1 Manipulation de variables

Pour affecter une valeur ou le résultat d'une expression à une variable (sorte de case mémoire), on utilise le signe « égal ».
Il est recommandé d'utiliser des noms explicites pour les variables : cela aide la lecture du code.

Exemple :

```
distance = 5                    # en m
temps = 2                       # en s
vitesse = distance / temps      # en m/s
```

Sans savoir de quoi il s'agit, on devine (subtilement !) que ce programme calcule une vitesse à partir de la donnée d'une distance parcourue et du temps de parcours.

Remarque : il peut être intéressant d'ajouter des commentaires dans le code, après le caractère « dièse », pour expliciter certains points qui peuvent être plus délicats.

2 Définition d'une fonction

Une fonction est un bloc de code qui permet d'effectuer une tache spécifique et élémentaire.

Exemple : on désire créer et utiliser une fonction permettant de calculer une vitesse.
Cette fonction prend donc 2 paramètres en entrée (la distance et le temps) et renvoie un nombre en sortie (la vitesse).

Définition de la fonction :

```
def vitesse(d, t):
    """
    Sortie : renvoie la vitesse (rapport d/t)
    Entrées :
    - d : distance en m
    - t : temps en s
    """
    v = d / t
    return v
```

Appel de la fonction :

```
>>> vitesse(5, 2)
2.5
```

À retenir :
La définition d'une fonction commence par le mot clef `def`, qui est suivi du nom arbitraire que l'on choisit pour la fonction.
Le nom est suivi des paramètres d'entrée notés entre parenthèses.
Cette première ligne se termine par la ponctuation « 2 points ».
La valeur renvoyée par la fonction est introduite par le mot clef `return`.

Tout le bloc de code qui définit la fonction doit être *indenté* (décalé vers la droite) par rapport à la première ligne de la définition qui se termine par les 2 points.
Par la suite, on peut appeler cette fonction par son nom n'importe où dans le programme, et en indiquant entre parenthèses les valeurs des paramètres d'entrée.

3 Interaction avec l'utilisateur

3.1 Affichage : print

Pour afficher une chaîne de caractères (càd du texte entre guillemets doubles " ou simples '), il faut utiliser la fonction intégrée de Python `print`.
On peut séparer plusieurs morceaux de texte ou de variables à afficher par des virgules.

Exemple :

```
distance = 5                    # en m
temps = 2                       # en s
vitesse = distance / temps      # en m/s
print("La vitesse vaut : ", vitesse, "m/s.")
```

3.2 Saisie de données : input

Pour demander à l'utilisateur de saisir des données au clavier, il faut utiliser la fonction intégrée de Python `input`.
On peut préciser une chaîne de caractères comme paramètre de cette fonction (pour communiquer plus explicitement avec l'utilisateur).

Attention : piège classique ! La saisie au clavier est toujours interprétée par défaut comme une chaîne de caractères, or en sciences on demandera le plus souvent à l'utilisateur d'entrer des valeurs numériques. Il faut donc penser à convertir la chaîne de caractères en nombre avec la fonction intégrée de Python `float`.

Exemple :

```
dist = float(input("Saisir la distance (en m) : "))
tps = float(input("Saisir la temps (en s) : "))
v = vitesse(dist, tps)
print("La vitesse vaut : ", v, "m/s.")
```

Remarque : dans cet exemple, on a directement affecté les expressions saisies par l'utilisateur dans des variables (dist et tps). On a aussi utilisé la fonction `vitesse` définie précédemment en affectant son résultat à la variable v.
On remarquera aussi que le nom d'une variable à passer en paramètre d'une fonction n'est pas nécessairement celui utiliser dans la définition de la fonction (c'est l'ordre des paramètres qui importe !).

4 Structures de contrôles

4.1 Structure conditionnelle

Il est fréquent de devoir effectuer telle ou telle action en fonction d'une condition particulière. Dans ce cas, on utilise la structure conditionnelle **Si .. Sinon ... Alors** (if ... elif ... else).

