Python pour les débutants

Bienvenue! Vous êtes complètement novice en programmation? Heureusement, n'importe quel programmeur peut apprendre à programmer en Python. Il est facile à utiliser et à apprendre pour les débutants, alors lancez-vous !

```
def exemple():
    print("Bonjour, le monde!")
```

Pourquoi Python?

Python est un langage de programmation populaire. Il est utilisé pour de nombreux types de projets, tels que:

- Développement Web
- Analyse de données
- Intelligence artificielle
- Automatisation
- Et bien plus encore!

Installation de Python

Table des matières

Installation sur Windows

Étape 1 : Télécharger Python

1. Rendez-vous sur le site officiel de Python.
2. Téléchargez l'installateur pour Windows (la dernière version stable).

Étape 2 : Installation de Python

1. Exécutez l'installateur que vous venez de télécharger.
2. **Important :** Cochez la case "Add Python to PATH" avant de cliquer sur "Install Now".
3. Cliquez sur **Install Now** et suivez les instructions.

Étape 3 : Vérifier l'installation

1. Ouvrez l'invite de commande (Windows + R, tapez `cmd`).
2. Tapez `python --version` pour vérifier la version installée.

Installation sur Linux

Étape 1 : Mettre à jour les packages

Avant d'installer Python, assurez-vous que les paquets sont à jour :

```
sudo apt update && sudo apt upgrade
```

Étape 2 : Installer Python

Pour Ubuntu ou Debian, utilisez la commande suivante :

```
sudo apt install python3
```

Pour Fedora :

```
sudo dnf install python3
```

Pour Arch Linux :

```
sudo pacman -S python
```

Étape 3 : Vérifier l'installation

Tapez `python3 --version` pour vérifier la version installée.

Installation sur macOS

2 méthodes sont possibles pour installer Python sur macOS :

Via l'installateur officiel. (1ère méthode)

Étape 1 : Télécharger Python

1. Rendez-vous sur le site officiel de Python.

2. Téléchargez l'installateur pour macOS (la dernière version stable).

Étape 2 : Installation de Python

1. Ouvrez le fichier `.pkg` téléchargé.
2. Suivez les instructions de l'installateur.

Étape 3 : Vérifier l'installation

1. Ouvrez le terminal.
2. Tapez `python3 --version` pour vérifier la version installée.

Via Homebrew. (2eme méthode)

Étape 1 : Installer Homebrew

Si Homebrew n'est pas installé, vous pouvez l'installer en exécutant cette commande dans le terminal :

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/ins
```

Étape 2 : Installer Python

Avec Homebrew, installez Python en exécutant la commande suivante :

```
brew install python
```

Étape 3 : Vérifier l'installation

Une fois l'installation terminée, vérifiez la version de Python :

```
python3 --version
```

Apprendre Python pour Développeurs Débutants

Table des matières

Introduction à Python

Python est un langage de programmation puissant et facile à lire, avec une syntaxe simple qui en fait un excellent choix pour les débutants. Utilisé dans divers domaines comme le développement web, l'analyse de données, et l'intelligence artificielle, il est aussi apprécié pour sa large communauté et ses nombreuses bibliothèques.

Premiers Pas

Voici un exemple de programme simple en Python, le célèbre "Hello, World!".

```
print("Hello, World!")
```

Python exécute les instructions ligne par ligne, et chaque instruction doit être correctement indentée.

Les Bases de Python

Variables et Types de Données

En Python, il n'est pas nécessaire de déclarer explicitement les types de variables, ils sont déduits automatiquement.

```
# Exemple de types de données
x = 5          # entier
y = 3.14       # flottant
name = "Alice" # chaîne de caractères
is_valid = True # booléen
```

Les principaux types de données incluent :

int : nombres entiers
float : nombres à virgule flottante
str : chaînes de caractères
bool : booléens (True ou False)

Opérateurs

Python prend en charge les opérateurs arithmétiques, de comparaison et logiques. Voici quelques exemples :

```
# Opérateurs arithmétiques
a = 10
b = 3
print(a + b)  # Addition
print(a - b)  # Soustraction
print(a * b)  # Multiplication
print(a / b)  # Division
print(a % b)  # Modulo (reste de la division)

# Opérateurs de comparaison
print(a > b)  # True
print(a == b) # False
```

Structures Conditionnelles

Les conditions permettent de prendre des décisions dans le code.

