**TP2 POA Question 4**

**Partie A : Étude des Concepts Python**

# 1. Introduction

Python est un langage de programmation puissant et flexible. Dans ce document, nous allons explorer trois concepts clés en Python : la généricité, la modularité, et le polymorphisme (avec surcharge et redéfinition des méthodes).

# 2. Généricité en Python

La généricité en Python permet de créer des structures de données et des fonctions qui peuvent opérer sur différents types sans être spécifiquement liées à un type particulier. Python utilise le module **typing** pour implémenter la généricité.

Le code du fichier **Generics/generic\_box.py** suivant nous montre une implémentation de ce concept

1. from typing import TypeVar, Generic

2.

3. T = TypeVar("T") # Declare type variable

4.

5.

6. class Box(Generic[T]):

7. """

8. Generic class to hold items of any type.

9. """

10.

11. def \_\_init\_\_(self, item: T):

12. self.item = item

13.

14. def get\_item(self) -> T:

15. """

16. Returns the item contained in the box.

17. """

18. return self.item

19.

20.

21. # Create instances of Box with different types

22. int\_box = Box(5)

23. str\_box = Box("Hello")

24.

25. int\_box\_item = int\_box.get\_item()

26. str\_box\_item = str\_box.get\_item()

27.

28. print(f"int\_box\_item: {int\_box\_item}\nstr\_box\_item: {str\_box\_item}")

Dans cet exemple, la classe **Box** est définie comme une classe générique. Elle est capable de contenir un élément de n'importe quel type grâce à la variable de type **T**. Cette variable de type est définie par **TypeVar**, qui est une façon de déclarer un type sans spécifier exactement quel est ce type. C'est ce qui donne à **Box** sa flexibilité.

Le code montre ensuite la création de deux instances de la classe **Box** : une pour contenir un entier (**int\_box**) et une autre pour contenir une chaîne de caractères (**str\_box**). Lors de l'exécution, ces **Box** renvoient respectivement les valeurs **5** et **Hello**, montrant ainsi que la classe générique fonctionne comme prévu pour différents types.

## Résultat d'exécution :

int\_box\_item: 5

str\_box\_item: Hello

Cet exemple illustre l'un des principaux avantages de la généricité : la réutilisabilité. Au lieu de définir une classe distincte pour chaque type d'élément que l’on pourrait vouloir stocker, on peut définir une seule classe générique qui fonctionne pour tous les types.

## Qualité offert par la généricité :

**Flexibilité :** Permet aux structures de données et fonctions d'opérer sur divers types sans être spécifiquement liées à un type particulier.

**Réutilisabilité :** Un seul code générique peut être utilisé pour différents types, évitant ainsi la redondance.

**Sécurité de type :** En utilisant le module typing, il est possible d'assurer que seuls les types souhaités sont utilisés avec le code générique.

**Lisibilité :** En utilisant des types génériques, le code peut être plus explicite sur les types d'opérations qu'il supporte.

# 3. Modularité en Python

En programmation, on entend par modularité la capacité de diviser un programme en petites unités ou modules indépendants. Chaque module est conçu pour remplir une fonction spécifique et peut fonctionner de manière autonome, sans dépendre des autres modules. En adoptant cette approche, on bénéficie de nombreux avantages, tels qu'une meilleure maintenance, la possibilité de réutiliser le code et une lisibilité accrue.

L'archive **Modularity** offre une manière dont on peut appliquer la modularité en Python.

**math\_module.py :** Dans ce module, on a défini une fonction, **add**, destinée à additionner deux nombres.

1. def add(a, b):

2. """

3. Adds two numbers together and returns the result.

4.

5. :param a: int: The first number.

6. :param b: int: The second number.

7.

8. :return: The sum of the two numbers.

9. :rtype: int

10. """

11. return a + b

**main\_program.py :** Il s'agit du programme principal qui utilise la fonction **add** du module **math\_module**. Le code nous montre comment importer une fonction d'un autre module et l'utiliser. Cette séparation des préoccupations rend le code plus organisé et maintenable.

1. from math\_module import add

2.

3.