Exemple : on affiche un message particulier en fonction de la vitesse du véhicule.

```
v_limite = 50   # vitesse limite autorisée en ville (en km/h)
dist = float(input("Saisir la distance (en km) : "))
tps = float(input("Saisir la temps (en h) : "))
v = vitesse(dist, tps)
if v > v_limite:
  print("Vous êtes en excès de vitesse, ralentissez immédiatement !")
elif 0.9 * v_limite < v <= v_limite:
  print("Prudence : vous frôler la vitesse limite...")
else:
  print("Félicitations, vous conduisez en toute prudence.")
```

Remarque : Bien noter l'indentation pour chaque bloc de code après les « 2 points » du if, elif et else.

4.2 Boucle non bornée

Il est aussi possible d'effectuer une action tant qu'une condition est satisfaite. Dans ce cas, on utilise la boucle non bornée **Tant que** (while).

Exemple : on désire contrôler l'accélération d'un véhicule autonome (on veut accélérer jusqu'à ce que la vitesse atteigne 90% de la vitesse limite).

```
v_limite = 50                # vitesse limite autorisée (en km/h)
v = 10                       # vitesse initiale du véhicule
while v <= 0.9 * v_limite:   # tant que v < 90% de v_limite
  v = v * 1.1                # augmentation de la vitesse de 10% de sa valeur
```

Remarque : Noter à nouveau l'indentation pour le bloc de code dans le cœur de la boucle.

4.3 Boucle bornée

Enfin, on peut vouloir effectuer une action un certain nombre de fois (connu à l'avance). Dans ce cas, on utilise la boucle bornée **Pour** (for).

Exemple : on désire donner 5 coups d'accélérateur qui augmente chacun la vitesse de 3 km/h.

```
v = 10               # vitesse initiale du véhicule
for i in range(5):   # action effectuée 5 fois
  v = v + 3          # augmentation de la vitesse de 3 km/h
```

Remarque : Noter comme toujours l'indentation pour le bloc de code dans la boucle.

5 Graphique et utilisation de listes

Pour tracer un graphique à l'issue d'une expérience, on a couramment une liste d'abscisses et une liste d'ordonnées.
Deux façons de représenter ces listes sont possibles en Python.

5.1 Méthode 1 : utiliser les listes Python

Python propose un type de données `list` et comme son l'indique, ça semble bien adapter à notre problématique !

Exemple : `listeX = [2, 3, 4, 5, 6, 7]` définit de façon simple une liste contenant les valeurs entières de 2 à 7.

On peut créer la même liste de façon automatisée ainsi :

```
listeX = []                 # création d'une liste vide
for i in range(2, 8):       # la variable i va prendre successivement
                            # toutes les valeurs de 2 à 7 (8 est exclu)
  listeX.append(i)          # ajout de la valeur de i à la fin de la liste
```

On peut aussi créer la même liste de façon plus concise :

```
listeX = [i for i in range(2, 8)]
```

En supposant que `listeX` soit une liste d'abscisses, on peut calculer une liste d'ordonnées `listeY` dépendant de ces abscisses suivant une certaine fonction.

Par exemple si $y = 2x + 3$, on créera ainsi la liste des ordonnées :

```
listeY = []                 # création d'une liste d'ordonnées vide
for x in listeX:            # la variable x va prendre successivement
                            # toutes les valeurs contenues dans listeX
  listeY.append(2*x+3)      # ajout de l'ordonnée calculée à la fin de la listeY
```

On peut aussi créer la même liste de façon plus concise :

```
listeY = [2*x+3 for x in listeX]
```

Deux fonctionnalités utiles :

— La taille d'une liste (ou sa longueur *length* in English) s'obtient ainsi : `len(liste)`.
Exemple : `len(listeX)` vaut 6.

— Pour accéder à un élément d'une liste à partir de son indice (attention : **le 1er élément est à l'indice 0** !!!), on écrit simplement : `liste[i]`.