```
x = 10
if x > 5:
    print("x est supérieur à 5")
elif x == 5:
    print("x est égal à 5")
else:
    print("x est inférieur à 5")
```

Boucles

Les boucles permettent de répéter des instructions.

La boucle `for`

```
for i in range(5):
    print(i)  # Affiche les nombres de 0 à 4
```

La boucle `while`

```
x = 0
while x < 5:
    print(x)
    x += 1  # Incrémente x
```

Fonctions

Les fonctions permettent de regrouper du code réutilisable. Elles sont définies à l'aide du mot-clé def.

```
def saluer(nom):
    return f"Bonjour, {nom}!"

print(saluer("Alice"))  # Appel de la fonction
```

Structures de Données

Listes

Les listes sont des collections ordonnées et modifiables.

```
fruits = ["pomme", "banane", "cerise"]
print(fruits[0])  # Accéder au premier élément
fruits.append("orange")  # Ajouter un élément
```

Tuples

Les tuples sont des collections ordonnées mais non modifiables.

```
point = (4, 5)
print(point[0])  # Accéder au premier élément
```

Dictionnaires

Les dictionnaires stockent des paires clé-valeur.

```
personne = {"nom": "Alice", "âge": 25}
print(personne["nom"])  # Accéder à une valeur via la clé
```

Programmation Orientée Objet

Python est un langage orienté objet, ce qui signifie que vous pouvez créer des **classes** et des **objets**.

```
class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def se_presenter(self):
        return f"Je m'appelle {self.nom} et j'ai {self.age} ans."

# Instanciation d'un objet
alice = Personne("Alice", 25)
print(alice.se_presenter())
```

Concepts clés :

- Classe : modèle pour créer des objets.
- Objet : instance d'une classe.
- Méthode : fonction définie dans une classe.
- Attributs : variables associées à un objet.

Gestion des Exceptions

Les exceptions permettent de gérer les erreurs dans votre programme. Vous pouvez les attraper à l'aide des blocs `try-except` .

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Erreur : division par zéro.")
```

Cela permet d'éviter que le programme ne s'arrête brusquement.

Ressources Supplémentaires

- Documentation officielle de Python
- W3Schools Python Tutorial
- Real Python
- Codecademy - Learn Python

Avec ces bases, vous êtes prêt à explorer plus en profondeur le langage Python et à développer vos propres projets. Bon apprentissage !

Conventions et bonnes pratiques

Cette page répertorie toutes les conventions que nous appliquerons aux développements impliquant du Python,
elles permettront d'uniformiser notre code afin de faciliter la reprise, la maintenance et la compréhension par nos successeurs.
La documentation qualité s'appuiera sur la PEP 8 afin de la rendre la plus générique et évolutive possible.Tout en ajoutant
et en modifiant certains éléments. (PEP 8)

La PEP 8

Introduction

La PEP 8 est un document écrit par les créateurs de Python, il contient les conventions de codage Python les plus répendues de nos jours,
elle est régulièrement mise à jour et disponible pour tous.
Il en existe également une déclinaison pour la langage C, mais nous ne la détaillerons pas ici.
L'objectif principal des normes est de rendre le code plus lisible et plus simple à reprendre,
on n'a pas forcément envie de reprendre 3000 lignes empactées sans instructions, pas vrai ? 😉
Voici donc quelques crédos qui pourront vous guider dans votre reflexion : PEP 20

