# Partie 7

# Langages de programmation

17 décembre 2019

# Plan

1. Langages de programmation

2. Catégorisation des langages

3. Comparaison des langages

# Plan

1. Langages de programmation

2. Catégorisation des langages

3. Comparaison des langages

# Langages de programmation

Un langage de programmation est une notation permettant de décrire des calculs d'une manière compréhensible à la fois pour l'ordinateur et pour le programmeur.

Les premiers langages ont été inventés à la fin de seconde guerre mondiale (Plankalkül, 1945).

Depuis lors, une multitude de langages ont été inventé. Wikipedia liste ±680 langages différents.


Source

# Pourquoi autant de langages ?

- Beaucoup de langages sont spécifiques à une machine
- De nouveaux langages sont constamment introduits pour résoudre les limitations (supposées ou non) des langages existants.
- Les langages doivent suivre les évolutions des machines et les nouveaux développements en génie logiciel
- Pour des questions de hype et de marketing
- Le nombre de degrés de liberté pour le design d'un langage est très important.
- Beaucoup de langages sont spécialisés :
  web (JS, PHP, HTML), bases de données (SQL), texte (PERL, TeX), banques (COBOL), maths (Matlab, Mathematica), GPU (Cuda, OpenCL), etc.

# Pourquoi apprendre plusieurs langages ?

- Il n'existe pas de langage universel ultime qui conviendrait pour tout.
- Pour étoffer sa boîte à outils et aborder plus efficacement de nouveaux problèmes.
- Pour s'adapter à un domaine/environnement de travail, pour pouvoir réutiliser/améliorer les outils de ce domaine.
- Pour apprendre d'autre paradigmes de programmation.
  - Le langage utilisé façonne le processus de raisonnement
  - Différents problèmes sont résolus de manière efficace par différentes approches.
- Tant qu'on n'a pas testé un langage, on ne sait pas ce qu'on manque.
- Pour le challenge intellectuel.

# Plan

# Catégorisation des langages

Les langages sont généralement caractérisés selon un ensemble de propriétés, par exemple :

- Expressivité
- Niveau d'abstraction
- Paradigme de programmation
- Processus de compilation/exécution des programmes
- Approche de gestion de la mémoire
- Système de typage des variables/valeurs
- ...

Le **C** est un langage turing complet, de bas niveau, à typage statique faible, de type impératif procédural, compilé et dont la mémoire est gérée manuellement.

# Expressivité

Le C est un langage...**Turing-complet**...

La théorie de l'informatique étudie la classe des fonctions qui sont calculables par une procédure effective, c'est-à-dire un suite finie d'instructions qui renvoie toujours une réponse correcte.

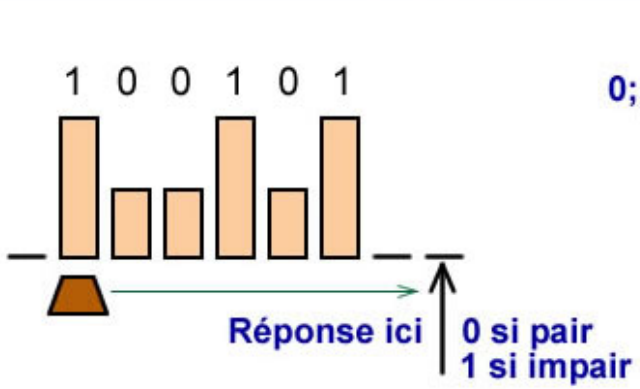
Thèse de Turing-Church : une fonction est calculable par une procédure effective si et seulement si elle est calculable par une machine de Turing (voir le slide suivant)

