Introduction à Python

Etienne Dagorn

Université de Lille – LEM

Qui suis-je ?

⇒ **Nouveau maître de conférences** à l'Université de Lille ;

⇒ **Thèmes de recherche** : économie expérimentale, économie de l'éducation ;

⇒ **Parcours académique** :

- Doctorat en économie à l'Université de Rennes 1 ;
- Postdoctorat à l'INED (Institut national d'études démographiques).

⇒ **Expériences d'enseignement** :

- Méthodes quantitatives, économétrie appliquée ;
- Économie de l'éducation, politiques publiques.

Objectifs du cours

Introduction pratique à la programmation en Python et ses applications

1. Acquérir les bases de la programmation (variables, boucles, fonctions).
2. Manipuler des jeux de données avec *pandas*.
3. Visualiser des données avec *matplotlib*.
4. Être capable d'écrire des scripts simples pour analyser des données économiques.

Pourquoi ce cours ?

⇒ Quel est l'intérêt de ce cours ?

⇒ "faire parler les données" + visualiser ⇒ Diffusser la connaissance

Pourquoi ce cours ?

⇒ Quel est l'intérêt de ce cours ?

⇒ "faire parler les données" + visualiser ⇒ Diffusser la connaissance

Data Scientist: The Sexiest Job of the 21st Century

Meet the people who can coax treasure out of messy, unstructured data. by Thomas H. Davenport and DJ Patil

From the Magazine (October 2012)

Andrew J Buboltz, silk screen on a page from a high school yearbook, 8.5" x 12", 2011 Tamar Cohen

Source : Harvard Buisness Review

Aspects pratiques

⇒ Contact uniquement par mail : etienne.dagorn@univ-lille.fr

⇒ Réponse uniquement le lundi matin ;

⇒ Nécessaire pour avoir une réponse :

- Mail avec formules de politesse
- sujet du mail clair + indication sur le groupe/cours

Organisation

⇒ 12 séances de cours pendant le semestre

⇒ Introduction au début du cours + mise en pratique ensuite ;

⇒ Rendu du script sur moodle à **maximum** 23 :59 à j+1

⇒ Evaluation à la dernière séance

Cours 1

Introduction à la programmation

Objectifs de la séance

1. Comment **penser** comme une **machine** ?
2. Comprendre fichiers, dossiers et chemins (relatif vs absolu).
3. **Parler au terminal** : se déplacer, lancer Python/Jupyter.
4. Organiser un **mini-projet** (data / notebooks / src / outputs).

Qu'est-ce que la programmation ?

Qu'est-ce que la programmation ?

⇒ **But** : décrire **précisément** une suite d'actions qu'une machine exécute.

⇒ **Cycle** : **Problème** → **Modélisation** → **Algorithme** → **Programme** → **Résultat**.

⇒ **Algorithme** : procédure finie, non ambigüe, reproductible.

⇒ **Programme** : implémentation concrète (peu importe le langage).

⇒ **IO** (Input/Output) : fichiers, clavier, réseau, écran, figures...

Qu'est-ce que la programmation ?

⇒ **But** : décrire **précisément** une suite d'actions qu'une machine exécute.

⇒ **Cycle** : **Problème** → **Modélisation** → **Algorithme** → **Programme** → **Résultat**.

⇒ **Algorithme** : procédure finie, non ambigüe, reproductible.

⇒ **Programme** : implémentation concrète (peu importe le langage).

⇒ **IO** (Input/Output) : fichiers, clavier, réseau, écran, figures...

Idée clé : on passe d'un monde flou à des **étapes vérifiables**, avec **contraintes** (temps, mémoire, précision).

Penser comme une machine : 4 piliers

Penser comme une machine : 4 piliers

1. **Décomposition** : couper un gros problème en sous-problèmes.

Penser comme une machine : 4 piliers

1. **Décomposition** : couper un gros problème en sous-problèmes.
2. **Reconnaissance de motifs** : repérer des schémas qui se répètent.

Penser comme une machine : 4 piliers

1. **Décomposition** : couper un gros problème en sous-problèmes.
2. **Reconnaissance de motifs** : repérer des schémas qui se répètent.
3. **Abstraction** : ne garder que l'essentiel (entrées/sorties, contraintes).

Penser comme une machine : 4 piliers

1. **Décomposition** : couper un gros problème en sous-problèmes.
2. **Reconnaissance de motifs** : repérer des schémas qui se répètent.
3. **Abstraction** : ne garder que l'essentiel (entrées/sorties, contraintes).
4. **Conception d'algorithmes** : écrire des étapes non ambiguës.

Penser comme une machine : 4 piliers

1. **Décomposition** : couper un gros problème en sous-problèmes.
2. **Reconnaissance de motifs** : repérer des schémas qui se répètent.
3. **Abstraction** : ne garder que l'essentiel (entrées/sorties, contraintes).
4. **Conception d'algorithmes** : écrire des étapes non ambiguës.

MAN machine, semi human being

Man machine, super human being

Pseudocode : conventions simples

```
ENTRÉES : données d'entrée (ex. N, prix, liste)
SORTIES : résultat attendu (ex. total, tri, décision)

ALGORITHME Exemple
    initialiser total <- 0
    POUR i DE 1 À N FAIRE
        total <- total + valeur(i)
    FIN POUR
    RETOURNER total
```

⇒ Mots-clés : `SI/ALORS/SINON`, `POUR`, `TANT QUE`, `RETOURNER`.

⇒ Éviter l'ambiguïté : **une** action par ligne, noms clairs.

Pseudocode — « Choisir un stand »

“Choisir un stand à la braderie” =

1. filtrer par prix,
2. trier par file d’attente,
3. prendre le premier.

Pseudocode — « Choisir un stand »

“Choisir un stand à la braderie” =

1. filtrer par prix,
2. trier par file d’attente,
3. prendre le premier.

```
ALGO CHOISIR_STAND(STANDS, BUDGET):
  CANDIDATS <- { s dans STANDS | s.prix <= BUDGET }
  SI vide(CANDIDATS) ALORS RETOURNER RIEN
  TRIER CANDIDATS PAR (file croissant, prix croissant, nom
    croissant)
  RETOURNER CANDIDATS[0].nom
FIN ALGO
```

Pseudocode — « Choisir un stand » (python)

```
stands = [
    {"nom":"Moules", "prix":12.0, "file":6},
    {"nom":"Frites", "prix":4.0,  "file":12},
    {"nom":"Gaufres","prix":3.5,  "file":3},
]
budget = 6

# 1) Filtrer par prix (<= budget)
candidats = [s for s in stands if s["prix"] <= budget]

# 2) Choisir la file la plus courte (puis prix, puis nom en cas d'
    égalité)
choix = min(candidats, key=lambda s: (s["file"], s["prix"], s["nom
    "])) if candidats else None

print("Aucun stand dans le budget" if not choix else f"Je prends {
    choix['nom']} ({choix['prix']})")
```

Comment modéliser ?

SCIENCES • CORONAVIRUS ET PANDÉMIE DE COVID-19

Comment l'épidémiologie tente de cerner l'épidémie due au nouveau coronavirus

Propagation, mortalité, impact du confinement… Si « tous les modèles sont faux, certains sont utiles », disent les statisticiens, qui disposent d'outils pour anticiper la suite de la pandémie.

Par David Larousserie

Publié le 30 mars 2020 à 18h30, modifié le 31 mars 2020 à 18h02 • Lecture 11 min.

Comment modéliser ?