Application

Le plus important lors de l'écriture d'un code est l'assiduité, et encore plus lors de l'écriture d'une fonction ou d'un petit bout de code,
c'est à ce moment qu'il faudra la rendre la plus générique possible pour qu'elle soit potentiellement réutilisable dans des projets futurs ou même dans le même projet,
c'est toujours plaisant de pouvoir réutiliser quelque chose qui marche bien sans avoir à tout réécrire !
Cependant dans certains cas ces recommendations ne sont pas forcément à appliquer surtout si le projet ne s'y prête pas,
ou bien dans le cas de l'utilisation d'un framework particulier qui nécessiterait une syntaxe particulière (exemple: décorateurs des routes Flask)

Quelques exemples de cas où l'application des principes pourrait être à éviter:

- Suivre le guide rendrait le code moins lisible
- Le code autours suit une autre convention

- La version de Python utilisée dans le reste du code n'est pas compatible avec la nouvelle syntaxe

Structuration du code

Indentation

L'indentation des lignes se fait à l'aide des espaces, **1 indentation = 4 espaces**
Si vous utilisez VSCode pour travailler, vous pouvez définir ce paramètre dans la barre d'outils tout en bas : Spaces > Change tab display size > 4.

Les fonctions

Lors de l'écriture d'une fonction ou d'un appel fonction, les arguments peuvent prendre énormément de place sur la ligne,
nous verrons plus tard la limite à respecter, en attendant, lorsque l'appel ou la liste d'arguments est trop longue,
les arguments doivent être placés à la ligne:

```python
# On peut aligner les arguments sur la paranthèse d'ouverture
def nom_de_fonction_long(param_1, param_2,
                         param_3, param_4):
    print(param_1)

resultat = nom_de_fonction_long(param_1, param_2,
                                param_3, param_4)

# Ou alors
# On placera les arguments en dessous avec une indentation supplémentaire par rapport à
# Afin de différencier les arguments du code
def nom_de_fonction_long(
        param_1, param_2, param_3,
        param_4):
    print(param_1)

resultat = nom_de_fonction_long(
    param_1, param_2, param_3,
    param_4)

# Ou:
resultat = nom_de_fonction_long(
    param_1, param_2,
    param_3, param_4
)
```

Les conditions

Lors de la constitution d'une condition, il se peut qu'il y ait plusieurs "and" ou "or", cela peut rendre la ligne particulièrement longue,
ainsi nous ferons le découpage de la manière suivante:

```python
# Chaque 'and' ou 'or' sera sur une nouvelle ligne 4 espaces plus loin que la parenthès
if (condition1
        and condition2
        and condition3
        or condition4):
    return
```

Les listes et dictionnaires

Afin de conserver une forme qui reste lisible lors de la définition d'une liste ou d'un dictionnaire, s'il s'agit d'un objet à plusieurs dimensions,
on gardera toujours une forme cohérente avec la forme réelle qu'aura l'objet.

```
# Dans le cas d'une liste à 1 dimension:
ma_liste = [
    1, 2, 3,
    4, 5, 6,
    7, 8, 9
]

# 2 dimensions:
ma_liste = [
    [1, 2, 3, 4, 5],
    [6, 7, 8, 9, 10]
]

# 3 dimensions:
ma_liste = [
    [
        [1, 2, 3],
        [4, 5, 6]
    ],
    [
        [7, 8, 9],
        [10, 11, 12]
    ]
]
# Et ainsi de suite..

# Pour les dictionnaires :
mon_dico = {
    '1': 'oui',
    '2': 'stiti'
}
```

Longueur des lignes à respecter

Quand on lit un code, c'est pas marrant de scroller à l'infini à l'horizontale !
Pour éviter ça, on va limiter la longueur des lignes de code de manière à tout garder visible à l'écran pour un maximum de résolutions différentes,
il a donc été décidé que la longueur parfaite était de **100 caractères maximum par ligne de code**
Pour ce qui est des **commentaires ou des docstrings**, on se limitera à **72 caractères**
La seconde raison de cette limite est la possibilité d'avoir plusieurs éditeurs de code ouverts simultanément sur un même écran, en les mettant côte à côte,
avoir des lignes pas trop longues permet d'avoir une vue d'ensemble de tous les codes que nous souhaitons voir, ainsi pour les review ou même pour coder sur
différentes composantes en même temps, il est plus simple de se restreindre.

