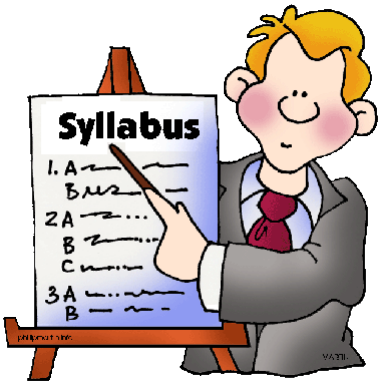
Python comme Ruby sont d'excellentes solutions pour la découverte de la programmation

Sommaire

Python et la saga des langages

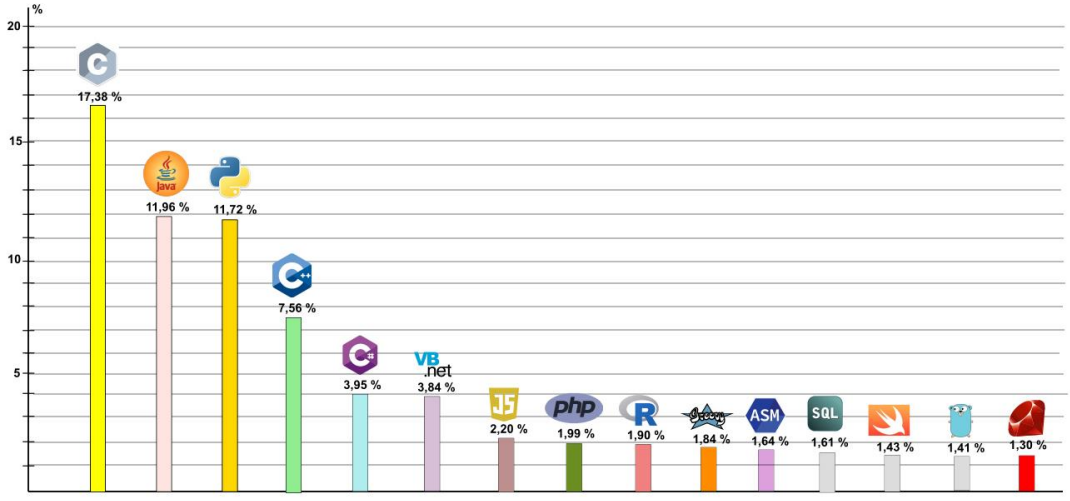
- *La liberté retrouvée*
- *Le palmarès des langages en 2021*
- *Comment expliquer l'engouement de Python*
- *La rupture entre les versions 2 et 3*
- *Esthétisme et lisibilité*
- *Une typologie moderne*
- *La nécessaire modularité*
- *Des structures de données avancées*
- *Les ingrédients de programmation fonctionnelle*
- *Les gestionnaires de paquets*
- *Multi-plateforme*
- *Tout n'est pas parfait, mais…*
- *L'écosystème Python*



La liberté retrouvée

- Début des années 90
- De nombreux développeurs s'affranchissent de la tutelle des éditeurs : Microsoft, IBM, Oracle… pour créer des langages nouveaux, libres et originaux
- Guido Van Rossum avec Python, Anders Ejlsberg avec Delphi et C#, Matsumoto avec Ruby et d'autres qui viendront après : Dahl (Node), Ashkenas (CoffeeScript), Eich (JavaScript)…
- Van Rossum est le symbole de cette liberté… passé désormais chez Microsoft (comme Ejlsberg)
- Auparavant c'était IBM et quelques autres qui décidaient de l'évolution des langages (40 ans avec Cobol chez IBM), désormais ce sont les individus, dans des projets de recherche universitaires qui ont les idées, les mettent en pratique, mais aussi Google et Facebook qui inventent pour leurs propres besoins
- Aujourd'hui : 100 000 langages, dont on ne connaît qu'une infime partie