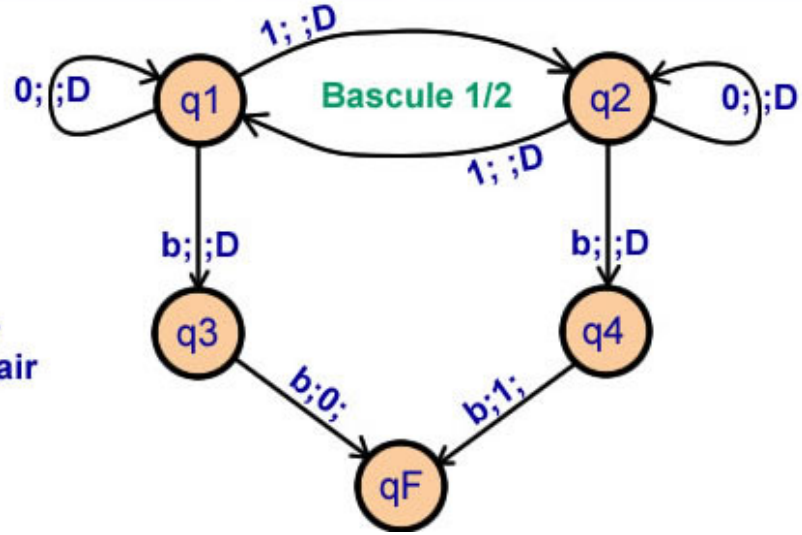
Un langage de programmation est Turing-complet s'il est au moins aussi expressif qu'une machine de Turing.

Pas déterminant dans le choix d'un langage car la plupart des langages pratiques sont Turing-complet.

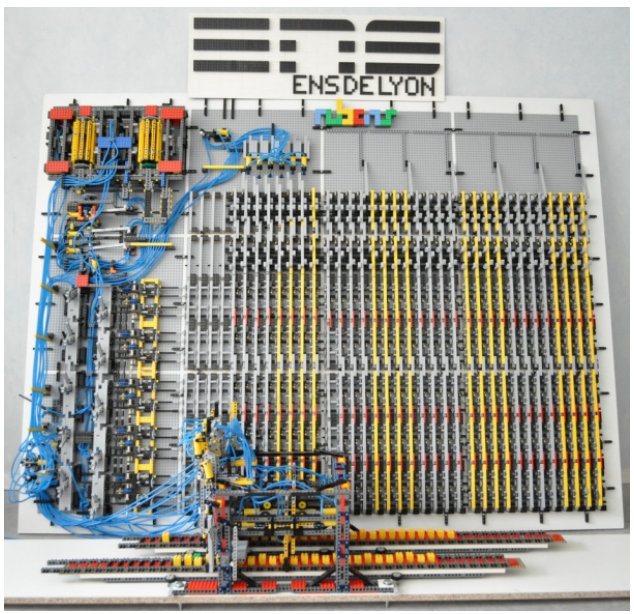
# Machine de Turing

Programme calculant la parité du nombre de 1 dans une séquence :




Source : Le monde, 2017




Source : Wikipedia

# Niveau d'un langage

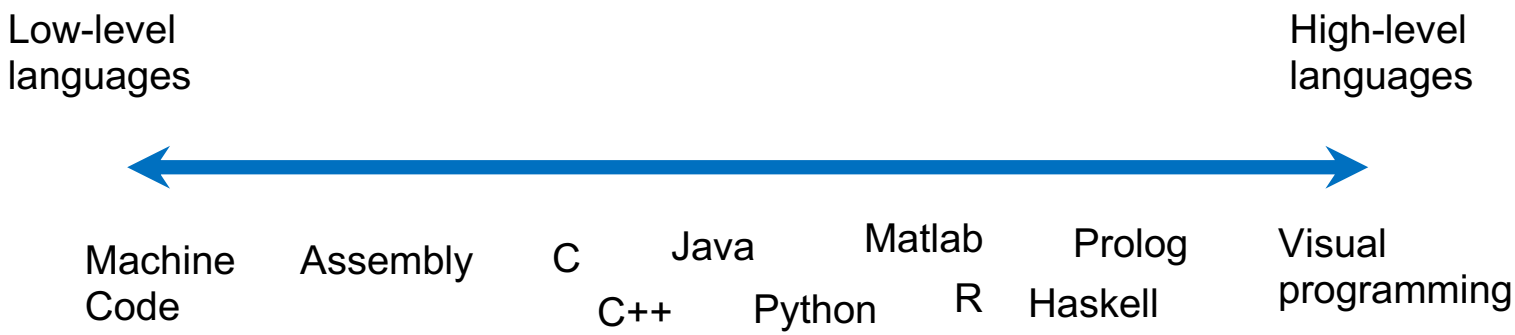
C est un langage...**de (relativement) bas niveau**...