Pour éviter de dépasser cette limite, on peut faire comme vu précédemment, lorsqu'on est dans une parenthèse on peut passer à la ligne. En revanche dans le cas
d'un "with" avec plusieurs ouvertures, ou d'un "assert", l'utilisation du **** peut être utile. Voici un exemple d'utilisation:

```
# Chaque 'and' ou 'or' sera sur une nouvelle ligne 4 espaces plus loin que la parenthè
with open('/chemin/vers/un/fichier') as fichier_1, \\
        open('/chemin/vers/un/fichier', 'w') as fichier_2:
    fichier_2.write(fichier_1.read())
```

Les opérateurs

Vous avez beaucoup de calculs à faire ? Vous n'allez tout de même pas tout mettre sur la même ligne après tout ce qu'on vient de dire ? 😉
Pour chaque bloc d'opération, nous séparerons en faisant un saut à la ligne et en mettant l'opérateur en début de ligne comme ceci:

```
# L'opérateur sera aligné avec la parenthèse d'ouverture
revenu = (salaire
        - budget_codage
        + prime_de_votre_gentil_chef
        * 1.1) #Pour le plaisir
```

Les sauts de lignes

C'est bien connu, un gros bloc, ça donne pas envie de lire.
On va donc espacer notre code en sautant des lignes **aux bons endroits** et sans en abuser pour ne pas retirer la cohérence du code.

```python
# 1. Un saut de ligne entre les fonctions
def func_1():
    return

def func_2():
    return

# 2. Deux sauts de ligne entre les méthodes de classe et les fonctions
def methode(self):
    return


def fonction():
    return

# 3. Un saut de ligne entre un code et une condition
variable = 1
test = 2

if variable == 1:
    return

# 4. Dans les fonctions, 1 saut de ligne entre chaque bloc logique
def fonction():
    variable = 1
    autre_variable = 1

    with open('test') as test:
        print(test)

    return
```

Les imports

Les imports devront toujours se situer **au début du code après le header de documentation**, jamais plus loin. Il faudra les découpler sur
plusieurs lignes s'ils ne sont pas liés entre eux.
Ils devront être importés dans l'ordre suivant:

1. Librairies standard Python
2. Librairies tierces
3. Modules internes au code

```
# Correct:
import os
import sys

# Dans le cas d'imports venant d'un même package, il est recommandé de les mettre sur
from subprocess import Popen, PIPE

# Les imports absolus sont recommandés quand la structure du projet est complexe
from . import voisin
from .voisin import fonction
```

La documentation du module

Au début d'un module, avant tout code, il faut bien écrire la documentation afin de savoir tout de suite quelle est son utilité ou encore à qui
se référer s'il y a un problème ou des questions.
Cette documentation fait partie du **"docstring"** qui est un peu la "carte d'identité" du projet, on va documenter chaque module et chaque fonction.

Tout d'abord pour les modules, la documentation se présentera comme ceci:

```
"""Module d exemple

Description de la fonction principale du module, ex: Permet d exporter des données au
"""

from __future__ import annotations

__all__ = ["variable1", "variable2", "variable3"]
__version__ = "1.0"
__author__ = "Alex Lovergne"

import os
import sys

def fonction_exemple(arg1: String, arg2: String) -> String:
    """Fonction d exemple, définition de son utilité
        args:
            - arg1 : String -> Première chaine de caractère
            - arg2 : String -> Deuxième chaine de caractère à concaténer
        returns: String -> Concatenation des deux chaines
    """
    return arg1 + arg2
```

La variable __all__ permet de définir le nom des variables qui seront exportées du module, par exemple si on a une variable **a = 0**
définie dans le module et que l'on importe le module ailleurs, seule la variable **a** sera importée.