SCIENCES • CORONAVIRUS ET PANDÉMIE DE COVID-19

Comment l'épidémiologie tente de cerner l'épidémie due au nouveau coronavirus

Propagation, mortalité, impact du confinement… Si « tous les modèles sont faux, certains sont utiles », disent les statisticiens, qui disposent d'outils pour anticiper la suite de la pandémie.

Par David Larousserie

Publié le 30 mars 2020 à 18h30, modifié le 31 mars 2020 à 18h02 • Lecture 11 min.

1. **État** : photographie du système à un instant (*stock*, *budget*, *position*).
2. **Variable** : quantité qui peut changer (entier, réel, texte, booléen).
3. **Invariants** : conditions **toujours vraies** (ex. stock ≥ 0).
4. **Transition** : passage d'un état à un autre après une action.

Comment modéliser ?

SCIENCES • CORONAVIRUS ET PANDÉMIE DE COVID-19

Comment l'épidémiologie tente de cerner l'épidémie due au nouveau coronavirus

Propagation, mortalité, impact du confinement… Si « tous les modèles sont faux, certains sont utiles », disent les statisticiens, qui disposent d'outils pour anticiper la suite de la pandémie.

Par David Larousserie

Publié le 30 mars 2020 à 18h30, modifié le 31 mars 2020 à 18h02 • Lecture 11 min.

1. **État** : photographie du système à un instant (*stock*, *budget*, *position*).
2. **Variable** : quantité qui peut changer (entier, réel, texte, booléen).
3. **Invariants** : conditions **toujours vraies** (ex. stock ≥ 0).
4. **Transition** : passage d'un état à un autre après une action.

Exo (3 min) : Pour une file d'attente, *i)* indiquez l'état minimal à suivre (nombre de clients, statut du serveur, instants de la prochaine arrivée et du prochain départ) ; *ii)* proposez un invariant que cet état doit toujours respecter (par ex. : si le serveur est libre,

File d'attente

État minimal à suivre

- ⇒ **W** : nombre de clients en file.
- ⇒ **serveur_occupé** : `vrai`/`faux`.
- ⇒ **t** : temps courant.
- ⇒ **prochaine_arrivée**, **prochain_départ** (ou ∞ si aucun service en cours).

File d'attente

État minimal à suivre

- ⇒ **W** : nombre de clients en file.
- ⇒ **serveur_occupé** : `vrai`/`faux`.
- ⇒ **t** : temps courant.
- ⇒ **prochaine_arrivée**, **prochain_départ** (ou ∞ si aucun service en cours).

Invariant (toujours vrai)

- ⇒ $W \geq 0$.
- ⇒ Si `serveur_occupé = faux`, alors `prochain_départ =` ∞.
- ⇒ Le temps **t** ne recule jamais.

File d'attente

Règle d'évolution (3 réflexes)

1. **Comparer** : si `prochaine_arrivée` $\leq$ `prochain_départ` $\Rightarrow$ *Arrivée*, sinon *Départ*.
2. **Arrivée** : avancer $t \leftarrow$ `prochaine_arrivée`.
   Si le serveur est libre $\Rightarrow$ le rendre `occupé` et fixer `prochain_départ = t + durée_service`.
   Sinon $\Rightarrow$ $W \leftarrow W + 1$. Puis programmer `prochaine_arrivée = t + inter_arrivée`.
3. **Départ** : avancer $t \leftarrow$ `prochain_départ`.
   Si $W > 0 \Rightarrow W \leftarrow W - 1$ et `prochain_départ = t + durée_service`.
   Sinon $\Rightarrow$ `serveur_occupé = faux`, `prochain_départ` $= \infty$.

Idée clé : on n'avance pas « seconde par seconde » ; on **saute d'un événement au suivant**. L'invariant garantit la cohérence de l'état.

Tracer un algorithme : à quoi ça sert ?

⇒ **Suivre** le programme **ligne par ligne**.

⇒ **Observer** après chaque ligne : état des variables et sortie.

⇒ **Utiliser** la trace pour : **comprendre**, **déboguer**, **expliquer**.

3 étapes clés :

1. Choisir des *entrées*.
2. Écrire le *pseudocode*.
3. Remplir le *tableau de trace*.

Exemple rapide :

`x ← 2`	$x = 2$
`x ← x+1`	$x = 3$

Tracer un algorithme : gabarit de trace

Ligne	**État des variables**	**Sortie / Commentaire**
1	(initialisation)	
2	(après affectation)	
3	(test Vrai/Faux ?)	
...	...	...

File d'attente — état minimal & invariant

```
init t=0, W=0, serveur_occupé=faux
prochaine_arrivée = t + inter_arrivée()
prochain_départ = INF

while t < Tfin:
  if prochaine_arrivée <= prochain_départ:
    t = prochaine_arrivée
    if serveur_occupé == faux:
      serveur_occupé = vrai
      prochain_départ = t + durée_service()
    else:
      W = W + 1
    prochaine_arrivée = t + inter_arrivée()
  else:
    t = prochain_départ
    if W > 0:
      W = W - 1
      prochain_départ = t + durée_service()
    else:
      serveur_occupé = faux
      prochain_départ = INF
```

Avant Python : penser comme un ordinateur

- ⇒ Prog : donner des **instructions** à l'ordinateur ;
- ⇒ **Séquenciel** : exécution *ligne par ligne*.
- ⇒ **Déterministe** : il fait exactement ce qu'on lui dit.
- ⇒ **Pseudocode** : décrire l'algorithme en français structuré.

Avant Python : penser comme un ordinateur

- ⇒ Prog : donner des **instructions** à l'ordinateur ;
- ⇒ **Séquenciel** : exécution *ligne par ligne*.
- ⇒ **Déterministe** : il fait exactement ce qu'on lui dit.
- ⇒ **Pseudocode** : décrire l'algorithme en français structuré.

Exo (5 min) : écrire en 6 étapes pour servir des moules-frites à la braderie. Puis indiquer 1 condition (*si pas de pain, alors...*).

Correction #1 — Braderie : Servir des moules-frites

Entrées : nb_personnes, prix_moules, prix_frites, budget
Sortie : commande prête (ou message d'échec)

Correction #1 — Braderie : Servir des moules-frites

Entrées : nb_personnes, prix_moules, prix_frites, budget
Sortie : commande prête (ou message d'échec)

Préconditions : nb_personnes ≥ 1, prix_moules > 0, prix_frites > 0, liste de stands ouverte.
Postcondition : une portion / *personne* servie ou l'échec annoncé.

Correction #1 — Braderie : Servir des moules-frites

Entrées : nb_personnes, prix_moules, prix_frites, budget
Sortie : commande prête (ou message d'échec)

> **Préconditions** : nb_personnes ≥ 1, prix_moules > 0, prix_frites > 0, liste de stands ouverte.
> **Postcondition** : une portion / *personne* servie ou l'échec annoncé.

1. **Choisir un stand** : file d'attente la plus courte **et** stock.
2. **Calculer** le coût total : `total = nb_personnes * (prix_moules + prix_frites)`.
3. Si `budget < total` $\Rightarrow$ **STOP** : afficher "budget insuffisant".
4. **Payer** et **vérifier** le rendu (monnaie exacte).
5. **Récupérer** les plateaux ; **vérifier** qu'il y a nb_personnes portions.
6. **Distribuer** 1 portion par personne.