Un langage est d'autant plus de **bas niveau** qu'il est proche du langage machine

- Contrôle fin de l'exécution et efficacité mais programmation plus laborieuse.

Un langage est d'autant plus de **haut niveau** qu'il est proche du langage humain

- Programmation plus facile mais contrôle et efficacité moindre.



# Paradigmes de programmation

Le C est un langage...**impératif procédural**...

Il existe de multiple styles (= paradigmes) de programmation

Principaux paradigmes :

- Impératif :
  - Procédural : C, fortran...
  - Orienté objet : C++, Java...
- Déclaratif :
  - Fonctionnel : Lisp, Scheme, OCaml, Haskell...
  - Logique : Prolog, Oz, Coq...

Historiquement, les langages étaient mono-paradigmes. De nos jours, la plupart des langages sont multi-paradigmes, même si tous les paradigmes ne sont pas implémentés avec la même efficacité.

# Programmation impérative procédurale *C, fortran...*

La programmation impérative consiste à décrire les calculs comme une série de commandes/instructions modifiant directement l'état du programme (les valeurs stockées dans la mémoire).

Procédural signifie que les séquences d'instructions sont organisées en procédures, fonctions ou sous-routines qui peuvent s'appeler les unes les autres.

Paradigme très populaire, généralement des langages simples de bas niveau. Pas adapté pour des projets de grandes envergures.

# Programmation impérative orientée objet *C++, Java...*

Impérative : le programme modifie directement l'état du programme (les champs associés aux objets).

Orientée objet : la programmation est centrée sur les (types de) données.

- On identifie les "choses" qui font partie du problème ou de la solution.
- On décrit ces choses par des variables (champs).
- Ces choses interagissent avec le monde via des méthodes.

Programmation orientée "nom", plutôt que "verbe", contrairement au procédural.

Basé sur l'encapsulation : les données (objets) et les méthodes associées sont définies conjointement (=classes) et l'accès direct aux données est interdit (= types opaques).

Très bien adaptée à de gros projets collaboratifs. Très utilisée dans l'industrie.

# Exemple en C

L'utilisation de types de données abstraits et de types opaques en C permet d'imiter certains mécanismes de la programmation orientée objet.

complex.h

```
typedef struct Complex_t Complex;

Complex *complex_new(double, double);
void     complex_free(Complex *);

double   complex_re(Complex *);
double   complex_im(Complex *);

Complex *complex_plus(Complex *, Complex *);
Complex *complex_times(Complex *, Complex *);
double   complex_abs(Complex *);
...
```

complex.c

```
#include <math.h>
#include "complex.h"

struct Complex_t {
  double re, im;
};

Complex *complex_new(double re, double im) {
  Complex *c = (Complex *)malloc(sizeof(Complex));
  c->re = re;
  c->im = im;
  return c;
}

double complex_re(Complex *a) {
  return a->re;
}

Complex *complex_plus(Complex *a, Complex *b) {
  double real = a->re + b->re;
  double imag = a->im + b->im;
  return complex_new(real, imag);
}
...
```

# Exemple en Java

## Complex.java

```
public class Complex
{
  private final double re; // type opaque
  private final double im;
  public Complex(double real, double imag)
  {  re = real; im = imag;  }
  public Complex plus(Complex b)
  {  // Return the sum of this number and b.
    double real = re + b.re;
    double imag = im + b.im;
    return new Complex(real, imag);
  }
  public Complex times(Complex b)
  {  // Return the product of this number and b.
    double real = re * b.re - im * b.im;
    double imag = re * b.im + im * b.re;
    return new Complex(real, imag);
  }
  public double abs()
  {  return Math.sqrt(re*re + im*im);  }
  public double re()  { return re; }
  public double im()  { return im; }
  public String toString()
  {  return re + " + " + im + "i";  }
}
```

## Exemple d'utilisation

```
Complex z0 = new Complex(1.0, 1.0);
Complex z = z0;
z = z.times(z).plus(z0);
z = z.times(z).plus(z0);
StdOut.println(z);
```

Remarques :

- `new` replace le `malloc`. Pas besoin de `free` (voir plus loin)
- Méthodes directement associées aux objets ($\neq$ procédures appliquées aux objets).
- En Java, pas de pointeurs.