Les espaces

Les espacements dans le code sont très utiles pour améliorer sa lisibilité et le rendre plus aéré. **On aime bien avoir de la place entre son canapé**
et sa télévision pour bien voir, dans le code c'est pareil.
Il est important toute fois de ne pas mettre des espaces partout, il y a des emplacements bien définis pour faciliter la lecture et la détection
rapide de certaines instructions ou syntaxes.

```
# Entre les parenthèses et les valeurs, pas d'espaces
acheter(viande[1], {oeufs: 2})
exemple = (0,)

# 1 espace après les ':' et les ',' mais pas avant.
# Jamais d espaces entre le nom de fonction et ses arguments. Entre une liste et ses c
if x == 4: print(x, y); x, y = y, x
def fonction_exemple(arg1: String, arg2: String) -> String:
dct['key'] = lst[index]

# Toujours un espace de chaque côté du '=', '+', '-', '/', '->'
i = i + 1
ajout += 1
x = x*2 - 1
hypot2 = x*x + y*y + y / 2
c = (a+b) * (a-b)
```

Les commentaires

Rapide point sur les commentaires, vous devez en avoir marre, les commentaires c'est long et barbant à faire mais malheureusement
c'est ce qui va aider votre prochain (on est pas à l'église rassurez vous).
Ainsi écrire des commentaires pour expliquer les points importants du code sont primordiaux surtout s'il s'agit d'un traitement métier qu'un
autre développeur pourrait ne pas maîtriser.
Attention cependant à les utiliser avec modération, **Un commentaire à chaque ligne, c'est comme de la moquette sur un mur, c'est dérangeant**
Vous pouvez les mettre sur une nouvelle ligne ou bien en fin de ligne à 2 espaces d'intervalle de la

fin de la ligne de code.
N'oubliez pas la limite de caractères par ligne !

Les conventions

Nommage

Les variables et fonctions

Les variables doivent **toujours avoir un nom explicite**, il faut éviter les variables ayant un nom comme "a" ou "b", mais plutôt "liste*mots*" *ou* "*dico_noms*" *par exemple.*
*Ensuite, il y a des conventions sur le nommage des variables, en effet on privilégiera les **variables en lettres minuscules, s'il y a plusieurs mots on les séparera*
avec un "", de la forme : liste_mots_a_trier** (c'est un exemple, ne faites pas des noms aussi longs).
Pour ce qui est des fonctions, nous partirons sur le même principe, il s'agit du snake_case.

Les classes et modules

Les noms de classes et les modules ont une syntaxe différente, au lieu d'utiliser des lettres en minuscule et des tirets du bas, on va utiliser le **CamelCase**.
Il s'agit de nommer en collant les mots, à chaque nouveau mot on met une majuscule de cette manière: JeSuisUnNomDeVariable (pas aussi long svp).

Les constantes

Les constantes seront toujours définies au début du script après la documentation et les imports.
Ou alors dans les fichiers de config,
ils seront définis en **majuscules, les mots seront séparés par des "_"**
De la manière suivante: JE_SUIS_UNE_CONSTANTE

> **Petit rappel: Une constante est une variable qui sera définie lors de l'écriture du code et ne sera JAMAIS modifiée après.**

Configuration déportée

Quand on a de l'argent liquide, on se promène jamais avec tout sur nous ? Dans le code c'est pareil, on ne veut pas que nos mots de passes soient visibles pour tous les gens
qui auraient accès à notre repo git.
On n'a pas non plus envie d'aller modifier partout si un mot de passe ou une valeur change ?
C'est pour ça qu'on va déporter notre configuration, de cette manière on évite tous les problèmes occasionnés par les chaines écrites en dur dans le code, mais aussi
on renforce la sécurité de notre programme étant donné qu'on ne publiera pas le fichier de config

rempli mais plutôt une version d'exemple pour aiguiller une personne
qui voudrait reprendre notre code mais qui n'aurait pas les mêmes accès que nous.

Mais comment on fait ?
C'est très simple, dans le dossier racine de notre projet, on va créer un nouveau dossier **settings**. Dans ce dossier nous allons créer un fichier **config.ini** et **config_exemple.ini**. Dans le fichier config.ini nous allons placer nos variables
qui seront accessibles depuis notre code comme ceci :

```
[exemple_1]
host=srv-exemple1
database=microcapteurs
user=exemple
password=exemple

[exemple_2]
host=srv-exemple2
database=microcapteurs
user=exemple2
password=exemple2
```

Ensuite, dans notre fichier **config_exemple.ini** nous allons placer exactement les mêmes variables mais en renseignant de **fausses informations**.