Variante — Servir N personnes (boucle)

Variante — Servir N personnes (boucle)

Entrées : nb_personnes, stock_moules, stock_frites, budget, prix_moules, prix_frites

Variante — Servir N personnes (boucle)

Entrées : nb_personnes, stock_moules, stock_frites, budget, prix_moules, prix_frites

1. Si `stock_moules < nb_personnes` **ou** `stock_frites < nb_personnes` ⇒ **STOP** : « stock insuffisant ».
2. **Pour** `i` allant de 1 à `nb_personnes` :
   - 2.1 **Si** `budget < prix_moules + prix_frites` ⇒ **STOP** : « plus assez de budget ».
   - 2.2 **Débiter** le budget du prix d'une portion ; **décrémenter** les stocks.
   - 2.3 **Servir** 1 portion à la personne `i`.
3. **Fin** : tout le monde est servi ; **afficher** budget restant.

Idée clé : on *répète* une même procédure ⇒ notion d'itération + garde-fous (budget/stock).

Exemple #2 — Jeu « Deviner un nombre » (pseudocode)

Exemple #2 — Jeu « Deviner un nombre » (pseudocode)

```
# Entrées : nombre_secret (1..100)
# Sorties : nombre d'essais, message de succès
essais = 0
proposition = None

while proposition != nombre_secret:
    afficher("Proposez un nombre entre 1 et 100 :")
    proposition = lire_entier()      # input utilisateur
    essais = essais + 1

    if proposition < nombre_secret:
        afficher("Trop petit.")
    elif proposition > nombre_secret:
        afficher("Trop grand.")
    else:
        afficher("Bravo ! Trouvé en", essais, "essais.")
# Postcondition : proposition == nombre_secret
```

Patrons universels

⇒ **Compter/Accumuler** : somme, moyenne, min/max.

⇒ **Filtrer** → **Transformer** → **Agréger** : nettoyer puis résumer.

⇒ **Chercher** : linéaire ($\mathcal{O}(n)$) vs binaire ($\mathcal{O}(\log n)$) si trié.

Exo (5 min) : Liste de prix → retirer les valeurs NA, garder `prix<15`, calculer la moyenne.

Corrigé — Logique (pseudocode)

Objectif : calculer la moyenne des prix valides inférieurs à 15.

Pseudocode :

1. Initialiser `somme = 0`, `compte = 0`.
2. Pour chaque élément `p` de la liste :
   - Si `p` n'est pas manquant (NA) et `p < 15` :
     - `somme` ⇐ `somme + p`
     - `compte` ⇐ `compte + 1`
3. Si `compte > 0` : `moyenne = somme / compte`. Sinon : `moyenne = NA`.

Patron universel mobilisé : **Filtrer** ⇒ **Accumuler** ⇒ **Agréger**.

Corrigé — Implémentation (Python)

```
propres = [p for p in prix if isinstance(p,(int,float)) and
isfinite(p) and p < 15]
moyenne = sum(propres)/len(propres) if propres else NaN
```

Idée :

⇒ **Filtrer** : exclure NA et valeurs ≥ 15.

⇒ **Accumuler** : somme et effectif.

⇒ **Agréger** : calculer la moyenne.

Patron 1 — Accumuler / Compter / Résumer

Entrée : liste de stands avec `prix_menu`, `ventes`.
Sortie : `recette_totale`

```
recette_totale <- 0
POUR chaque stand s DANS stands FAIRE
    SI s.prix_menu <= 15 ALORS
        recette_totale <- recette_totale + s.ventes * s.prix_menu
    FIN SI
FIN POUR
RETOURNER recette_totale
```

Variantes : moyenne (= somme / compteur), min/max (= garder le meilleur vu).

Patron 2 — Filtrer ⇒ Transformer ⇒ Agréger

```
liste_filtrée <- []
POUR s DANS stands
    SI s.prix_ht <= 15 ALORS
        s.prix_ttc <- s.prix_ht * 1.20      # transformer
        AJOUTER s DANS liste_filtrée
    FIN SI
FIN POUR

somme <- 0 ; n <- 0
POUR s DANS liste_filtrée
    somme <- somme + s.prix_ttc
    n <- n + 1
FIN POUR
moyenne <- somme / n
```

Idée clé : séparer les étapes clarifie et évite les bugs.

Patron 3 — Tri + Parcours (groupements)

```
TRIER stands PAR quartier
courant_quartier <- None
somme <- 0
POUR s DANS stands
    SI s.quartier != courant_quartier ALORS
        SI courant_quartier != None ALORS
            AFFICHER courant_quartier, somme
        FIN SI
        courant_quartier <- s.quartier
        somme <- 0
    FIN SI
    somme <- somme + s.ventes
FIN POUR
AFFICHER courant_quartier, somme
```

Variables & types (modèle «boîte»)

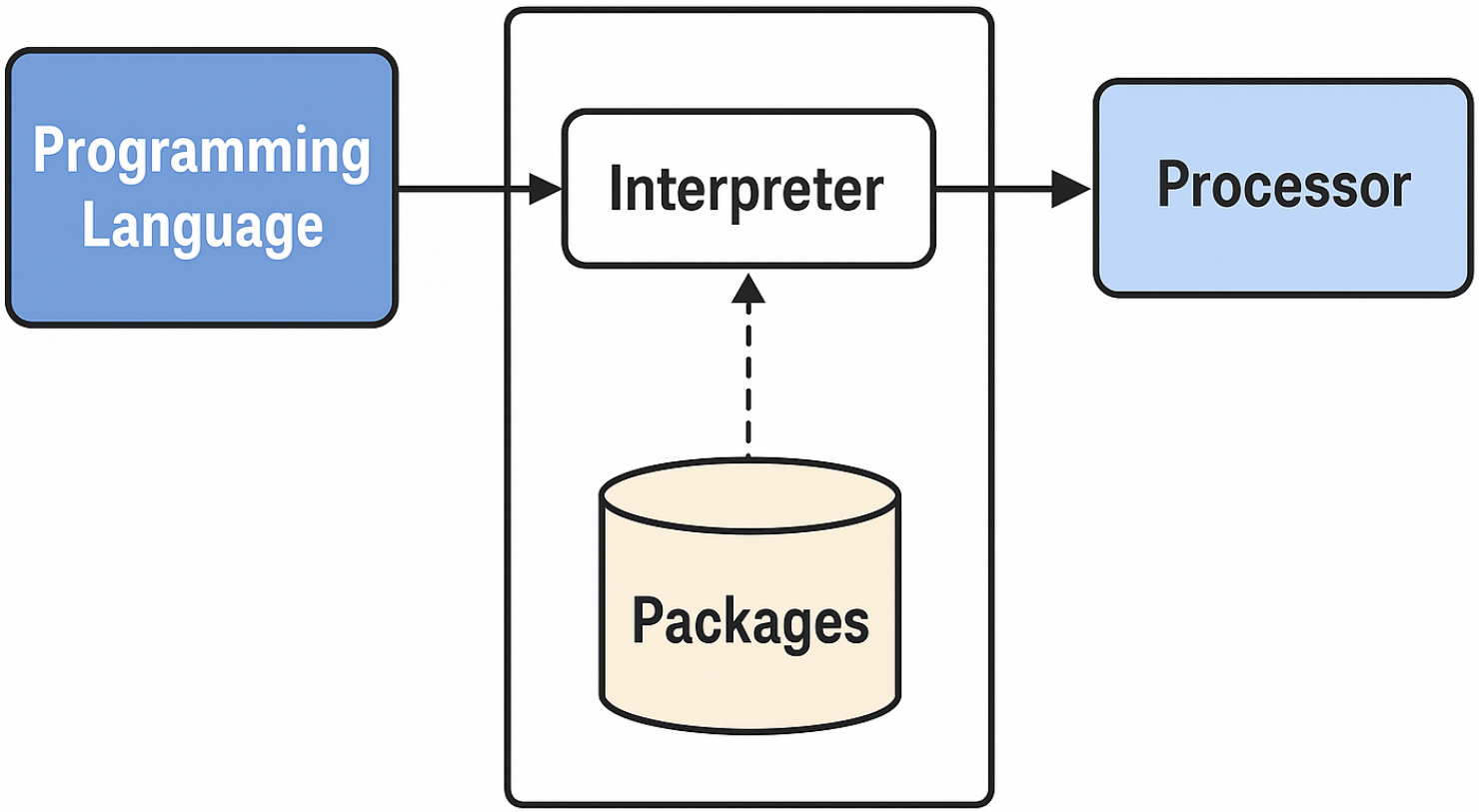
⇒ **Variable** = étiquette collée sur une *valeur*.