Le palmarès des langages en 2021



Comment expliquer l'engouement pour Python

- Python est un langage orienté objet, interprété, doté d'un typage dynamique fort, d'une gestion automatique des objets par ramasse-miettes et d'un excellent système de gestion des exceptions.
- Langage de débutant, très utilisé en apprentissage (serait mieux que le low code…)
- multi-plates-formes
- Libre et Open Source
- Orienté objet
- Grande diversité des librairies (Scikit-Learn pour IA, Pandas pour l'analyse de données, NumPy pour le calcul vectoriel et matriciel, SciPy pour l'ingénierie…) et des applications
- Importance de la communauté : près de 30 ans d'existence, très gros apport de la communauté scientifique
- Adéquation avec les IoT
- Très bon pour le prototypage
- Extensibilité : on peut écrire un code en Python et l'augmenter avec des modules C, C++…
- Langage mature utilisé par des grands noms : Yahoo, YouTube, DropBox

La rupture entre les versions 2 et 3

- Guido Van Rossum, conscient de certaines incohérences de son langage, a pris la décision courageuse en décembre 2008, de provoquer une scission en annonçant la version 3 du langage, incompatible avec la précédente.
- Il en a profité pour nettoyer les bibliothèques de base de ses éléments obsolètes et donner une image nouvelle d'un outil, qui pouvait déjà compter sur une communauté acquise.
- Il existait cependant de nombreuses applications écrites en Python 2, qu'il était la plupart du temps, exclu de réécrire.
- La version 3 bénéficiait de nombreux avantages :
  - Le **print**, devenu une fonction
  - Certains itérateurs qui ont été revus et ne renvoient plus de listes
  - Refonte des opérateurs de comparaison
  - Un seul type d'entier
  - Renommage plus cohérent de certaines bibliothèques de base
- Les dernières versions de Python sont la 2.7.16, disponible depuis le 3 mars 2019 et qui sera la dernière de la filière Python 2, qui devait disparaître en 2020 et la 3.10 en novembre 2020.
- Des indiscrétions ont évoqué la sortie d'un Python 4…

La machine virtuelle de Python

- La PVM ("Python Virtual Machine") est un programme qui fournit un environnement paramétrable : une MV légère.
- Python est un interpréteur hybride : quand un programme s'exécute, il l'assemble dans un bytecode qui peut être exécuté par l'interpréteur.
- La machine virtuelle Python permet d'exécuter le code directement sans le compiler.


```
>>> import dis
>>> def example(x):
        for i in range (x):
            print(2*i)
>>> dis.dis(example)
  2           0 SETUP_LOOP              28 (to 30)
              2 LOAD_GLOBAL              0 (range)
              4 LOAD_FAST                0 (x)
              6 CALL_FUNCTION            1
              8 GET_ITER
        >>   10 FOR_ITER                16 (to 28)
             12 STORE_FAST               1 (i)

  3          14 LOAD_GLOBAL              1 (print)
             16 LOAD_CONST               1 (2)
             18 LOAD_FAST                1 (i)
             20 BINARY_MULTIPLY
             22 CALL_FUNCTION            1
             24 POP_TOP
             26 JUMP_ABSOLUTE           10
        >> 28 POP_BLOCK
        >> 30 LOAD_CONST                0 (None)
             32 RETURN_VALUE
>>>
```

- Python a la chance de pouvoir s'appuyer sur des plates-formes virtuelles, distinctes pour les versions 2 et 3.
- Ce sera le cas si l'on doit fonctionner avec des versions différentes des mêmes packages.
- Si cela se produit, on appréciera de compter sur Virtualenv, le run-time virtuel pour Python 2 et sur venv, son équivalent en Python 3.
- Chaque environnement virtuel comportera sa propre copie de l'interpréteur et consommera environ 25 MB d'espace sur disque (raisonnable).

