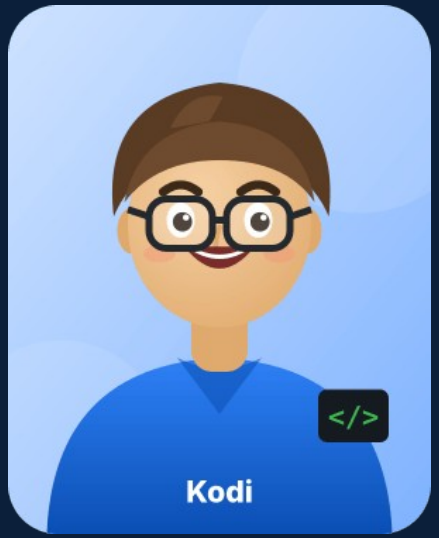
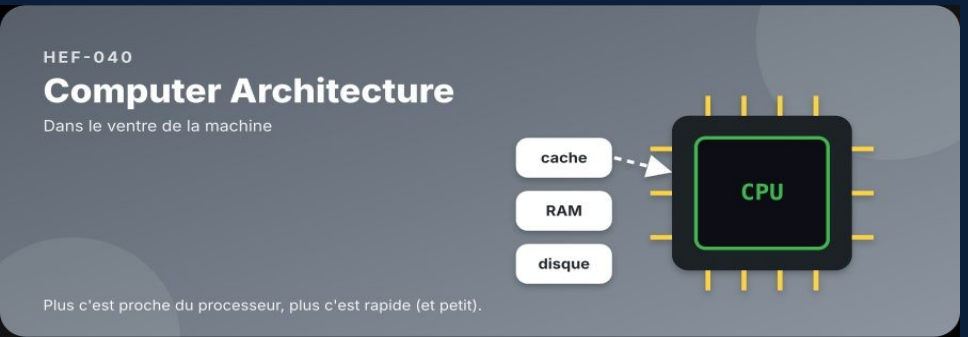
HEF-040 · MISSION 01 / 10

# Tout est bits : la représentation binaire

*Un ordinateur ne connaît que 0 et 1. Tout le reste (nombres, texte, images) est une convention.*



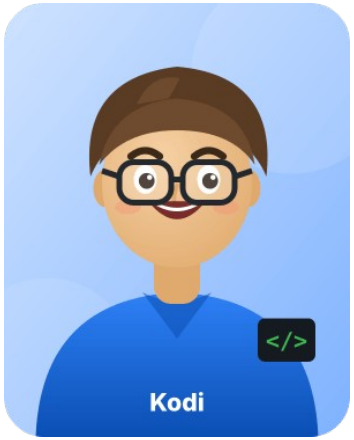

**Badge : Traducteur de bits**

70 min · débutant

# L'histoire

Kodi regarde un fichier avec un éditeur hexadécimal et voit `48 61 73 68`. Il demande : « C'est du chinois ? »

> *« C'est du texte, dit Sam. "Hash" en ASCII. L'ordinateur ne stocke ni lettres ni nombres : seulement des 0 et des 1. C'est nous qui décidons comment les lire. »*




**Dans cette mission, tu apprends à lire et écrire la langue maternelle des machines.**

Ta mission

# À la fin, je serai capable de…

1. convertir un nombre entre décimal, binaire et hexadécimal
2. expliquer ce qu'est un octet et la taille des entiers
3. représenter un entier négatif en complément à deux
4. expliquer comment le texte est codé (ASCII, UTF-8)
5. expliquer pourquoi 0,1 + 0,2 ≠ 0,3 en machine

# Défi d'échauffement

Tu as 5 interrupteurs alignés, chacun allumé (1) ou éteint (0). Combien de **combinaisons différentes** peux-tu former ? Et avec 8 interrupteurs ? Avec 10 ? Écris la règle que tu observes.

Le concept

# Le bit et l'octet

Un **bit** vaut 0 ou 1. Un **octet** (*byte*) regroupe **8 bits** et peut prendre $2^8 = 256$ valeurs (0 à 255).
Avec n bits : **$2^n$** combinaisons.

| Bits | Valeurs possibles |
|---|---|
| 1 | 2 |
| 8 | 256 |
| 16 | 65 536 |
| 32 | environ 4,3 milliards |
| 64 | environ $1{,}8 \times 10^{19}$ |

Le concept

# Le système binaire

En décimal, les positions valent 1, 10, 100… En binaire : **1, 2, 4, 8, 16…**

```
binaire   1 0 1 1
valeur    8 4 2 1   →  8 + 0 + 2 + 1 = 11
```

## L'hexadécimal

L'**hexadécimal** (base 16) utilise 0-9 puis A-F. **Un chiffre hexadécimal = 4 bits**, donc un octet s'écrit avec 2 chiffres. C'est un raccourci lisible pour le binaire.