⇒ Types de base : entier (3), réel (3.14), texte ("Lille"), booléen (Vrai/Faux).

⇒ Opérateurs : `+ - * /` ; comparaisons : `< <= > >= == !=` ; logiques : `ET, OU, NON`.

Idée : “remplacer et recalculer” — à chaque ligne, la valeur peut changer.

Qu'est-ce qu'un langage de programmation ?



Un langage, c'est quoi ?

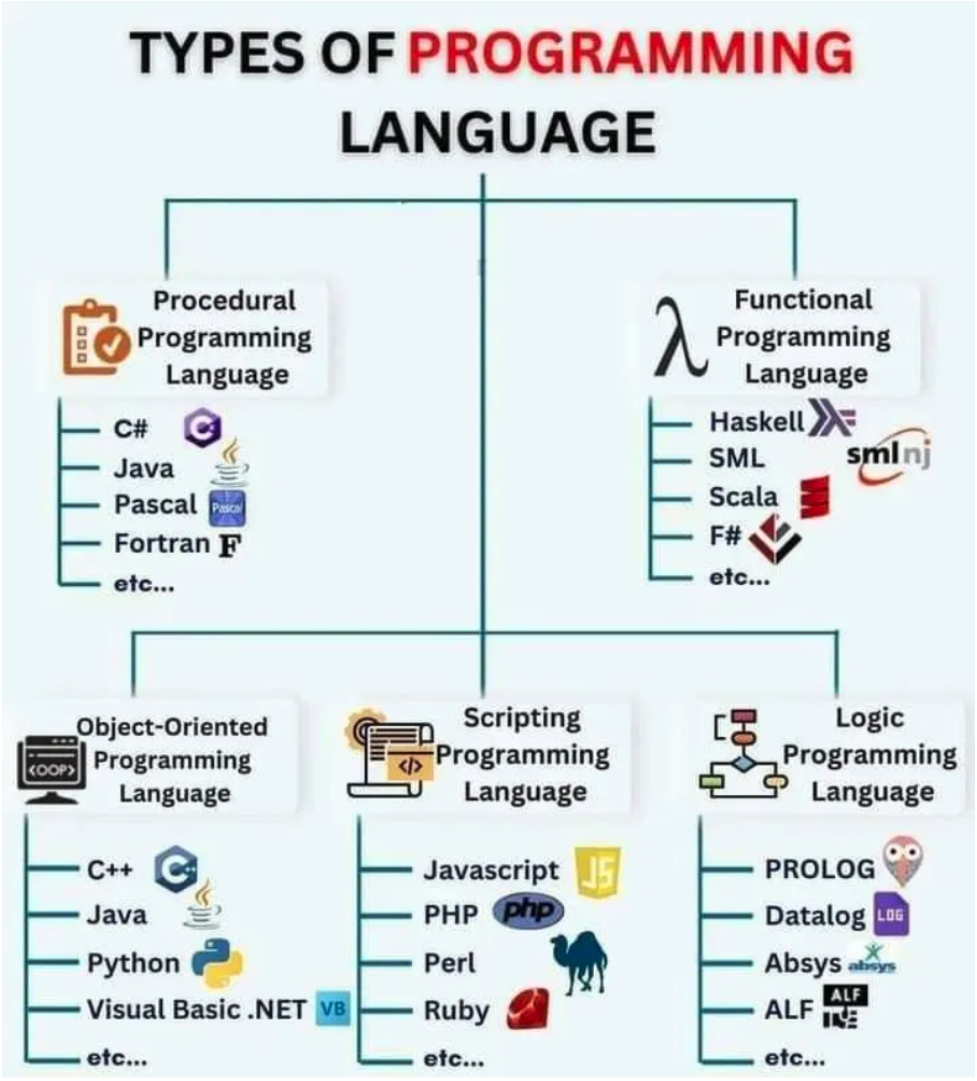
⇒ Un **langage de programmation** décrit *exactement* les actions à effectuer.

⇒ Deux grandes familles :

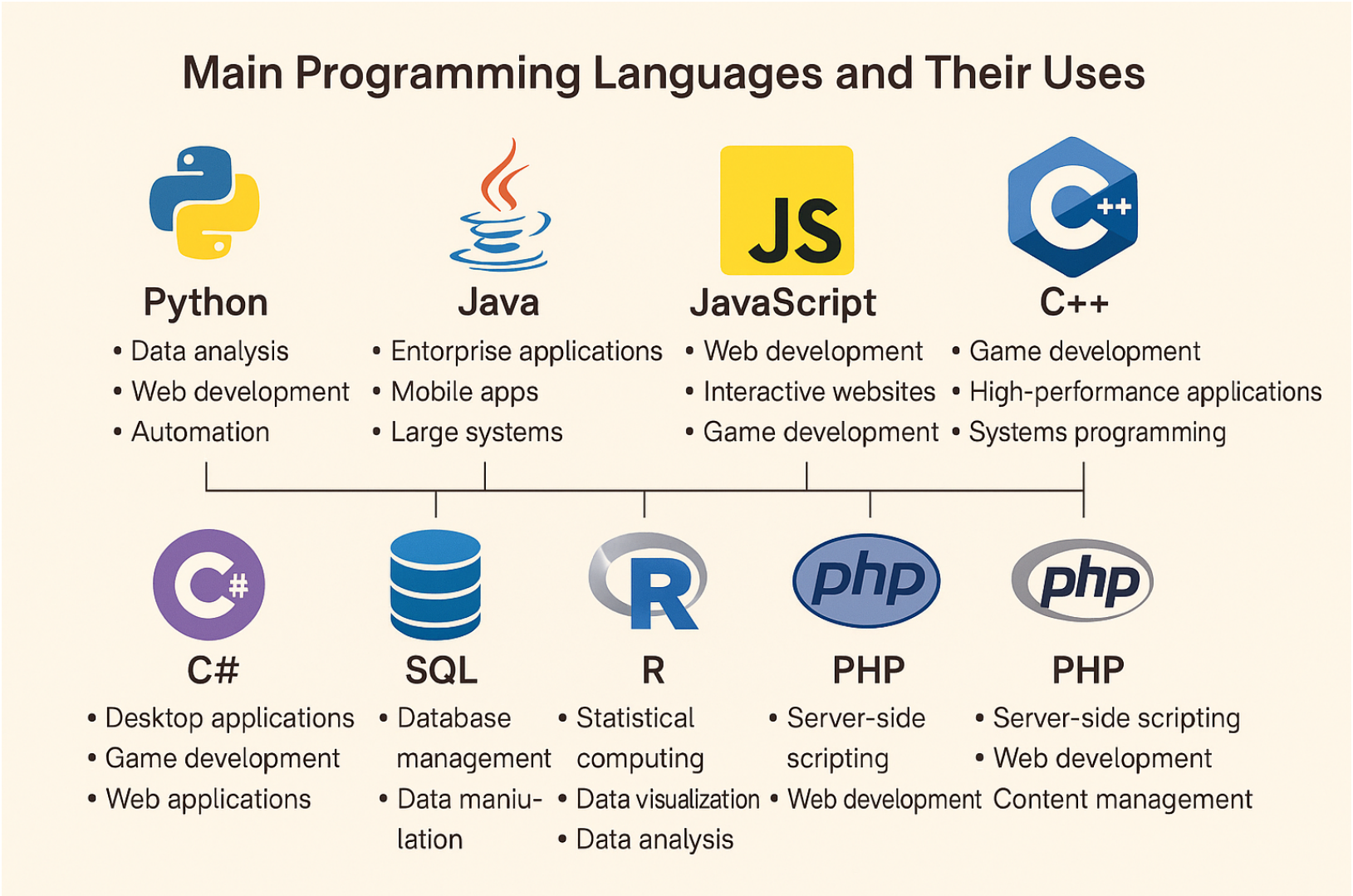
- **Compilés** (C/C++) : traduits en code machine avant l'exécution.
- **Interprétés** (Python) : exécutés *ligne par ligne* par un **interpréteur**.