# Programmation fonctionnelle *Lisp, Scheme, OCaml, Haskell...*

Approche plus abstraite de la programmation. on décrit quoi calculer plutôt que comment le calculer (= déclaratif).

Un programme calcule en évaluant des fonctions, au sens mathématique du terme.

Caractéristiques principales :

- Fonctions d'ordre supérieur : fonctions pouvant prendre comme arguments des fonctions et/ou en renvoyer en sortie.
- La récursion est utilisée pour la construction de boucles.
- Les listes sont la structures de base (map/reduce/filter)
- Pas d'effets de bord et d'état partagé ($\neq$ impératif)

Avantages : idempotence des fonctions, optimisation, test et déboguage plus aisés, permet le calcul concurrent, concision, etc.

Pas "mainstream" mais des touches de fonctionnelles sont présentes dans beaucoup de langages multi-paradigmes (Python, Rust, Swift, Javascript...).

# Fonctions d'ordre supérieur en Python

Python n'est pas un langage purement fonctionnel mais il en a quelques caractéristiques.

Fonctions d'ordre supérieur :

```
def maxVal(f, g, x):
    return max(f(x), g(x))

def f(x):
    return x+2

g = lambda x: 6

maxVal(f,g,3) #=> 6
```

```
def maxFun(f, g):
    def maximumFunction(x):
        return max(f(x), g(x))
    return maximumFunction

maxFun(f,g)(3)    #=> 6
mv = maxFun(f,g)
mv(3)             #=> 6
```

- `lambda` permet de définir une fonction non nommée.
- MaxFun renvoie une fermeture (*closure*), une fonction avec des variables liées (ici, `f` et `g`).

# Listes, map, filter et reduce

```
l = [2, 18, 9, 22, 17, 24, 8, 12, 27]
print(list(filter(lambda x: x % 3 == 0, l)))
#[18, 9, 24, 12, 27]
print(list(map(lambda x: x * 2 + 10, l)))
# [14, 46, 28, 54, 44, 58, 26, 34, 64]
print(reduce(lambda x,y: x + y, foo))
# 139
```

- `map(f,l)` applique `f` à tous les éléments de `l`, `filter(f,l)` renvoie la liste des éléments tels que `f(x)==True`.
- `reduce(f,[x1,x2,...,xN])` renvoie

$$f(f(...f(f(f(x1,x2),x3),x4),...),xN)$$

- `map` et `filter` sont évalués paresseusement, c'est-à-dire seulement quand on y accède (via `list` par exemple).

# Test de primalité en Python : approche impérative

Un nombre naturel $n$ est premier si et seulement si :

$$\nexists k \in [2, n[ : n \bmod k = 0.$$

Solution impérative procédurale en python :

```
def is_prime(n):
    k = 2
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True
```

Liste d'instructions à exécuter pas à pas, effets de bords locaux (incrémentation de `k`).

# Test de primalité en Python : approche fonctionnelle

```
def is_prime(n):
   return not any(filter(lambda k: n%k==0, range(2,n)))

def primes(m):
   return list(filter(is_prime, range(1,m)))

print(primes(20))
#[1, 2, 3, 5, 7, 11, 13, 17, 19]
```

Remarques :

- `range(n,m)` renvoie la liste `[n,n+1,...,m-1]`.
- `any` renvoie `True` si au moins une valeur (non `False`) dans la liste, `False` si la liste est vide

Solution particulièrement compacte et lisible (car proche de la définition).

Une autre solution possible en Python, encore plus lisible :

```
def is_prime(n):
   return not any(n%k==0 for k in range(2,n))
```

# Programmation logique

Encore plus déclaratif que le fonctionnel (Rappel : on décrit le *quoi* plutôt que le *comment*).