Exemple : `listeX[0]` vaut 2, `listeY[3]` vaut 13 ($2 \times 5 + 3$).

5.2 Méthode 2 : utiliser les array du module numpy

Le module **numpy** propose un type de donnée `array` que l'on peut utiliser de façon basique pour stocker des listes. Le traitement des listes peut être très simplifié par cette méthode, notamment le calcul des ordonnées.

Il faut d'abord importer le module numpy (abrégé en `np` par l'ensemble de la communauté scientifique) : `import numpy as np`

La création de liste est très similaire aux listes Python : en fait on convertit une liste standard en array.
Exemple : `listeX = np.array([i for i in range(10])`

Remarque : numpy propose aussi ses propres fonctions de création de listes « régulières » :

```
# crée une 'liste-array' de 2 à 7 (8 est exclu)
listeX = np.arange(2, 8)
# crée une 'liste-array' de 10 valeurs entre 2 et 7 (inclus !)
listeX = np.linspace(2, 7, 10)
```

En supposant que `listeX` soit une liste d'abscisses, on peut calculer beaucoup plus rapidement une liste d'ordonnées `listeY`.
Par exemple si $y = 2x + 3$, on créera ainsi la liste des ordonnées :

```
listeY = 2*listeX + 3  # le calcul se fait élément par élément
                       # évidemment listeY a la même taille que listeX
```

5.3 Graphique

À partir de ces listes d'abscisses et d'ordonnées, on peut ensuite tracer le graphique $y = f(x)$.

Il faut d'abord importer le sous-module `matplotlib.pyplot` (abrégé en `plt` par l'ensemble de la communauté scientifique) : `import matplotlib.pyplot as plt`

La fonction pour tracer la courbe est alors simplement : `plt.plot(listeX, listeY)`

Il existe bien entendu des tas de fonctionnalités et d'options pour mettre en forme le graphique.

Quelques exemples classiques :

```
# 1ere courbe :
plt.plot(listeX, listeY1, label="courbe 1", marker="+", color="blue")
# 2ème courbe (partage les mêmes abscisses) :
plt.plot(listeX, listeY2, label="courbe 2", marker="o", color="red")
plt.legend()   # affiche la légende définie par les labels
plt.axis(xmin=0, xmax=10, ymin=-2, ymax=5)
plt.show()     # force l'affichage du graphique
```

5.4 Modélisation (fit)

Modéliser des données expérimentales consiste à chercher la fonction f qui représente le mieux ces données : $y = f(x)$ (avec les notations mathématiques habituelles). Bien souvent le modèle est connu à l'avance pour un problème étudié (ex : fonction linéaire, affine, exponentielle...).

5.4.1 Modélisation par une fonction affine (cas le plus fréquent)

Il faut importer la fonction `linregress` du module `numpy.stats`.

Cette fonction prend 2 paramètres en entrée :
- liste des abscisses (x).
- liste des ordonnées expérimentales (y_{exp}).

Elle renvoie plusieurs valeurs dont les 2 premières sont le coefficient directeur et l'ordonnée à l'origine de la fonction affine.

Dans la pratique on pourra donc écrire :