⇒ **Python** : simple à lire, énorme communauté, idéal pour la data.

Une multitude de langages



Les langages et leurs applications



Un langage… et son écosystème

⇒ Un **langage** (Python, R, JavaScript, Julia…) = règles pour écrire des programmes.

⇒ Autour du langage : un **écosystème** qui fournit des briques prêtes à l'emploi.

⇒ Trois couches (idée générale, valable dans d'autres langages) :

1. **Noyau** du langage + **librairie standard** (ex. lire un fichier, faire des maths).
2. **Packages externes** maintenus par la communauté (*pandas*, *matplotlib*…).
3. **Votre code** (fonctions, modules) qui assemble tout ça pour résoudre votre problème.

Image mentale : **langage = grammaire**, **librairie standard = vocabulaire de base**, **packages = jargon spécialisé**, **votre code = votre discours**.

Module, package, bibliothèque : qui est qui ?

⇒ **Fonction** : action élémentaire (`moyenne()`, `tracer()`).

⇒ **Module** : fichier qui regroupe des fonctions & variables (ex. `statistiques.py`).

⇒ **Package** : **collection** de modules (ex. `pandas`, `matplotlib`).

⇒ **Bibliothèque** (ou *library*) : terme générique pour un ensemble de modules/packages.

Analogie : **fonction** = **outil**, **module** = **tiroir**, **package** = **boîte à outils**.

Importer : utiliser des outils sans les réécrire

```
# importer un module de la librairie standard
import math
rayon = 3
aire = math.pi * (rayon**2)

# importer une fonction précise d'un module
from statistics import mean
m = mean([10, 12, 15])

# donner un alias pour éviter les noms trop longs
import matplotlib.pyplot as plt
```

⇒ **Espace de noms** : on évite les collisions (deux fonctions nommées `mean`).

⇒ **Lisibilité** : alias courts mais **standards** (`plt`, `pd`, `np`...).

Standard vs. externe : où sont les frontières ?

Librairie standard (incluse)

- ⇒ Fichiers texte/CSV simples, dates, maths de base, statistiques basiques.
- ⇒ Toujours disponible avec le langage.
- ⇒ Ex. Python : `csv`, `json`, `datetime`, `statistics`.

Packages externes (à installer)

- ⇒ Dataframes, graphes avancés, machine learning, web, cartes...
- ⇒ Installés via un **gestionnaire de paquets**.
- ⇒ Ex. Python : `pandas`, `matplotlib`, `scikit-learn`.

Règle d'or : **stdlib pour le simple**, **packages** dès que ça devient **répétitif ou avancé**.

Gestionnaire de paquets : l'App Store du code

Idée générale (tous langages) : un outil **centralise** l'installation, les mises à jour et les **dépendances**.

Python	`pip`
R	`install.packages()` / CRAN
JavaScript	`npm` / `yarn`
Julia	`Pkg.add()`

Ex. (Python) installer `pandas` et `matplotlib` :

```
pip install pandas matplotlib
```

(Nous verrons l'isolation des installations — "environnements" — au cours suivant.)

Pourquoi des packages ? Exemple “Braderie”

Même tâche : moyenne du prix des moules-frites par quartier.
Sans package (stdlib) :

```
import csv, statistics
prix_par_q = {}
with open("data/stands_braderie.csv", encoding="utf-8") as f:
    for row in csv.DictReader(f):
        if row["menu"]=="moules-frites":
            q = row["quartier"]
            prix_par_q.setdefault(q, []).append(float(row["prix"])
    )
moyennes = {q: statistics.mean(v) for q, v in prix_par_q.items()}
```

Avec un package (pandas) :

```
import pandas as pd
df = pd.read_csv("data/stands_braderie.csv")
moyennes = (df.query("menu=='moules-frites'")
              .groupby("quartier")["prix"].mean())
```

Comment choisir un “bon” package ? (checklist)

⇒ **Usage courant** dans votre domaine (ex. `pandas` pour les données tabulaires).

⇒ **Documentation** claire + exemples (site, docstrings).

⇒ **Communauté** active (mises à jour récentes).

⇒ **API stable** (changements annoncés, sémantique de version).

⇒ **Licence** compatible (open source, réutilisable).

Principe : **peu de packages, bien choisis**. Éviter “tout installer” ⇒ dette technique.

Mini-reproductibilité (premier pas)

```
# lister ce que vous avez utilisé aujourd'hui
pip freeze > requirements.txt
# rejouer ailleurs (même versions)
pip install -r requirements.txt
```

Idée : noter les versions fige les résultats. (*On isolera proprement via environnements au cours 2.*)

Comment l'ordinateur voit vos fichiers

⇒ **Tout est fichier** : documents, images, code (`.py`) sont des *fichiers texte* ou binaires.

⇒ **Arborescence** : des *dossiers* (répertoires) contiennent des fichiers et d'autres dossiers.

⇒ **Chemin** = adresse du fichier :

- **Absolu** : depuis la racine (`/Users/Alice/proj/data/iris.csv`).
- **Relatif** : depuis le dossier courant (`data/iris.csv`).

⇒ **Raccourcis** : `.` (ici), `..` (dossier parent), `~` (dossier utilisateur).

Astuce : Évitez les espaces et accents dans les noms de fichiers (`mes-donnees.csv` plutôt que `Mes données.csv`).

Pourquoi rendre son code réutilisable ?

Estimating the reproducibility of psychological science

OPEN SCIENCE COLLABORATION Authors Info & Affiliations

SCIENCE · 28 Aug 2015 · Vol 349, Issue 6251 · DOI: 10.1126/science.aac4716

Empirically analyzing empirical evidence

One of the central goals in any scientific endeavor is to understand causality. Experiments that seek to demonstrate a cause/effect relation most often manipulate the postulated causal factor. Aarts *et al.* describe the replication of 100 experiments reported in papers published in 2008 in three high-ranking psychology journals. Assessing whether the replication and the original experiment yielded the same result according to several criteria, they find that about one-third to one-half of the original findings were also observed in the replication study.

Science, this issue 10.1126/science.aac4716

Comment rendre son code réutilisable ?