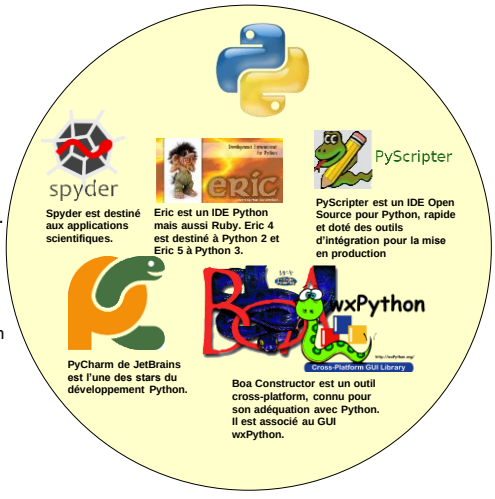
Python est un langage objet

- Les types de base, les fonctions, les instances de classes, tout est objet, au sens habituel de la POO.
- Une classe Python, définie par le mot clé **`class`**, supporte l'héritage multiple, sans les restrictions Java.
- Contrairement à d'autres langages où il faut passer par des mots clés dédiés (**extends** en Java…), rien de tel en Python. Pour exprimer l'héritage, il suffit d'écrire un code du type :

```
class MaClasse1 :
     #le code de MaClasse1
class MaClasse2(MaClasse1): #définition de l'héritage
     #le code de MaClasse2
```

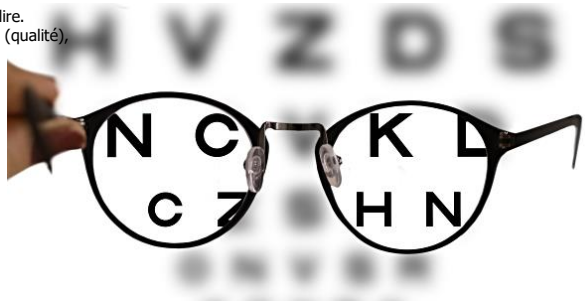
- La construction d'un objet se fait via la méthode **__init__()**.
- Encapsulation : il n'y a pas de variables privées ou publiques et on fait confiance aux utilisateurs.
- Si une variable n'a pas pour vocation d'être vue de l'extérieur, alors l'utilisateur n'y accède pas ! C'est confus : s'il n'y a pas d'encapsulation, on peut quand même s'arranger pour que l'on ne puisse accéder aux variables qu'à travers des accesseurs et des mutateurs.
- Une autre caractéristique énerve les développeurs : le **self,** un paramètre très important, qui précise l'instance de classe à laquelle il s'applique. L'instance en cours.
- En Python l'usage du self s'explique par le fait que l'on peut utiliser l'héritage multiple et que l'on ne pourrait pas distinguer une variable locale d'une variable de niveau supérieur.
- C'est l'équivalent du **`this`** de JavaScript et du **`$this`** :

```
class MaClasse :
     def __init__(self):
          self.attribut = "valeur"
          def ma_methode(self)
               print(self.attribut)
```



Esthétisme et lisibilité

- Python a une écriture simple, peu encombrée : "on parle" Python, plutôt qu'on le programme.
- Indentation forcée, ce qui donne un code mieux structuré et plus agréable à lire.
- Il n'y a pas d'accolades et c'est l'indentation qui va jouer leur rôle, esthétique (qualité), mais aussi de structuration.
- Par convention, un niveau d'indentation correspond à quatre blancs.