Comment récupérer nos variables de configuration ?
C'est très simple, dans notre programme Python nous allons utiliser le module **"configparser"**, il s'utilise comme ceci:

```
import configparser

config = configparser.ConfigParser() # Initialisation du module
config.read('./settings/config.ini') # Import de notre configuration

print(config['exemple_1']['host']) # On peut récupérer notre configuration à la manièr
print(config['exemple_2']['user'])
```

Logging

Il est toujours important de garder une trace de l'exécution de notre programme afin de faciliter son debuggage en cas de problème. Pour cela nous n'utiliserons évidemment
pas la fonction print mais bien la **librairie logging**
Cette librairie Python permet de paramétrer une ou plusieurs sorties pour ce qu'on voudrait

afficher/stocker. Il est ainsi possible par exemple pour un logging.info()
d'enregistrer le résultat dans un fichier .log mais en même temps de l'afficher dans la console.

Comment l'utiliser ?

Commençons tout d'abord par créer un fichier **log-config.ini** qui sera situé dans un dossier settings à la racine du programme. Dans ce fichier nous plaçerons ce contenu:

```
[loggers]
keys=root

[handlers]
keys=consoleHandler,fileHandler

[formatters]
keys=default

[logger_root]
level=NOTSET
handlers=consoleHandler,fileHandler

[handler_consoleHandler]
class=StreamHandler
level=NOTSET
formatter=default
args=(sys.stdout,)

[handler_fileHandler]
class=handlers.RotatingFileHandler
level=NOTSET
args=('debug.log',2000,100)
formatter=default
encoding=utf-8

[formatter_default]
format=%(asctime)s %(levelname)s %(message)s
datefmt=
style=%
validate=True
class=logging.Formatter
```

Il s'agira de notre configuration standard qui sera utilisée pour tous nos projets, elle permet de log dans la console mais aussi dans un fichier debug.log à la racine.
Le format de sortie sera du type: **YY-MM-DD H:M:S,MS INFO message affiché**

Maintenant que notre fichier de config est créé, nous pouvons l'importer dans notre programme principal, et en suite utiliser la bibiothèque logging.

```
import logging
import logging.config

logging.config.fileConfig('settings/log-config.ini')  # Chargement de la config -> 1 s

logging.info('information') # Pour afficher une étape par exemple
logging.warning('attention') # Exemple: condition non respectée mais non bloquante
logging.error('erreur') # Erreur, fin du programme
```

Dépendances

Qui n'a jamais rêvé de tout installer en une seule commande ? C'est bien pour ça que les requirements existent et sont utilisés.
Ils vont permettre d'installer toutes les dépendances de notre projet avec une unique commande, ce qui permet d'éviter les oublis, de fixer une
version bien définie avec laquelle le programme est compatible mais aussi de gagner du temps et éviter les potentielles failles de sécurité.

Mais comment qu'on fait ?
C'est très simple, il suffit de créer un fichier **requirements.txt** à la racine du projet.
Dans ce fichier nous allons lister nos dépendances ainsi que les versions de cette manière:

```
numpy==1.24.2
requests==2.28.2
urllib3==1.26.14 # Module == version
```

Ensuite une fois que c'est fait, on a déjà fini.
Pour **installer les dépendances** il faut taper la commande: **pip install -r requirements.txt**

Modularisation

On ne veut pas avoir tout notre code dans un seul fichier, c'est moche de scroll à l'infini pas vrai ?
On va donc chercher à découpler un maximum notre code en plusieurs modules, dans l'idéal 1 module par utilité.

Exemple:
J'ai un fichier **main.py** avec 6 fonctions, une pour exporter des données dans une BDD, une autre

pour importer les données d'une source (microcapteur par exemple),
et 4 autres pour faire des conversions.