⇒ Partage du code et des données des articles scientifique ;

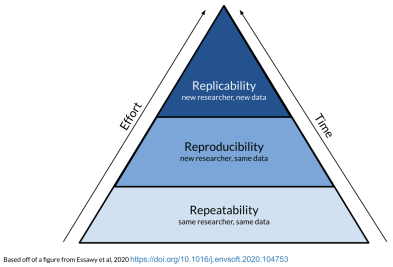
⇒ **Exemples** : Harvard Data Verse & *Open Science Foundation*

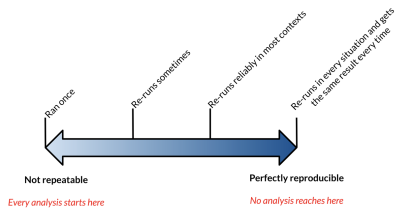
⇒ **Exercices** : prenez quelques exemples pour l'organisation du *workflow* ;

1. Commment les fichiers sont-ils organisés ?
2. Qu'est-ce qui fait que le code est reproductible ?

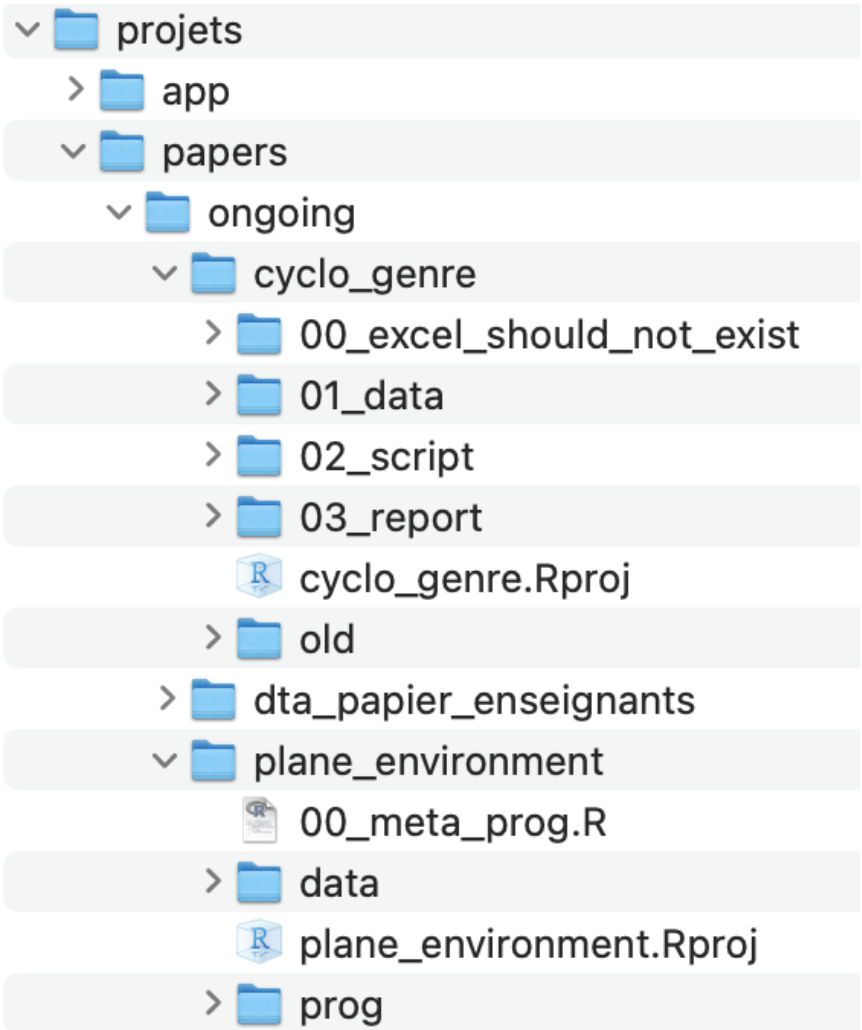
Comment rendre son code réutilisable ?

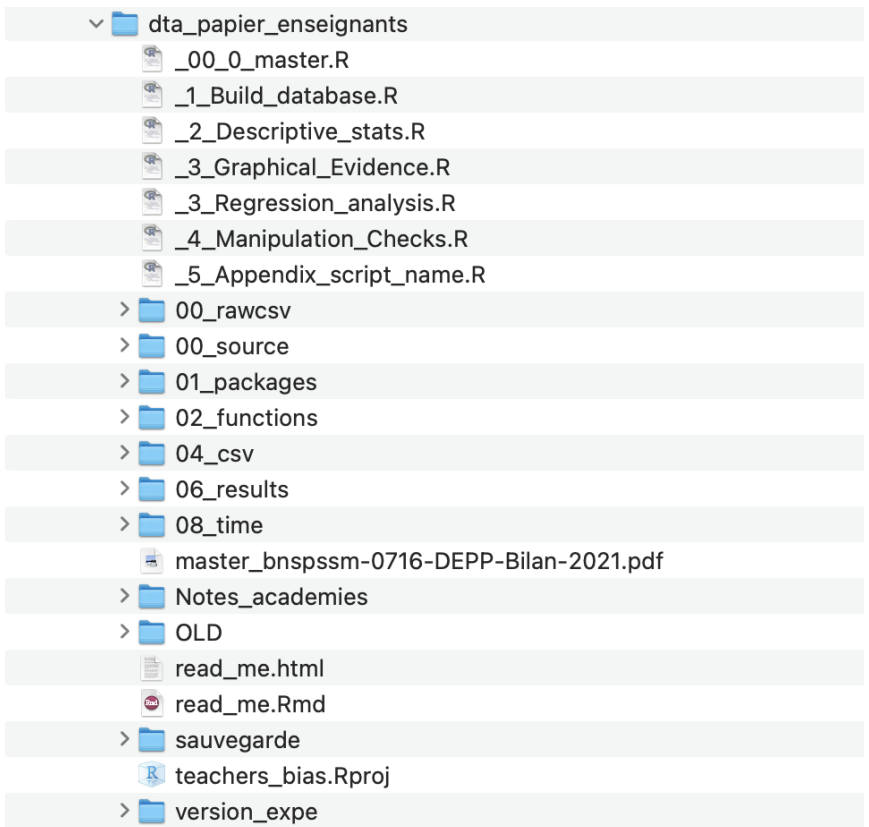
- ⇒ Partage du code et des données des articles scientifique ;
- ⇒ **Exemples** : Harvard Data Verse & *Open Science Foundation*
- ⇒ **Exercices** : prenez quelques exemples pour l'organisation du *workflow* ;
  1. Commment les fichiers sont-ils organisés ?
  2. Qu'est-ce qui fait que le code est reproductible ?





Arborescence- Exemples





Terminal : se déplacer et voir où on est

macOS / Linux (bash/zsh)

```
pwd              # où suis-je ?
ls               # lister
cd proj          # entrer dans proj/
cd ..            # remonter
mkdir data       # créer un dossier
```

Windows (PowerShell)

```
pwd              # où suis-je ?
ls               # lister
cd proj          # entrer dans proj\
cd ..            # remonter
mkdir data       # créer un dossier
```

Règle d'or : placez-vous (`cd`) dans le dossier du projet avant de lancer Python.

Coder efficacement : quelques principes clés

⇒ **Comprendre le fonctionnement d'un ordinateur :**

- L'ordinateur ne "*comprend*" pas : il suit des instructions exactes.
- Il exécute uniquement ce qu'on lui demande, même si cela n'a aucun sens logique.
- Un code ambigu ou mal structuré produira des erreurs... ou de mauvais résultats.