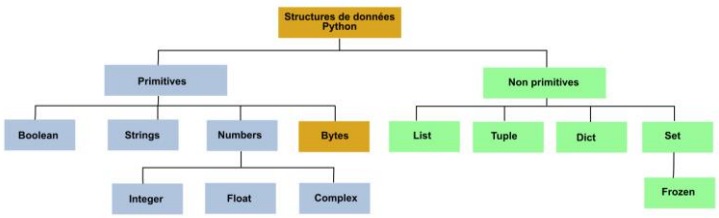
Exemple de "Bonjour le Monde", comparé à Java :

```
# En Python
print "Bonjour le Monde" #c'est tout

// En Java
class BonjourMonde {
      public static void main(String[] args)
      {
            System.out.println("Bonjour le Monde") ;
      }
}
//on est déjà fatigué…
```


Une typologie moderne

- Python s'appuie sur une typologie très complète, qui va de pair avec la diversité de ses structures de données, mais aussi le concept d'itération, très important chez lui…
- Il y a les types numériques, mais surtout les objets itérables, dont on peut parcourir les valeurs, à l'aide d'un for, par exemple.
- La liste de ces objets itérables est longue :
  - les « **tuple »**, des listes immuables d'objets hétérogènes
  - les « **list** », des tableaux dynamiques qui s'étendent automatiquement quand on en a besoin et sont constitués d'objets hétérogènes
  - les « **set** », des ensembles non ordonnés d'objets, dont les « frozenset » sont une variante immuable, donc fonctionnelle
  - les « **dict** » pour dictionnaires, des tableaux qui associent une clé à une valeur et dont on aura bien besoin pour manipuler des bases de données de type clé-valeur ou documents MongoDB, par exemple
  - les « **str** », des chaînes de caractères immuables (Unicode depuis Python 3.0)
  - les « **bytearray** », des chaînes d'octets modifiables
  - les « **file** », pour les fichiers avec la méthode **open()**, entre autres
  - voire encore les « **xrange** » (avec la méthode **range()**) et surtout les types associés aux méthodes de dictionnaires : **.keys()**, **.values()**, **.items()**, certains étant immuables et d'autres non.



Des structures de données avancées

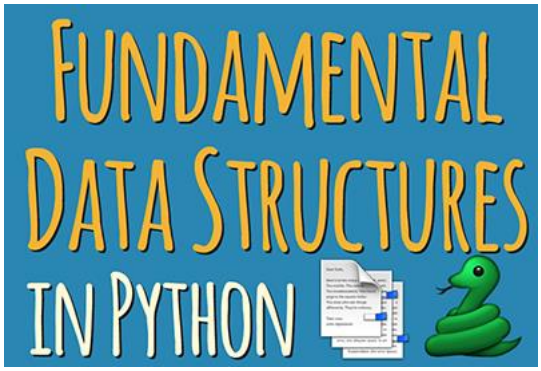
- Listes, dans la tradition de LISP (McCarthy) : ensembles de données hétérogènes, pouvant être manipulés globalement.
- Python est perçu comme un excellent outil pour traiter des séries temporelles, pour les IoT.

```
maListe=["Stratégie", "Technologie", "Datacenter"]
```
#Python depuis sa version 3 est unicode

- Il y a deux manières d'attaquer une liste, soit directement, soit par les méthodes associées à l'objet liste

```
maListe[2] = "Les réseaux"
```
l'index commence toujours à 0, ici pour ajouter un élément
```
del(maListe[2])
```
pour supprimer un élément

- Le recours aux méthodes est plus élégant, bien qu'un peu plus verbeux :
- `maListe.remove(2)` # pour supprimer l'élément en position 2, avec la méthode remove
- `maListe.insert(2, "Développement")` # pour insérer un élément en position 2 avec la méthode insert
- `maListe.append("Collaboration")` # pour ajouter un élément en fin de liste, concaténer deux listes, etc.