Comment organiser mon code ?
Une proposition pourrait être de l'organiser de la manière suivante:
On va s'assurer que chaque module de données soit générique, pour se faire nous utiliserons des **interfaces** (Sensor.py)

```
.
├── main.py
├── requirements.txt
├── debug.log
├── settings/
│   ├── config.ini
│   ├── config_example.ini
│   └── log-config.ini
├── utils/
│   └── converters.py
├── database/
│   └── Database.py
└── sensors/
    ├── Kunak.py
    └── Sensor.py
```

Les interfaces

Qu'est-ce qu'une interface ? Une interface est une classe Python qui permet de définir un cadre qui sera appliqué aux classes héritantes.
Mais ça veut dire quoi ?

Voici un exemple simple:

- J'ai une interface Sensor
- J'ai une classe Kunak

La classe Sensor se définit de la manière suivante:

```
class Sensor:
    def __init__(self, name: str, constructor: str) -> None:
        self._name = name
        self._constructor = constructor

    @property
    def name(self) -> str:
        return self._name

    @name.setter
    def name(self, new_name: str):
        self._name = new_name

    @property
    def constructor(self) -> str:
        return self._constructor

    @constructor.setter
    def constructor(self, new_constructor: str):
        self._constructor = new_constructor

    def __repr__(self) -> str:
        return "<sensor %s - constructor:%s>" % (
            self._name, self._constructor)
```

La classe Kunak, quant a elle est définie comme ceci:

```
class Kunak(Sensor): # Hérite de l'interface Sensor
    def __init__(self, *args) -> None:
        super().__init__(*args) # Charge la classe parent
```

Ainsi, si on créé un objet de la classe Kunak on aura:

```
kun = Kunak('K100', 'Kunak')
print(kun)

>>> <sensor K100 - Kunak>
```

La classe de l'objet sera donc 'Kunak', héritant de Sensor. Elle bénéficie donc de toutes les méthodes et attributs définits dans la classe parent.

Boîte à outils

SQL Alchemy

SQL Alchemy est un **module Python qui sert à intéragir avec la base de données**, on appelle ça un **ORM** ou **Object-Relational Mapping**. Il s'agit d'un
type de programme informatique qui se place en interface entre un programme applicatif et une base de données relationnelle pour simuler une base de données orientée objet.
(Voir wikipédia)

Pour utiliser SQL Alchemy et le relier à notre base de données, il est nécessaire de créer nos **modèles**.

Qu'est-ce qu'un modèle ? Un modèle dans notre cas est une classe Python portant le nom d'une table de notre base de données, et qui répertorie les colonnes dont on va avoir besoin.
Elles doivent exister dans la BDD. Dans le cas où la BDD n'existerait pas encore, les tables seront créées en fonction des modèles définis dans le code.
Voici un exemple de modèle:

```python
import sqlalchemy as sa

from dbModels.Engine import Base

class Capteurs(Base):

    __tablename__ = 'capteurs'

    id = sa.Column(sa.Integer, nullable=False, primary_key=True)
    constructeur = sa.Column(sa.VARCHAR)
    reference_constructeur = sa.Column(sa.VARCHAR)
    position_actuelle = sa.Column(sa.VARCHAR)
    modele = sa.Column(sa.VARCHAR)
    notes = sa.Column(sa.Text)
    campagne_mesure_actuelle_id = sa.Column(sa.Integer)
    date_mise_en_service = sa.Column(sa.Date)
    date_arret = sa.Column(sa.Date)
    etat = sa.Column(sa.VARCHAR)

    def __init__(self, id, constructeur, reference_constructeur, position_actuelle,
                 modele, notes, campagne_mesure_actuelle_id, date_mise_en_service,
                 date_arret, etat):

        self.id = id
        self.constructeur = constructeur
        self.reference_constructeur = reference_constructeur
        self.position_actuelle = position_actuelle
        self.modele = modele
        self.notes = notes
        self.campagne_mesure_actuelle_id = campagne_mesure_actuelle_id
        self.date_mise_en_service = date_mise_en_service
        self.date_arret = date_arret
        self.etat = etat

    def __repr__(self):
        return "<capteur n°%s - %s - reference -> %s>" % (
            self.id, self.constructeur, self.reference_constructeur)
```