Coder efficacement : quelques principes clés

⇒ **Comprendre le fonctionnement d'un ordinateur :**

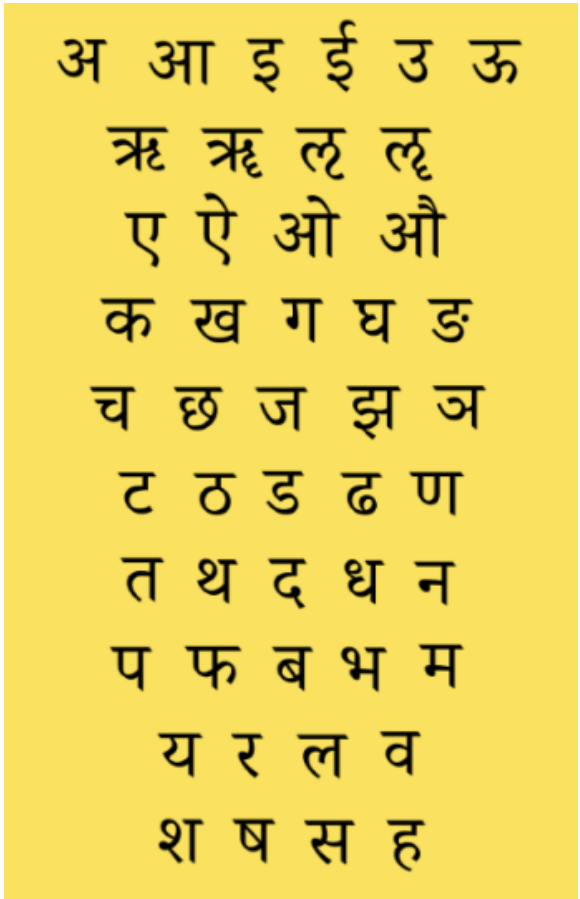
- L'ordinateur ne "*comprend*" pas : il suit des instructions exactes.
- Il exécute uniquement ce qu'on lui demande, même si cela n'a aucun sens logique.
- Un code ambigu ou mal structuré produira des erreurs... ou de mauvais résultats.

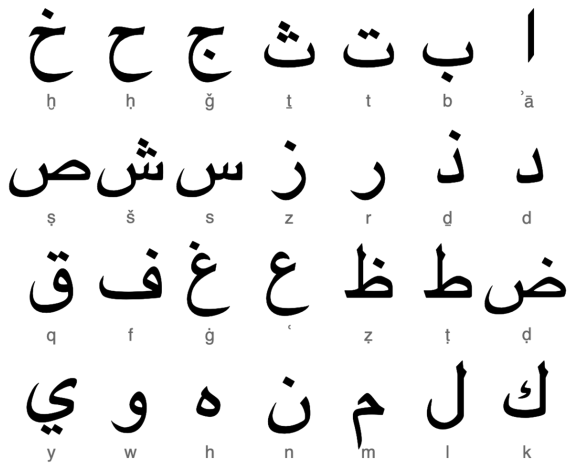
⇒ **Bonnes pratiques en programmation :**

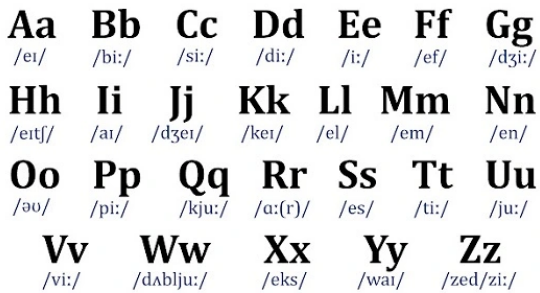
- Écrire un code clair, lisible et **réutilisable**.
- Ajouter des commentaires explicatifs **à chaque étape importante**.
- Limiter le nombre de packages : privilégier ceux bien documentés et maintenus.
- Toujours lire la documentation des packages utilisés.
- Automatiser les tâches répétitives dès que possible.

Règle d'or : Codez pour vous, mais surtout pour la personne qui relira votre code dans 6 mois (même si cette personne, c'est vous).

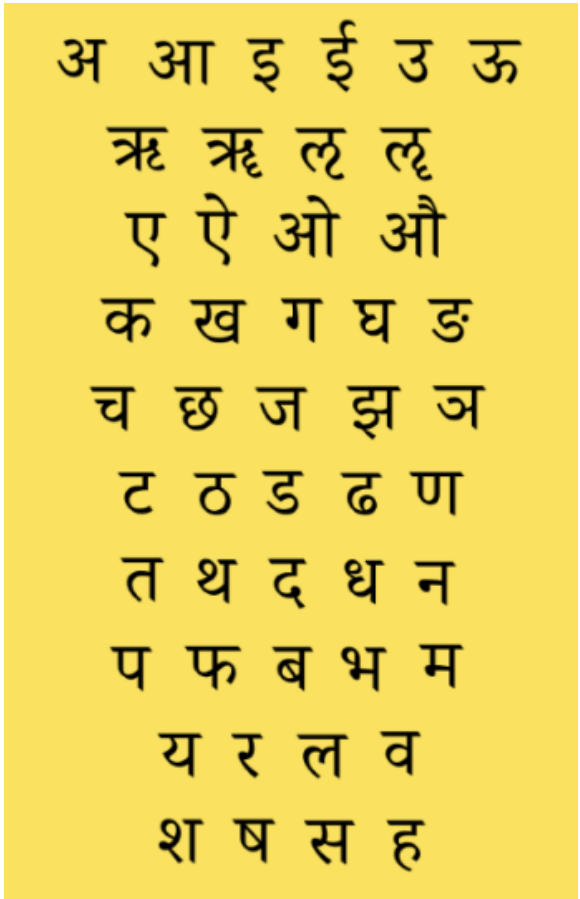
Les alphabets

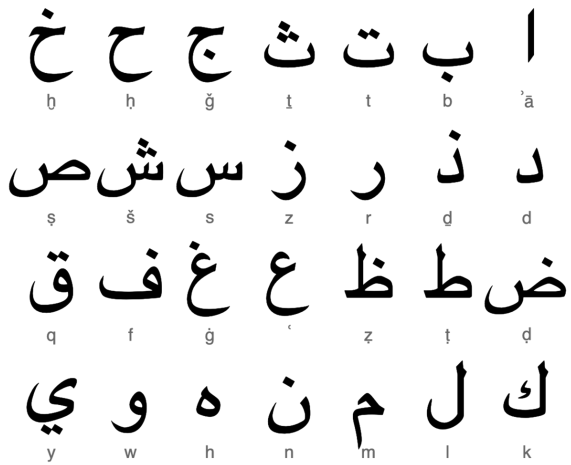


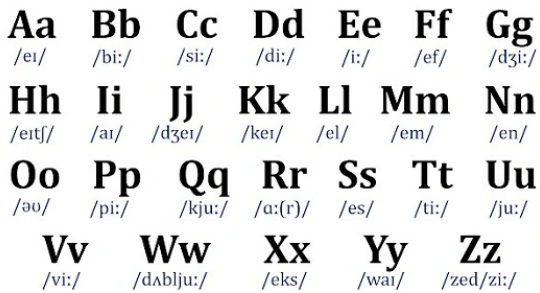




Les alphabets







Conventions : La programmation est occidentalo-centrée : on utilise les caractères anglo-saxons + l'ordinateur ne lit pas les espaces.

Pourquoi apprendre Python ?

- ⇒ Langage **open source**, gratuit et maintenu par une large communauté
- ⇒ Très utilisé dans le monde académique, la data science, l'économie, le journalisme, la finance, etc.
- ⇒ Syntaxe simple, proche du langage naturel
- ⇒ Outils puissants pour : traitement de données, visualisation, apprentissage automatique
- ⇒ S'intègre facilement avec d'autres langages ou logiciels (R, Stata, Excel)

Python est à la fois un langage d'apprentissage et un outil professionnel de premier plan.