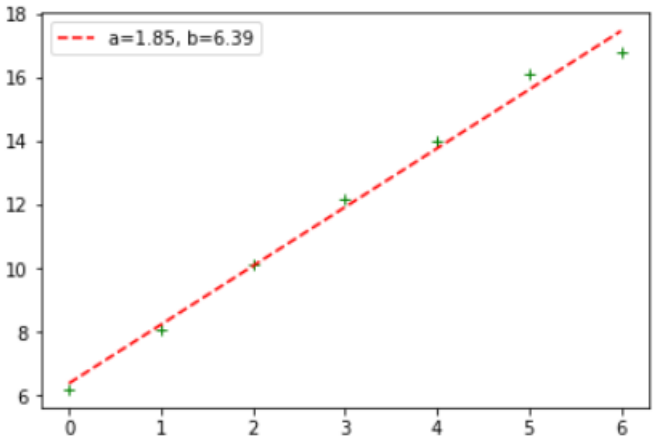
```
from scipy.stats import linregress # import de la fonction de modélisation

# données à modéliser :
x = np.array([0, 1, 2, 3, 4, 5, 6])
y_exp = np.array([6.2, 8.1, 10.1, 12.2, 14, 16.1, 16.8])

# obtention des paramètres du modèle :
a, b, *_ = linregress(x, y_exp)

# construction du modèle :
y_model = a * x + b

# graphique :
plt.plot(x, y_exp, 'g+')  # points expérimentaux
plt.plot(x, y_model, 'r--', label=f"a={a:1.2f}, b={b:1.2f}") # fonction modèle
plt.legend()
plt.show()
```



5.4.2 Modélisation par une fonction quelconque

La fonction précédente (linregress) ne permet que la modélisation par une fonction affine ; pour modéliser par une fonction de son choix (linéaire ou exponentielle par exemple), il faut importer la fonction `curve_fit` du module `numpy.optimize`.

Cette fonction prend 3 paramètres en entrée :
- la fonction modèle qui dépend de plusieurs paramètres (comme par exemple le coeff. directeur et l'ordonnée à l'origine dans le cas d'une fonction affine).
- liste des abscisses (x).
- liste des ordonnées expérimentales (y_{exp}).

Elle renvoie plusieurs valeurs dont la première est l'ensemble des paramètres de la fonction modèle avec leur valeur optimisée pour « fitter » au mieux la courbe expérimentale.

Dans la pratique on pourra donc écrire :

```
from scipy.optimize import curve_fit # import de la fonction de modélisation

# définition de la fonction modèle (ici exemple d'une fonction affine)
def modele(x, a, b):  # cette fonction dépend de 2 paramètres a et b
    return a * x + b

# obtention des paramètres du modèle :
(a, b), *_ = curve_fit(modele, x, y_exp) # noter les parenthèses obligatoires !

# construction du modèle :
y_model = a * x + b
```

Ensuite on peut construire les mêmes graphiques que précédemment.

5.5 Tracer de vecteurs

5.5.1 Méthode 1 : Tracer une flèche (arrow)

Pour tracer une simple flèche, on utilise la fonction `arrow`.
En notant x et y l'origine du vecteur et vx et vy ses coordonnées, on trace la flèche du vecteur ainsi :

```
plt.arrow(x, y, vx, vy, length_includes_head=True , head_width=0.2, \
                        head_length=0.5, overhang=0.2)
```

Quelques précisions :
- Toutes les grandeurs sont mesurées dans la même unité que x et y (vx et vy, ainsi que les paramètres optionnels).
- length_includes_head=True : la pointe de la flèche fait partie de la longueur du vecteur tracé.
- head_length = longueur de la flèche.
- head_width = largeur de la flèche.
- overhang = décalage du "début" de la flèche (= 0 par défaut, flèche triangulaire).

5.5.2 Méthode 2 : Tracer un champ de vecteurs (quiver)

On suppose qu'on connaît la valeur des cordonnées (Vx, Vy) d'un champ de vecteurs. On désire tracer ces vecteurs aux points de coordonnées (X, Y).
X, Y, Vx et Vy sont enregistrées dans des listes (ou array).

Pour tracer le champ de vecteurs, on utilise la fonction `quiver` ainsi :

```
plt.quiver(X, Y, Vx, Vy, angles='xy', scale_units='xy', scale=5)
```

Remarque : Parmi ces paramètres, on doit juste adapter la valeur de *scale*.
Si par exemple X et Y sont des cordonnées d'espace en m, et Vx et Vy des cordonnées de vitesses en m/s, alors une flèche correspondant à 5 m sur le graphique est en réalité une vitesse de 1 m/s (« 5 fois moins »).