4. def main():

5. """

6. Main function to demonstrate the use of math\_module.

7. """

8. result = add(10, 20)

9. print(f"Result: {result}")

10.

11.

12. if \_\_name\_\_ == "\_\_main\_\_":

13. main()

## Résultat d'exécution :

Result: 30

On peut voir que le résultat d’exécution est bien celui attendu puisque que nous souhaitions additionner 10 et 20 et la fonction add du module **math\_module.py** à fait le calcul.

## Qualité offert par la modularité :

**Organisation :** Divise le code en unités distinctes basées sur leur fonctionnalité.

**Maintenabilité :** Les modules indépendants sont plus faciles à comprendre, à tester et à déboguer.

**Réutilisabilité :** Les modules peuvent être réutilisés dans différents projets sans avoir à les réécrire.

**Lisibilité accrue :** En séparant le code en modules, il est plus facile pour un nouveau développeur de comprendre le flux du programme.

# 4. Polymorphisme, surcharge des méthodes (overloading) et redéfinition des méthodes (overriding) en Python

Le polymorphisme est un concept fondamental en programmation orientée objet qui permet à des objets de types différents de traiter des opérations de manière différente, mais d'être appelés par le même nom. En Python, le polymorphisme est principalement mis en œuvre grâce à l'héritage et la redéfinition des méthodes.

Examinons l'implémentation fournie dans le fichier **PolymorphismOverloadingOverriding/animals.py** :

1. class Animal:

2. """

3. Base class for all animals.

4. """

5.

6. def speak(self) -> str:

7. """

8. Returns the sound made by the animal.

9. """

10. return "Unknown sound"

11.

12.

13. class Dog(Animal):

14. """

15. Dog class inheriting from Animal.

16. """

17.

18. def speak(self) -> str:

19. """

20. Returns the sound made by the dog.

21. """

22. return "Woof!"

23.

24.

25. class Cat(Animal):

26. """

27. Cat class inheriting from Animal.

28. """

29.

30. def speak(self) -> str:

31. """

32. Returns the sound made by the cat.

33. """

34. return "Meow!"

35.

36.

37. # Test polymorphism

38. def animal\_sound(animal: Animal) -> str:

39. """

40. Returns the sound of the given animal.

41. """

42. return animal.speak()

43.

44.

45. # Testing the classes

46. dog = Dog()

47. cat = Cat()

48.

49. dog\_response = dog.speak()

50. cat\_response = cat.speak()

51. polymorphic\_response = animal\_sound(dog), animal\_sound(cat)

52.

53. print(f"dog: {dog\_response}\ncat: {cat\_response}\npolymorphic: { polymorphic\_response }")

54.

Ici, on a une classe de base, **Animal**, qui a une méthode **speak()**. Cette méthode est ensuite redéfinie (ou "overriden") dans les classes **Dog** et **Cat**. C'est un exemple de redéfinition de méthodes.

La méthode **animal\_sound** illustre le polymorphisme. Bien qu'elle accepte un paramètre de type **Animal**, elle peut traiter n'importe quel objet qui est un sous-type d'**Animal** (comme **Dog** ou **Cat**) et appeler la méthode **speak** appropriée pour cet objet.

## Résultat d'exécution :

dog: Woof!

cat: Meow!

polymorphic: ('Woof!', 'Meow!')

Le résultat montre clairement le polymorphisme en action. Lorsque la méthode **speak** est appelée sur un objet **Dog**, elle renvoie **"Woof!"**, et lorsqu'elle est appelée sur un objet **Cat**, elle renvoie **"Meow!"**. Dans le test polymorphique, la fonction **animal\_sound** est appelée pour chaque type d'animal (chien et chat), et les réponses sont regroupées en un tuple, ce qui donne (**'Woof!', 'Meow!'**).

## Qualité offert par le polymorphisme :

**Flexibilité :** Permet à différentes classes d'implémenter des méthodes avec le même nom mais des comportements différents.

**Extensibilité :** Les classes dérivées peuvent étendre ou modifier le comportement de méthodes héritées de la classe parent.