- Les dictionnaires constituent une autre structure de données importantes. Ils sont constituées de couples clés-valeurs :

```
monDico {
      "nom" : "Marson"
      "prenom" : "Claude"
      "age" : 18 #erreur à la compilation
}
```


Des structures de données avancées

- On trouve la même dualité d'accès et de modification des dictionnaires :
- Pour obtenir la valeur associée à une clé :

```
x = monDico["age"]
```
ou
```
x = monDico.get("age")
```

et modifier une valeur :

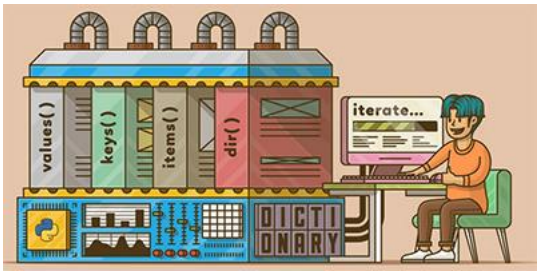
```
monDico["age"] = 35
```

- Les dictionnaires se prêtent également aux introspections par boucles for, sous la forme :

```
for x in monDico
      print(x)
```
#où tout autre traitement

On pourra, par exemple, passer par la méthode `values()` pour obtenir toutes les valeurs d'un dictionnaire :

```
for x in monDico.values() :
      print(x)
```



- on pourra vérifier si une clé existe, boucler à la fois sur les clés et les valeurs avec la méthode `items()`, obtenir le nombre de couples présents dans le dictionnaire avec la méthode `len()`, ajouter ou supprimer simplement des couples, voire même se servir d'un dictionnaire existant pour en fabriquer un autre, avec la méthode `dict()`.

Les itérateurs

- L'itération est très importante avec Python
- Réponse aux déplacements dans les structures, tableaux, dictionnaires…
- Les boucles for et les itérations constituent un élément clé de Python.
- Chacun des objets itérables dispose d'un itérateur, un objet particulier, qui permet de balayer ses valeurs. On peut fabriquer par héritage, des itérateurs spécifiques.
- L'itérateur n'est pas toujours obligatoire pour parcourir les éléments d'une liste, car on peut toujours se contenter d'un **while**, mais il est très pratique et contribue à l'esthétisme du code, voire il peut être la seule solution pour introspecter directement le contenu d'une collection.
- Les objets itérables sont parcourus par une boucle for de la forme :

```
chaine = 'Bonjour LeMarson'
iterateur = iter(chaine)
for i in range(len(chaine)):
    print(iterateur.next())
```

- Dans ce code, l'objet iterateur appelle la méthode **iter** : tous les itérateurs des listes itérables, en possèdent une, de même qu'une méthode **next**, iterateur devenant l'objet de scrutation de chaine.
- L'activité du for est bornée par un indice que l'on fait varier, de 1 à **len(chaine)**, la fonction **len** (une « built-in function » qui précise la longueur de la chaîne) et par **next**, qui va se charger d'incrémenter le compteur.
- Rien d'extraordinaire, mais l'itérateur nous cache une bonne partie des tâches et il est indépendant de la structure interne de l'objet itérable.
- Outre les itérateurs exposés de manière standard avec le langage, le développeur peut écrire son propre objet d'itération.
  - Il doit implémenter les méthodes **_iter_** et **_next_** et prévoir l'initialisation **init**. Cela lui permettra par exemple de prévoir des conditions de démarrage, un incrément de 6 pour une suite numérique (next) et des traitements particuliers.

L'atout de la modularité

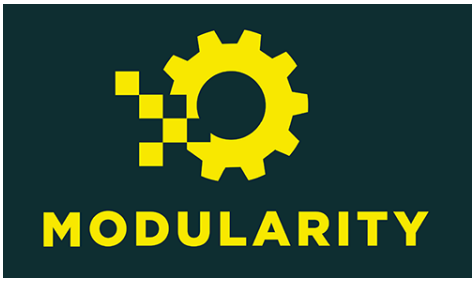
- Une fonction est un bloc de code qui traite d'une tâche particulière. Elle peut comporter des arguments, mais ce n'est pas obligatoire et retourne une ou plusieurs valeurs.
- Il y a dans Python, deux natures de fonctions, celles que l'utilisateur va écrire et définir avec le mot clé `def` :

```
def somme (num1, num2…)
    return (num1 + num2)
```