| Décimal | Binaire | Hexa |
|---|---|---|
| 10 | 1010 | A |

Le concept

# L'hexadécimal (suite)

| Décimal | Binaire | Hexa |
|---|---|---|
| 15 | 1111 | F |
| 255 | 1111 1111 | FF |

## Opérations sur les bits

| Opération | Symbole Python | Exemple |
|---|---|---|
| ET | & | `0b1100 & 0b1010 = 0b1000` |
| OU | `\` | ` |
| OU exclusif | ^ | `0b1100 ^ 0b1010 = 0b0110` |
| Décalage gauche | << | `1 << 3 = 8` (multiplie par $2^3$) |
| Décalage droite | >> | `8 >> 1 = 4` |

Le concept

# Opérations sur les bits (suite)

```
print(bin(11), hex(255), int("1011", 2), int("ff", 16))
print(0b1100 & 0b1010, 0b1100 | 0b1010, 0b1100 ^ 0b1010)
print(1 << 3, 8 >> 1)
```

## Entiers négatifs : le complément à deux

Sur 8 bits, on utilise le bit de poids fort comme signe : 0 à 127 sont positifs, 128 à 255 représentent −128 à −1. Pour obtenir −x : **inverser tous les bits, puis ajouter 1**.

```
 5  = 0000 0101
-5  = 1111 1011   (inverser : 1111 1010, puis +1)
```

L'avantage : l'addition fonctionne telle quelle (5 + (−5) = 0 avec dépassement ignoré).

Le concept

# Texte

Un caractère est un **nombre** associé par une table.

- **ASCII** : 128 caractères (lettres sans accent, chiffres, ponctuation), sur 7 bits.
- **Unicode** : plus d'un million de caractères possibles.
- **UTF-8** : encodage d'Unicode qui utilise **1 à 4 octets** par caractère (ASCII reste sur 1 octet).

```
print(list("Hash".encode("utf-8")))     # [72, 97, 115, 104]
print(len("é".encode("utf-8")))         # 2
print(len("€".encode("utf-8")))         # 3
```

## Les flottants et leurs limites

Le concept

# Les flottants et leurs limites (suite)

Les nombres à virgule sont stockés en binaire. Beaucoup de décimaux (0,1) n'ont **pas** de représentation exacte en binaire, comme 1/3 n'en a pas en décimal.

```
print(0.1 + 0.2)              # 0.30000000000000004
print(0.1 + 0.2 == 0.3)       # False
```

Règle : ne compare jamais deux flottants avec `==`. Utilise une tolérance, et pour l'argent, des entiers (centimes).

# Exemple pas à pas

**Problème** : afficher un entier en binaire sur 8 bits, et son opposé en complément à deux.

```
def en_binaire(n, bits=8):
    return format(n & (2 ** bits - 1), f"0{bits}b")

def complement_a_deux(n, bits=8):
    inverse = (~n) & (2 ** bits - 1)
    return (inverse + 1) & (2 ** bits - 1)

for n in (5, 1, 100):
    print(n, en_binaire(n), "→", en_binaire(complement_a_deux(n)))
# 5 00000101 → 11111011
# 1 00000001 → 11111111
# 100 01100100 → 10011100

assert (5 + complement_a_deux(5)) & 0xFF == 0     # 5 + (-5) = 0
```

# Exemple pas à pas (suite)

**Lecture** : 11111011 vaut 251 en non signé, mais −5 en complément à deux. Le même motif de bits a deux lectures : tout dépend de la convention choisie. C'est l'idée centrale du module : **les bits n'ont pas de sens par eux-mêmes.**

**Overflow** : sur 8 bits, 200 + 100 donne 300 mod 256 = 44.

```
print((200 + 100) & 0xFF)   # 44
```

# À toi, avec aide

1. Convertis 37 en binaire (à la main), puis vérifie avec `bin(37)`.
2. Convertis `0x2F` en décimal et en binaire.
3. Combien d'octets faut-il pour stocker le texte `"Café"` en UTF-8 ? Vérifie en Python.

À toi, avec aide · réponse

# Indices progressifs

## Indice 1

37 = 32 + 4 + 1.

## Indice 2

Un chiffre hexadécimal = 4 bits : 2 = 0010, F = 1111.

## Indice 3

Le « é » n'est pas dans l'ASCII.

À toi, avec aide · réponse

# Réponse

```
print(bin(37))                      # 0b100101
print(int("2F", 16), bin(0x2F))     # 47 0b101111
print(len("Café".encode("utf-8")))  # 5
```

1. 37 = 32 + 4 + 1 → `100101`. 2. `2F` = 2 × 16 + 15 = 47 = `0010 1111`. 3. C, a, f = 3 octets, é = 2 octets → 5 octets.