**Réutilisabilité :** Les méthodes de la classe parent peuvent être réutilisées par toutes les classes dérivées.

**Abstraction :** On peut définir une méthode dans une classe parent (par exemple, Animal) sans savoir comment cette méthode sera implémentée dans les classes dérivées (comme Dog ou Cat).

**Partie B : Avantage de Python pour les analyses de données**

Python est un langage très efficace pour la manipulation et l’analyse de données. Dans cette partie nous allons montrer des exemples d’utilisation de python avec un grand nombre de données de ventes de produits.

# Utilisation de pandas pour la manipulation des données

**sale\_dataframe\_creator.py :**

1. def \_create\_dataframe(self):

2. """

3. Create a DataFrame with fictitious data representing sales.

4.

5. :return: DataFrame containing the fictitious data.

6. :rtype: pd.DataFrame

7. """

8. # Settings

9. np.random.seed(int(time.time()))

10. dates = pd.date\_range("2023-10-01", "2023-10-31")

11. products = ["Computer", "Phone", "Chair", "Desk", "Headset"]

12.

13. # Create the DataFrame

14. data = {

15. "Date": np.random.choice(dates, 500),

16. "Store": np.random.choice(["Store\_A", "Store\_B", "Store\_C"], 500),

17. "Product": np.random.choice(products, 500),

18. "Sales": np.random.randint(1, 50, 500),

19. }

20.

21. return pd.DataFrame(data)

22.

Ce morceau de code illustre comment, avec quelques lignes, on peut générer un DataFrame de pandas avec des données fictives. C'est un exemple de la flexibilité et de l'efficacité de pandas. Cela nous permet de crée des expériences de Test pour de l’analyse de données. A chaque exécution du programme un jeu de test différent sera créé.

**sale\_analyzer.py :**

1. return self.dataframe.groupby('Product').sum(numeric\_only=True).sort\_values(by='Sales', ascending=False)['Sales']

Ici, en une seule commande, on peut regrouper les données par produit, sommer les ventes et les trier. Cela montre la capacité de pandas à réaliser des opérations complexes en une seule ligne.

# Modélisation et Prévision avec scikit-learn

**sale\_predictor.py :**

1. model = LinearRegression()

2. model.fit(x, y)

3. predicted\_sales = model.predict(future\_dates)

Avec **scikit-learn**, il est possible de créer et d'entraîner un modèle de régression linéaire en quelques lignes de code. Ce passage démontre la simplicité avec laquelle on peut mettre en œuvre des algorithmes d'apprentissage automatique en Python. Dans notre cas cela permet de prévoir les ventes à venir ce qui peut être très utile en analyse de données.

Sales for last 5 days:

Date Sales

2023-10-31 364

2023-10-30 571

2023-10-29 365

2023-10-28 334

2023-10-27 473

Predictions for next 5 days:

Date Predicted Sales

0 2023-11-01 433

1 2023-11-02 434

2 2023-11-03 436

3 2023-11-04 437

4 2023-11-05 439

On voit que notre modèle de régression linéaire nous donne les ventes à venir sur les 5 prochains jours.

# Visualisation des Données

**sale\_vizualiser.py :**

1. def plot\_sales\_trend(self):

2. """

3. Generates a plot of the sales trend for October 2023.

4. """

5. sales\_trend = self.dataframe.groupby('Date').sum(numeric\_only=True)['Sales']

6. sales\_trend.plot(figsize=(12, 6))

7. plt.title('Sales Trend for October 2023')

8. plt.xlabel('Date')

9. plt.ylabel('Total Sales')

10. plt.grid(True)