Figure 1.1, 38.8% des répondants indiquent développer en Python, soit 6.8 points de pourcentage de plus qu'un an auparavant, ce qui fait de ce langage de programmation celui dont la croissance a été la plus importante entre 2017 et 2018.

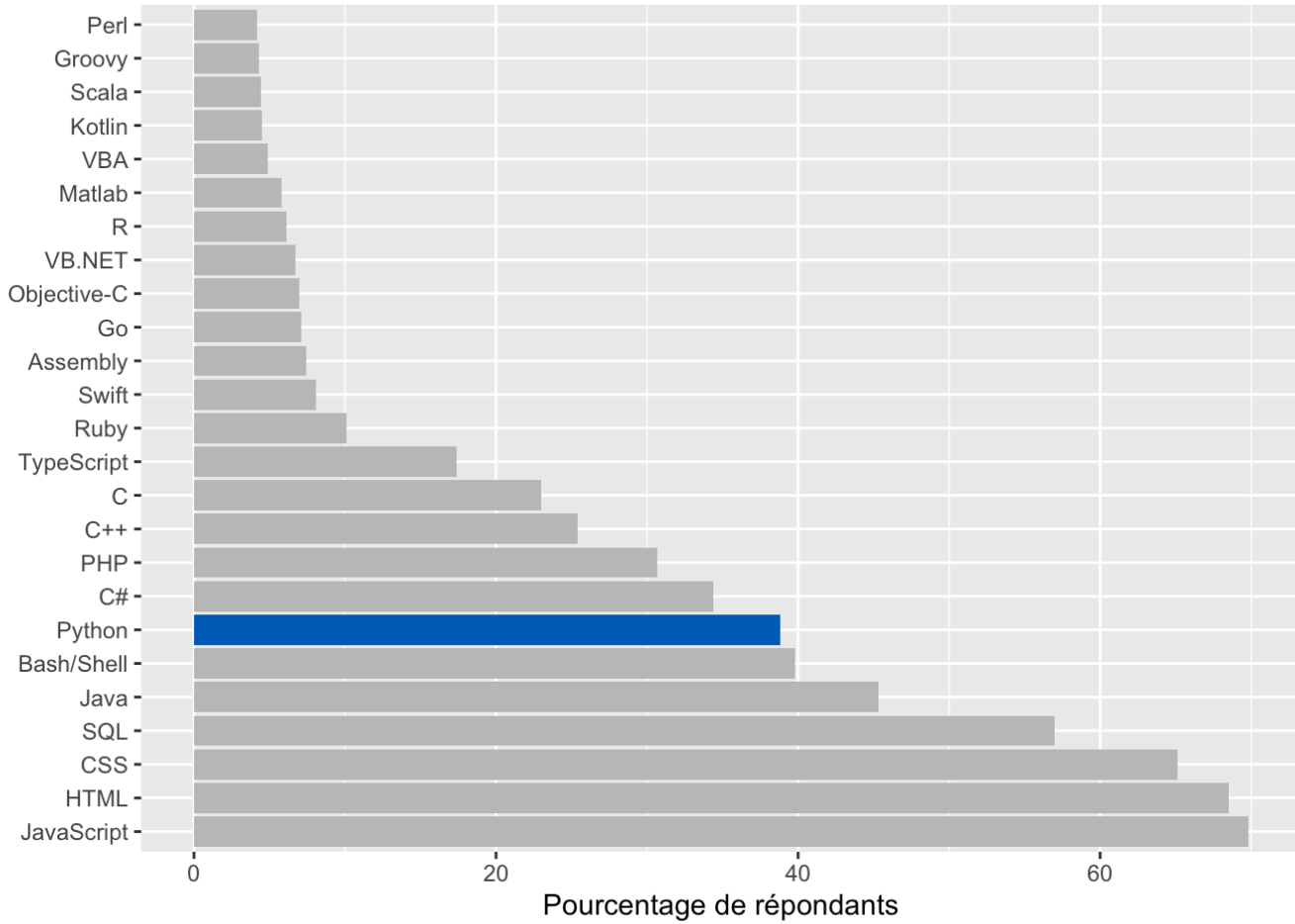


Figure 1.1: Langages de programmation, de scripting et de balisage.

: Source : Ewen Gallic - Python pour économistes

Un squelette de projet pour réussir

```
proj/
  data/            # CSV bruts
  notebooks/       # Jupyter (.ipynb)
  src/             # fonctions réutilisables (notre mini-module)
  scripts/         # scripts à exécuter
  outputs/         # figures, exports
  README.md        # comment lancer ?
```

⇒ **Séparer** données brutes / code / sorties.

⇒ **Documenter** dans `README.md` : 5 lignes suffisent.

Lire un message d'erreur (Traceback)

```
Traceback (most recent call last):
  File "scripts/analyse.py", line 4, in <module>
    import pandas as pd
ModuleNotFoundError: No module named 'pandas'
```

⇒ **Type d'erreur** : `ModuleNotFoundError` ⇒ package manquant.

⇒ **Ligne** : où ça casse (`analyse.py`, ligne 4).

⇒ **Solution** : activer l'environnement et `pip install pandas`.

Méthode : lire *du bas vers le haut*, identifier *type + ligne*, corriger, relancer.

Débogage sans paniquer (5 pas)

1. Lire le **type d'erreur** et la **ligne** indiquée
2. **Reproduire** l'erreur avec un *exemple minimal*
3. Inspecter les **valeurs** (afficher variables intermédiaires)
4. Vérifier **chemins/encodage/types** (les 3 suspects courants)
5. Modifier une chose à la fois & **relancer**

Mantra : "*Qu'attendais-je ? Qu'est-ce qui s'est réellement passé ? Où divergent-ils ?*"

Erreurs fréquentes (et comment les lire)

`NameError` : variable inconnue → faute de frappe, ordre d'exécution.
`IndentationError` : mauvais retrait → 4 espaces cohérents.
`TypeError` : types incompatibles → caster (`int/float/str`).
`ValueError` : valeur invalide → NA/texte dans une colonne num.
`FileNotFoundError` : mauvais chemin → `pwd` + chemin relatif.

```
# Exemples minimalistes à tester
prix = "12.5"
float(prix)        # OK
"moules" + 10      # TypeError
```

Écrire du code propre dès le début

⇒ **Noms clairs** : `prix_ttc`, `nb_eleves`, pas `x`, `a1`.

⇒ **Indentation** 4 espaces, pas de tabulations mixtes.

⇒ **Commentaires** : expliquer *pourquoi*, pas seulement *ce que* fait la ligne.

⇒ **Découper** en *fonctions* courtes (une idée = une fonction).

⇒ **Sauver/figer** l'environnement : `pip freeze > requirements.txt`.

Exercices pas à pas

1. Mise en place

1. Créer le dossier `proj/` avec la structure vue.
2. Créer et activer `.venv`, installer `jupyter pandas matplotlib`.
3. Lancer `jupyter notebook`.

2. Notebook

1. Lire `data/iris.csv`, afficher `head()`, `info()`, `describe()`.
2. Tracer un barplot de la moyenne de `petal_width` par `species`.
3. Sauver la figure dans `outputs/`.

3. Module & test

1. Créer `src/monpkg/utils.py` avec une fonction `taux_croissance(x_t, x_t1)`.
2. Importer la fonction dans le notebook et `assert` deux cas.

4. Script

1. Adapter `scripts/analyse.py` pour choisir la colonne à *médianer* au lieu de la moyenne.