# À toi, en solo

Une **couleur** est souvent stockée sur 3 octets (rouge, vert, bleu).
Écris :

1. `rgb_vers_hex(r, g, b)` qui renvoie la couleur au format `#RRGGBB` (ex. `(255, 128, 0)` → `#FF8000`) ;
2. `hex_vers_rgb(texte)` qui fait l'inverse ;
3. `eclaircir(texte, delta)` qui ajoute `delta` à chaque composante sans dépasser 255 (borne).

Teste avec au moins 5 couleurs, dont le noir et le blanc.

À toi, en solo · réponse

# Pistes de correction

```
def rgb_vers_hex(r, g, b):
    return "#{:02X}{:02X}{:02X}".format(r, g, b)

def hex_vers_rgb(texte):
    texte = texte.lstrip("#")
    return tuple(int(texte[i:i + 2], 16) for i in (0, 2, 4))

def eclaircir(texte, delta):
    r, g, b = hex_vers_rgb(texte)
    return rgb_vers_hex(min(255, r + delta), min(255, g + delta), min(255, b + del
   ta))

assert rgb_vers_hex(255, 128, 0) == "#FF8000"
assert hex_vers_rgb("#000000") == (0, 0, 0)
assert hex_vers_rgb("#FFFFFF") == (255, 255, 255)
assert eclaircir("#FFFFFF", 50) == "#FFFFFF"
assert eclaircir("#102030", 16) == "#203040"
print("ok")
```

À toi, en solo · réponse

# Pistes de correction (suite)

Chaque composante tient sur 1 octet : valeur 0 à 255, deux chiffres hexadécimaux.

# Pièges classiques

| Piège | Pourquoi c'est un problème | Comment l'éviter |
|---|---|---|
| **Comparer des flottants avec `==`** | 0,1 + 0,2 ≠ 0,3 | Utiliser une tolérance (`math.isclose`) |
| **Oublier le dépassement (overflow)** | 200 + 100 = 44 sur 8 bits | Connaître la taille des types |
| **Confondre octets et caractères** | « é » prend 2 octets en UTF-8 | Encoder explicitement |
| **Confondre valeur et représentation** | 11111011 est 251 ou −5 | Dire quelle convention on utilise |

Mini-boss

# Mini-boss

**Mini-bibliothèque de bits.** Écris un module `bits.py` :

1. `lire_bit(n, i)`, `mettre_bit(n, i)`, `effacer_bit(n, i)`, `basculer_bit(n, i)` ;
2. `compter_bits(n)` : nombre de bits à 1 ;
3. `est_puissance_de_deux(n)` en une seule expression avec `&` ;
4. `permissions` : représente des droits (lire = 4, écrire = 2, exécuter = 1) dans un entier, avec `donner(droits, d)`, `retirer(droits, d)`, `a_le_droit(droits, d)`, et `en_texte(droits)` (ex. 6 → `rw-`).

Teste tout avec des `assert`, dont des cas limites (0, grand nombre, bit 0, bit 31).

Mini-boss · réponse

# Pistes de correction

```
def lire_bit(n, i): return (n >> i) & 1
def mettre_bit(n, i): return n | (1 << i)
def effacer_bit(n, i): return n & ~(1 << i)
def basculer_bit(n, i): return n ^ (1 << i)
def compter_bits(n): return bin(n).count("1")
def est_puissance_de_deux(n): return n > 0 and (n & (n - 1)) == 0

LIRE, ECRIRE, EXEC = 4, 2, 1
def donner(droits, d): return droits | d
def retirer(droits, d): return droits & ~d
def a_le_droit(droits, d): return droits & d != 0
def en_texte(droits):
    return ("r" if droits & LIRE else "-") + ("w" if droits & ECRIRE else "-") + (
   "x" if droits & EXEC else "-")

assert lire_bit(0b100, 2) == 1 and lire_bit(0b100, 0) == 0
assert effacer_bit(mettre_bit(0, 31), 31) == 0
assert basculer_bit(5, 1) == 7
assert compter_bits(0) == 0 and compter_bits(255) == 8
```

Mini-boss · réponse

# Pistes de correction (suite)

```
assert est_puissance_de_deux(64) and not est_puissance_de_deux(0) and not est_puis
   sance_de_deux(12)
d = donner(donner(0, LIRE), ECRIRE)
assert en_texte(d) == "rw-" and a_le_droit(d, ECRIRE) and not a_le_droit(d, EXEC)
assert en_texte(retirer(d, ECRIRE)) == "r--"
print("ok")
```

Les permissions Unix fonctionnent exactement ainsi : un entier dont chaque bit est un droit.