- et celles qui sont proposées en standard, les « built-in », nombreuses et diversifiées dans l'usage, qui constituent une bibliothèque de base très appréciée.
- 70 fonctions : `complex()`, qui traduit un réel en nombre complexe, `dict()`, `dir()`, `enumerate()`, `filter()`, `hash()`, utilisé pour comparer les clés de dictionnaires, `id(object)` qui retourne l'identité d'un objet, unique pendant toute sa durée de vie, `map(function, iterable)` qui applique la fonction à chaque élément de l'itérable, `open()`, `pow(x,y)` qui élève x à la puissance y, `print()`, etc.
- Une méthode Python, ressemble beaucoup à une fonction, sauf qu'elle n'existe que dans un contexte objet, construit à partir d'une classe et n'accède qu'aux données exposées dans la classe.
- Une troisième nature de fonctions, dites `lambda`, définies par l'utilisateur, mais par le mot clé `lambda` et non pas `def`.
- Un `module` est un fichier qui contient du code et pour utiliser ce code, il faut passer par la fonction `import`.
- Plusieurs modules peuvent être regroupés dans des `packages`. Chaque package comportant systématiquement un fichier `__init__.py`, qui indique qu'il s'agit d'un package.

L'atout de la modularité

- Le module `math` avec un très grand nombre de fonctions mathématiques, son alter égo `cmath` étant dédié aux nombres complexes.
- 16 fonctions arithmétiques telles que `math.factorial(x)` qui retourne la factorielle de x (multiplication des x premiers nombres entiers) ou `math.gcd(x,y)` qui donne le PGCD (Plus grand Commun Diviseur) de x et y, mais aussi 8 fonctions logarithmiques et exponentielles, telles que `math.exp(x)`, `math.log10(x)` ou `math.sqrt(x)`, huit fonctions trigonométriques comme `math.acos(x)`, `math.hypot(x,y)` qui calcule la norme euclidienne sqrt(x*x + y*y), utilisée dans les fonctions d'erreurs d'IA, etc. ou deux fonctions de conversion angulaire, six fonctions hyperboliques, dont `math.atanh(x)` et `math.acosh(x)` et quatre fonctions spéciales.
- Le module `os` (pour Operating System) est un autre standard très employé, qui permet de traiter les variables shell, le lancement des programmes, la gestion des processus, la gestion des fichiers/verrous des descripteurs, le système de gestion de fichier lui-même, sans oublier les outils d'administration, de portabilité et de gestion des chemins.
- Le module `pickle` est dédié aux fonctions de sérialisation et désérialisation de certains objets Python, `random` est utilisé pour générer des nombres aléatoires, `re` pour effectuer des opérations sur des expressions régulières et `socket` pour les communications en mode socket.
- Un autre module est particulièrement important dans la gestion du système et les interactions avec la machine ou les capteurs, `sys` qui donne accès à 90 variables système.
- Python dispose également de modules liés au temps : `time`, `datetime`, `calendar`, etc.



Les ingrédients de programmation fonctionnelle

- Python peut être considéré comme un langage fonctionnel, mais avec quelques raccourcis :
  - Fonction pure.
  - Immutabilité, qui va de pair avec la pureté. Sans effet de bord.
  - Propriété de transparence référentielle.
  - Types et des structures de données de haut niveau, tels que des listes extensibles.
  - La récursivité, une fonction peut s'appeler elle-même.
  - La « composabilité » : composition de plusieurs fonctions, pour produire une autre fonction.
- Entorse : en Python, les listes, sets, objets et dictionnaires sont mutables et il n'y a guère que les tuples, les chaînes et les entiers qui ne le sont pas (une variable est mutable quand on ne peut la modifier que par une affectation).
- Les fonctions Python sont aussi des « citoyens de première classe », ce qui est considéré comme un pré-requis pour la programmation fonctionnelle, de sorte que l'on peut manipuler les fonctions elles-mêmes, les passer en paramètres, les retourner, etc.