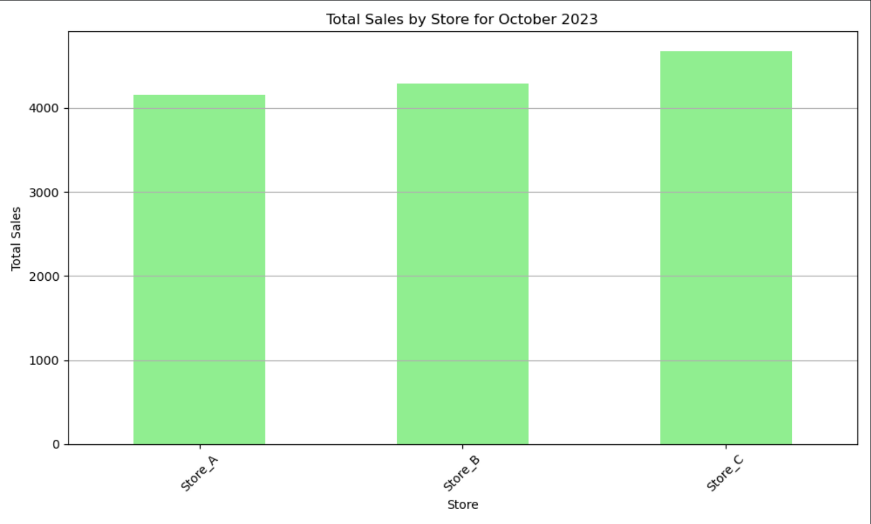
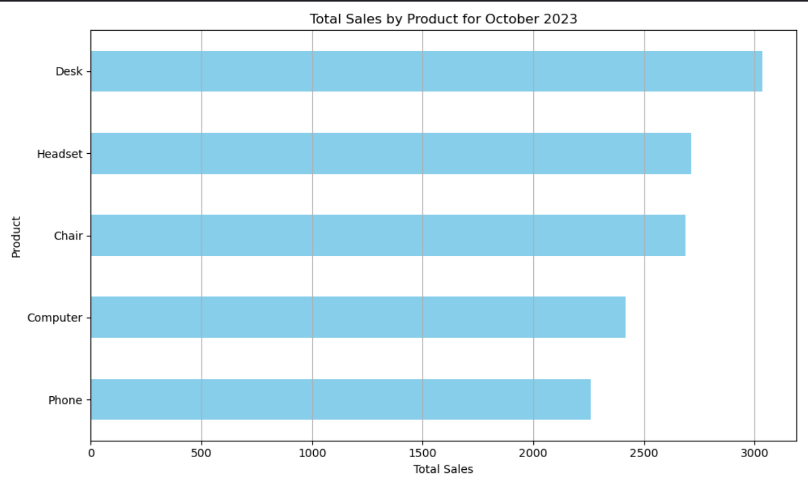
11. plt.show()

Ce code utilise **pandas** et **matplotlib** pour créer un diagramme à barres des ventes par produit. Cela illustre comment Python rend la visualisation de données simple et intuitive.

Une image contenant texte, ligne, diagramme, Tracé

Description générée automatiquement

Il est donc très facile avec python de faire des graphiques pour analyser des données.



# Utilité des Fonctions Auxiliaires

**helpers.py :**

1. return "${:,.2f}".format(value)

Les fonctions utilitaires comme celle-ci permettent de formatter les valeurs pour améliorer la lisibilité. C'est un exemple de la manière dont Python offre des outils pour gérer des tâches courantes de manière concise.

# Résultat d’éxécution du main.py :

Best Selling Products:

Date Sales

0 Desk 2572

… … …

4 Chair 2231

Best Selling Stores:

Date Sales

0 Store\_C 4351

… … …

2 Store\_B 3568

Best Selling Dates:

Date Sales

0 2023-10-08 512

… … …

30 2023-10-06 226

Total Sales:

$11,847.00

Sales for last 5 days:

Date Sales

0 2023-10-31 250

… … …

4 2023-10-27 388

Predictions for next 5 days:

Date Predicted Sales

0 2023-11-01 376

… … …

4 2023-11-05 374

# Conclusion

La puissance et la simplicité de Python, en particulier grâce aux bibliothèques pandas et scikit-learn, sont mises en évidence dans les exemples présentés. En quelques lignes, on peut générer des données, effectuer des analyses complexes et même prévoir des tendances futures grâce à l'apprentissage automatique. L'importance de la visualisation est également soulignée, tout comme la facilité avec laquelle Python gère des tâches quotidiennes, telles que le formatage des données. En résumé, ces exemples illustrent pourquoi Python est un choix privilégié pour l'analyse de données.