Quiz éclair

# Quiz éclair

**1.** Combien de valeurs peut prendre un octet ?

A. 256
B. 8
C. 1 024

**2.** Un chiffre hexadécimal représente :

A. 8 bits
B. 4 bits
C. 2 bits

**3.** Le complément à deux sert à représenter :

A. les nombres à virgule

# Quiz éclair (suite)

B. le texte
C. les entiers négatifs

**4.** 🔁 *Révision HEF-000, mission 01* : Une hypothèse est :

A. une explication possible non vérifiée
B. un fait
C. une solution

**5.** 🔁 *Révision HEF-030, mission 02* : $O(n^2)$ signifie que si n est multiplié par 10, le travail est multiplié par :

A. 10
B. environ 100

Quiz éclair

# Quiz éclair (suite)

C. 1 000

Quiz éclair · réponse

# Réponses

1. **A**.
2. **B**.
3. **C**.
4. **A**.
5. **B**.

Je réfléchis

# Je réfléchis

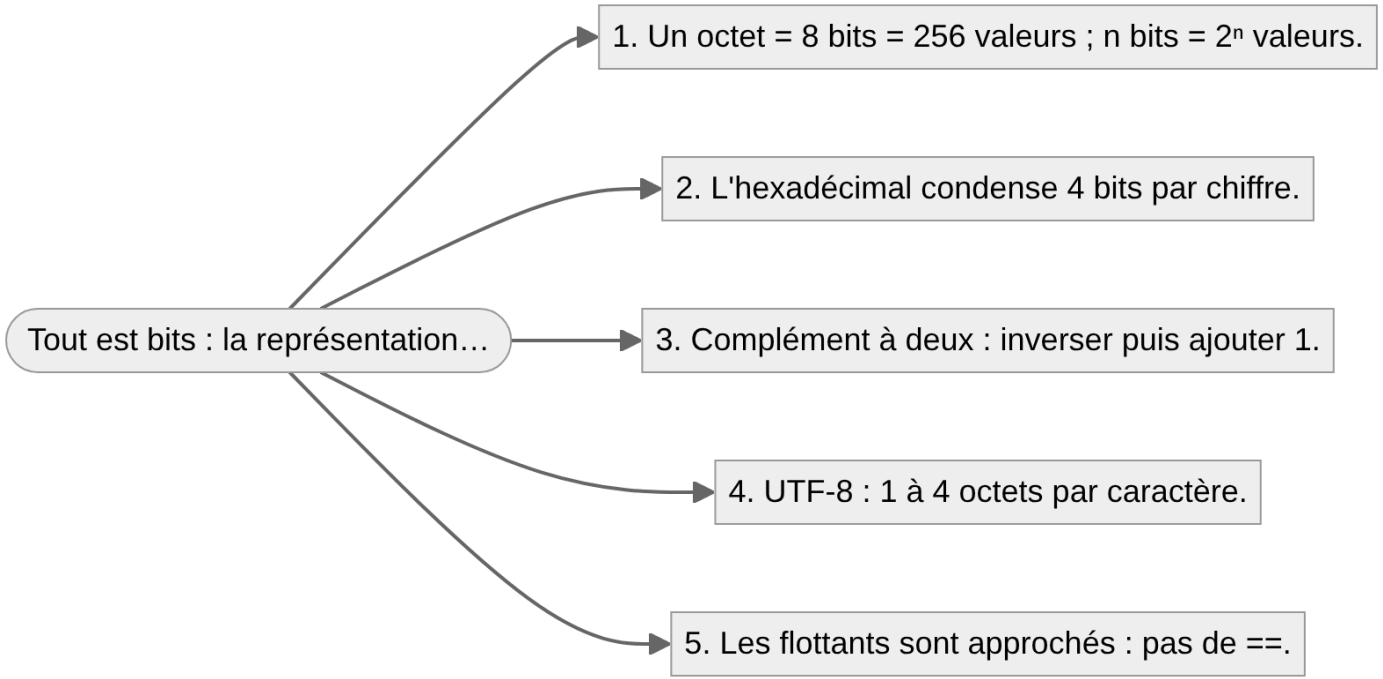
1. Où, dans ta vie numérique, as-tu déjà croisé de l'hexadécimal (couleurs, adresses, identifiants) ?
2. Pourquoi est-il important de savoir que les bits n'ont pas de sens par eux-mêmes ?

# À retenir

1. Un octet = 8 bits = 256 valeurs ; n bits = $2^n$ valeurs.
2. L'hexadécimal condense 4 bits par chiffre.
3. Complément à deux : inverser puis ajouter 1.
4. UTF-8 : 1 à 4 octets par caractère.
5. Les flottants sont approchés : pas de ==.

À retenir

# Le schéma de la mission



MISSION ACCOMPLIE

# +100 XP

## Badge débloqué : Traducteur de bits

Prochaine mission : Le CPU et son langage : le jeu d'instructions