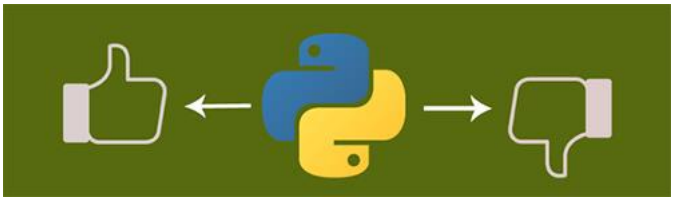
Les gestionnaires de paquets

- Comme Node et d'autres langages script, Python dispose de son propre gestionnaire de paquets pip (« Pip Installs Packages » ou « Pip Installs Python », qui permet d'installer et de gérer des paquets Python, la plupart d'entre eux pouvant être trouvés sur la plate-forme PyPI, le dépôt officiel du langage, avec plus de 100 000 programmes , utilisables dans tous les domaines, mais principalement en IA et science des données.
- Depuis les versions 2.7.9 et 3.4, pip est inclus par défaut

Multi-plateforme

- Le fait que Python soit disponible sur plusieurs plates-formes est présenté comme un avantage.
- Les implémentations les plus connues :
  - **Jython** (ex **JPython**), un compilateur Python écrit par Jim Hugunin en 1997, le code compilé étant du .class Java, susceptible d'hériter de classes Java
  - **IronPython**, l'équivalent dans l'environnement .NET, écrit par le même Jim Hugunin
  - **RubyPython,** une passerelle entre les interpréteurs Python et Ruby, pour enrober, convertir et appeler des objets Python dans un code Ruby
  - **PyObjc**, une passerelle entre les environnements Python et Objective-C.
  - **Brython**, qui veut remplacer JavaScript par Python dans un environnement HTML 5
  - **PyPy**, un interpréteur Python écrit avec un sous-ensemble de Python et compilable en C ou LLVM (excellent outil)
  - **Shed Skin,** un transpiler qui compile du Python en C++
- Par ailleurs, un certain nom de distributions Python sont proposées, qui le plus souvent comportent des bibliothèques dédiées à un usage donné, chacune d'elles étant basée sur Cython, la version officielle de l'interpréteur :
  - **ActivePython**, gratuit (hors production) ou version commerciale
  - **WinPython**, une distribution basée sur Qt et Spyder, pour les scientifiques
  - **Enthought Canopy**, en deux versions, gratuite ou commerciale, pour les scientifiques
  - **Anaconda**, la plus connue, surtout chez les adeptes du « machine learning », avec une communauté de plus de 6 millions d'usagers et plus de 250 paquets pour la science des données
  - La **distribution Intel de Python**, fondée sur Anaconda, avec la bibliothèque MKL d'Intel pour accélérer les calculs numériques de NumPy et SciPy
  - **Pyzo**, destinée au grand public

Tout n'est pas parfait...



- On peut lui reprocher une certaine confusion des genres, entre l'impératif et l'objet
- Lenteur, de 2 à 2,5 fois plus lent qu'un autre langage
- N'est pas adapté aux applications mobiles
- Pas très à l'aise avec les bases de données
- Gros consommateur mémoire, conséquence de la flexibilité de ses structures de données
- Garbage collector déficient
- Nécessite beaucoup de tests, car certaines erreurs ne se voient qu'à l'exécution (typage dynamique)
- Pas adapté aux traitements multitâches, au traitement parallèle, ce qui est lié au mécanisme de GIL ("Global Interpreter Lock"), dans lequel à un instant donné, il ne peut y avoir qu'un thread
- L'usage des fonctions lambda est jugé restrictif
- On peut juger que l'indentation automatique peut aussi poser des problèmes de maintenance