Algorithmique & programmation Python

SI · Terminale · Bac

3 structures algorithmiques

Tout algorithme se ramène à **3 briques** élémentaires combinées :

Séquence
instructions exécutées **l'une après l'autre**

Alternative
if / else · choix selon un test

Boucle
while / for · répétition

```
while True:
    valeur = lire_capteur()
    if valeur > seuil:
        activer_moteur()
    else:
        arreter_moteur()
```

Variables & types Python

Type	Exemple	Usage SI
int	`n = 42`	compteur, durée ms
float	`t = 21.5`	température, tension
bool	`ok = True`	capteur TOR, drapeau
str	`"erreur"`	message série
list	`[1, 2, 3]`	historique mesures

Python est **dynamiquement typé** · le type est déduit de l'affectation.

Lecture capteur · CAN

Un **convertisseur analogique-numérique** sur n bits code 2^n valeurs.

$$\Delta U = \frac{V_{ref}}{2^n} \quad \cdot \quad U = \frac{val}{2^n - 1} \cdot V_{ref}$$

Arduino Uno · CAN **10 bits**, $V_{ref} = 5$ V :

$$\Delta U = \frac{5}{1024} \approx 4{,}9 \text{ mV}$$

```
from machine import ADC
adc = ADC(0)
val = adc.read()     # 0 → 1023
U = val / 1023 * 5   # tension V
```

Commande actionneur · PWM

Le **PWM** (modulation de largeur d'impulsion) varie le **rapport cyclique** $\alpha \in [0;1]$ pour simuler une tension analogique.

$$\boxed{U_{moy} = \alpha \cdot V_{alim}}$$

Ex : $V_{alim} = 12$ V, $\alpha = 0{,}75$ → $U_{moy} = 9$ V · vitesse moteur, luminosité LED

Fonctions & modularité

Une **fonction** regroupe un bloc d'instructions réutilisable, avec des **paramètres** et une **valeur de retour**.

```
def tension(val, vref=5):
    """Convertit lecture CAN en V"""
    return val / 1023 * vref

u = tension(512)   # 2.5 V
```

→ Toujours **commenter** le rôle d'une fonction (docstring)
→ Décomposer un programme en fonctions **courtes et claires**
→ Tester chaque fonction **indépendamment**

Listes · stocker des mesures

```
mesures = []
for i in range(10):
    mesures.append(adc.read())

moyenne = sum(mesures) / len(mesures)
maxi    = max(mesures)
```

→ **`append()`** ajoute en fin · **`len()`** longueur
→ **`sum()`**, **`max()`**, **`min()`** agrégations utiles
→ Accès par indice · `mesures[0]`, `mesures[-1]`

Bonnes pratiques embarqué

→ **Commenter** les choix non triviaux
→ **Décomposer** en fonctions réutilisables
→ **Tester chaque sous-système** séparément avant intégration
→ Gérer les **cas limites** · valeur hors plage, capteur déconnecté
→ Préférer les **constantes nommées** aux valeurs brutes
→ Utiliser `time.sleep_ms()` avec parcimonie

Une boucle **while True** sans pause peut saturer le CPU.

Asservissement · correcteur PID

Le PID corrige l'**erreur** $e(t) =$ consigne $-$ mesure :

$$u(t) = K_p\, e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{de}{dt}$$

```
erreur   = consigne - mesure
integ   += erreur * dt
deriv    = (erreur - prev) / dt
commande = Kp*erreur + Ki*integ + Kd*deriv
prev     = erreur
```

K_p rapidité · K_i élimine l'erreur statique · K_d stabilise