Et l'engine qui représente la base de notre connexion à la BDD est difini de la manière suivante:

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

import configparser
config = configparser.ConfigParser()
config.read("./settings/config.ini")

def loadSession():
    metadata = Base.metadata
    Session = sessionmaker(bind=engine)
    session = Session()
    return session

engine = create_engine('postgresql+psycopg2://%s:%s@%s:5432/%s' %
                           (config['postgresql']['user'],
                           config['postgresql']['password'],
                           config['postgresql']['host'],
                           config['postgresql']['database']),
                       )

Base = declarative_base()
```

La variable **Base** définie ici servira à **instancier tous les modèles de la BDD par héritage**.

Flask

Flask est un framework Python qui permet de créer des sites web et API. Il ne s'agit pas forcément du framework le plus simple, surtout quand on en vient à la gestion du contexte, mais il reste très efficace et rapide. De plus il est léger et accompagné de plusieurs extensions avec des modules déjà existants.

Pour les projets en Flask, la structure la plus pratique et qui nous intéresse le plus en terme de qualité d'architecture est celle ci:

```
.
├── requirements.txt
├── Dockerfile
├── server-conf/
│   ├── nginx.conf
│   ├── uwsgi.ini
│   ├── supervisord.conf
│   └── flask-site-nginx.conf
└── app/
    ├── __init__.py
    ├── conf/
    │   ├── config.ini
    │   └── log-config.ini
    ├── donnees/
    │   ├── __init__.py
    │   └── routes.py
    ├── media/
    │   ├── __init__.py
    │   └── routes.py
    ├── polluant/
    │   ├── __init__.py
    │   └── routes.py
    ├── exploitation/
    │   └── debug.log
    ├── models/
    │   ├── Donnees.py
    │   ├── Medias.py
    │   ├── Polluant.py
    │   └── Engine.py
    └── utils/
        └── authUtils.py
```

Le __init**.py dans app/ va servir à instancier tous les blueprints et les enregistrer dans l'application, mais aussi à créer l'application en elle-même, définir son port d'accès, etc.
Les blueprints enregistrés dans le init à la racine proviendront des sous-dossiers. Chaque sous-dossier non commun correspond à un endpoint.
Chaque endpoint aura son blueprint qui est défini dans le __init**.py de chaque sous-dossier.
Ensuite les routes liées à ce blueprint sont créées dans le fichier routes.py
puis importées dans le __init__.py du sous-dossier.
Une chaine classique serait donc:

```python
# Finier __init__.py dans app/

from flask import Flask, jsonify, Blueprint
from flask_cors import CORS
from .donnees import dataBP
from .conf.config import Config
import logging, logging.config

def create_app(config_class=Config):
    app = Flask(__name__)
    app.config.from_object(config_class)

    app.register_blueprint(dataBP) #routes : /donnees, /donnees/<id>

    logging.config.fileConfig("/project/src/backoffice/app/conf/log-config.ini")

    cors = CORS(app, supports_credentials=True)

    bp = Blueprint('api', __name__, url_prefix="/api")

    return app

app = create_app()

if __name__ == '__main__':
    create_app().run(port=5001)


# Fichier __init__.py dans app/donnees/

from flask import Blueprint

dataBP = Blueprint('donnees', __name__, url_prefix="/api/donnees")

from . import route
```

```python
# Fichier routes.py dans app/donnees/

from flask import jsonify, request, current_app
from ..utils.authUtils import token_required
from . import dataBP # Import du blueprint pour déclarer nos routes

@dataBP.route('/', methods=['GET']) #Route de base /api/donnees -> toutes les données
@token_required
def getDonnees(user):
    return

@dataBP.route('/<int:id>', methods=['GET']) #Donnée spécifique
@token_required
def getDonnee(user, id):
    return

@dataBP.route('/', methods=['POST'])
@token_required
def setDonnee(user): #MAJ ou création d'une donnée
    return
```