, vous **combinerez** les **conditions**, les **boucles**, et la **gestion des erreurs** pour créer un jeu de devinette interactif avec un menu. Ce programme permettra à l'utilisateur de choisir entre jouer, voir les règles, ou quitter.

Objectif du programme

1. Le programme doit afficher un menu avec 3 options :

```
1. Jouer
2. Règles du jeu
3. Quitter
```

2. Si l'utilisateur choisit :

   - **1 (Jouer) :**
     - Le programme choisit un nombre aléatoire entre 1 et 100 (***cf. Module Random( )***).
     - L'utilisateur doit deviner ce nombre.
     - Le programme donne des indices comme "Trop grand" ou "Trop petit".
     - Une fois le nombre deviné, affichez le nombre de tentatives de l'utilisateur.
   - **2 (Règles du jeu) :**
     - Affichez un texte expliquant les règles.
   - **3 (Quitter) :**
     - Le programme affiche "Merci d'avoir joué !" et termine.

3. Le programme doit gérer les erreurs si l'utilisateur entre une option invalide ou s'il saisit autre chose qu'un nombre.

Exemple d'exécution

Menu principal :

```
Menu principal :
1. Jouer
2. Règles du jeu
3. Quitter
Choisissez une option : 1
```

Si l'utilisateur choisit "Jouer" :

```
Bienvenue dans le jeu de devinette !
Devinez le nombre (entre 1 et 100) : 50
Trop grand !
Devinez le nombre (entre 1 et 100) : 25
Trop petit !
Devinez le nombre (entre 1 et 100) : 30
Bravo ! Vous avez trouvé le nombre 30 en 3 tentatives.
```

Si l'utilisateur choisit "Règles du jeu" :

```
Règles du jeu :
1. Le programme choisit un nombre aléatoire entre 1 et 100.
2. Vous devez deviner ce nombre.
3. Le programme vous indiquera si votre nombre est trop grand ou trop
petit.
4. Continuez jusqu'à trouver le bon nombre ! Bonne chance.
```

Si l'utilisateur choisit "Quitter" :

```
Merci d'avoir joué ! À bientôt.
```

Partie 5. Le mot-clé match

Le mot-clé **match** est une fonctionnalité introduite dans Python 3.10. Il permet d'écrire des structures conditionnelles plus claires et concises lorsqu'il s'agit de **comparer une variable à plusieurs cas possibles**. Il fonctionne de manière similaire à une structure switch que l'on retrouve dans d'autres langages de programmation comme C ou JavaScript.

---

1. Qu'est-ce que match ?

Le mot-clé match permet **de comparer une variable à différents modèles** (patterns) et d'exécuter un bloc de code en fonction du modèle correspondant. Cela rend le code plus lisible et évite les longues chaînes de if-elif-else.

Syntaxe générale

Voici la syntaxe d'une structure match :

```
match variable:
    case valeur1:
        # Instructions si variable == valeur1
    case valeur2:
        # Instructions si variable == valeur2
    case _:
        # Instructions par défaut si aucune des valeurs ne correspond
```

- **match variable :** Spécifie la variable ou l'expression à tester.
- **case valeur :** Définit un cas spécifique à comparer avec la variable.
- **_ :** Correspond au cas par défaut (équivalent de else dans une structure if-elif-else).

Exemple simple

Supposons que vous voulez afficher un message basé sur un jour de la semaine :

Avec une structure **if-elif-else** :

```
jour = "lundi"

if jour == "lundi":
    print("Début de la semaine.")
elif jour == "samedi" or jour == "dimanche":
    print("C'est le week-end !")
else:
    print("Jour de semaine.")
```

Avec match :

```
jour = "lundi"

match jour:
    case "lundi":
        print("Début de la semaine.")
    case "samedi" | "dimanche":  # Pipe (|) pour "ou"
        print("C'est le week-end !")
    case _:
        print("Jour de semaine.")
```

---

2. Fonctionnement de match

1. Utilisation avec des valeurs simples

Vous pouvez utiliser match pour comparer une variable avec des valeurs fixes comme des chaînes, des nombres, ou des booléens.

Exemple :

```
note = 15

match note:
    case 20:
        print("Excellent !")
    case 15:
        print("Très bien.")
    case 10:
        print("Passable.")
    case _:
        print("Échec.")
```

Sortie

```
Très bien.
```

2. Cas multiples avec | (ou logique)

Vous pouvez regrouper plusieurs cas dans un seul case en utilisant le symbole **|**, qui signifie “**ou**”.

Exemple :

```
jour = "samedi"

match jour:
    case "samedi" | "dimanche":
        print("C'est le week-end !")
    case _:
        print("C'est un jour de semaine.")
```

Sortie :

```
C’est le week-end !
```

3. Cas par défaut avec _

Si aucune des correspondances définies dans les case ne correspond à la variable, le cas _ (underscore) est exécuté. C’est l’équivalent d’un else.

Exemple :

```
couleur = "violet"

match couleur:
    case "rouge":
        print("Couleur chaude.")
    case "bleu":
        print("Couleur froide.")
    case _:
        print("Couleur non reconnue.")
```

Sortie :

```
Couleur non reconnue.
```

4. Utilisation avec des listes

Vous pouvez également utiliser match pour comparer des listes ou des séquences de données.

Exemple :

```
nombres = [1, 2, 3]

match nombres:
    case [1, 2, 3]:
        print("C'est une liste exacte : [1, 2, 3]")
    case [1, *_]:
        print("La liste commence par 1.")
    case _:
        print("Liste non reconnue.")
```

Sortie :

```
C'est une liste exacte : [1, 2, 3]
```

5. Cas avancé : Combiner match avec des conditions supplémentaires

Vous pouvez utiliser une condition (via le mot-clé if) pour affiner un case.

Exemple :

```
nombre = 10

match nombre:
    case x if x > 0:
        print("Nombre positif.")
    case x if x < 0:
        print("Nombre négatif.")
    case _:
        print("C'est zéro.")
```

Sortie :

```
Nombre positif.
```

2. Différences entre if-elif-else et match

if-elif-else	match
Vérifie des conditions logiques (par exemple, >, <, ==)	Conçu pour comparer des valeurs précises ou des modèles.
Syntaxe plus flexible mais moins lisible pour de nombreux cas.	Syntaxe plus claire et lisible pour les comparaisons multiples.
Adapté aux conditions complexes.	Idéal pour les correspondances directes (valeurs simples, tuples, etc.).

EXERCICES

Exercice 1 : Jour de la semaine

Créez un programme qui utilise match pour afficher :

- **“Début de semaine”** si le jour est "lundi".
- **“Milieu de semaine”** si le jour est "mercredi".
- **“Week-end”** si le jour est "samedi" ou "dimanche".
- **“Jour de semaine**” dans tous les autres cas.
-

Exercice 2 : Calcul simple

Demandez à l’utilisateur d’entrer un **opérateur** (**+, -, *, /**) et utilisez match pour effectuer le calcul correspondant entre deux nombres prédéfinis (par exemple, a = 10 et b = 5).

Exemple attendu :

```
Entrez un opérateur (+, −, ∗, /) : +
Le résultat est : 15
```

Exercice 3 : Classification d’une couleur

Créez un programme qui utilise match pour classifier des couleurs :

- "rouge", "jaune", "orange" : Affichez “**Couleur chaude**”.
- "bleu", "vert" : Affichez “**Couleur froide**”.
- Dans tous les autres cas : Affichez “**Couleur non reconnue**”.

Exercice 4 : Réécriture

Réécrivez l’exercice : « Jeu de devinette avec menu interactif » mais cette fois-ci, en utilisant **match**.