Inclut plusieurs paradigmes :

- Programmation logique : principalement logique booléenne — *Prolog*
- Programmation sous contraintes : contraintes numériques — *Oz, AMPL*
- Assistants de preuve ("theorem provers") — *Coq, Agda, Isabelle*

Langages généralement assez inefficaces pour des tâches classiques, basés sur un algorithme de recherche/optimisation.

Plus spécialisés que les autres paradigmes.

# Exemples en Prolog : 3SUM et test de primalité

3SUM problem :

```
triplet0([A,B,C],L) :-
                member(A,L),
                member(B,L),
                member(C,L),
                all_distinct([A,B,C]),
                A+B+C =:= 0.

?- triplet0([A,B,C],[1,2,3,-2,-3,5,6,-1]).
```

Test de primalité :

```
is_prime(P) :- P < 4.
is_prime(P) :- integer(P), P > 3, not(has_factor(P,3)).
has_factor(N,L) :- N mod L =:= 0.
has_factor(N,L) :- L < N-1, L2 is L + 1, has_factor(N,L2).
```

# Exemples en Prolog : puzzle

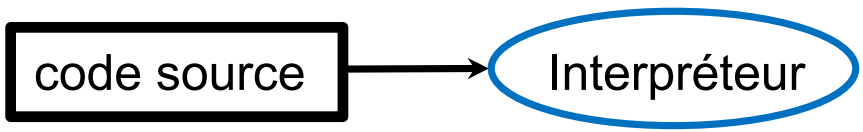
```
/* Houses logical puzzle: who owns the zebra and who drinks water?
   1) Five colored houses in a row, each with an owner, a pet,
   cigarettes, and a drink.
   2) The English lives in the red house.
   3) The Spanish has a dog.
   4) They drink coffee in the green house.
   ...
   15) The Norwegian lives near the blue house.
Who owns the zebra and who drinks water?*/
houses(Hs) :- length(Hs, 5),                                    %  1
        member(h(english,_,_,_,red), Hs),                       %  2
        member(h(spanish,dog,_,_,_), Hs),                       %  3
        member(h(_,_,_,coffee,green), Hs),                      %  4
        member(h(ukrainian,_,_,tea,_), Hs),                     %  5
        next(h(_,_,_,_,green), h(_,_,_,_,white), Hs),           %  6
        ...
        next(h(norwegian,_,_,_,_), h(_,_,_,_,blue), Hs),        % 15
        member(h(_,_,_,water,_), Hs),    % one of them drinks water
        member(h(_,zebra,_,_,_), Hs).          % one of them owns a zebra
?- zebra_owner(Owner).
?- water_drinker(Drinker).
```

# Langages compilés vs interprétés

Le C est un langage...**compilé**...

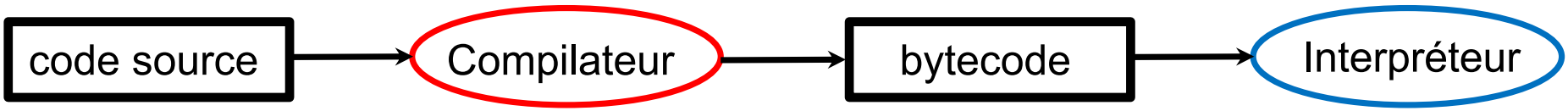
Langage compilé : un compilateur transforme le code source en un code machine qui peut être exécuté efficacement. P.ex., C, C++, fortran.


Langage interprété : un interpréteur lit et exécute (plus inefficacement) à la volée le code source. P.ex., Matlab, R, Python, Prolog.



# Approche hybride (bytecode)

Pour certains langages, un compilateur transforme le code source en un code intermédiaire, le bytecode, qui peut être exécuté plus efficacement par un interpréteur.




Exemples : Java, Python.

Avantage principal : portabilité du bytecode, plus efficace que l'interpréteur direct.

Interprétation peut être interactive (Python) ou pas (Java).

# Typage

Le C est un langage...**à typage statique faible**...

Le typage d'un langage définit comment sont gérés les types des variables :

- Statique (p.ex., C, Fortran, Java)
  - Toutes les variables ont un type.
  - Le système contrôle les erreurs de type lors de la compilation.
  - Plus contraignant mais plus efficace et prévient les erreurs.
- Dynamique (p.ex., Python)
  - Les valeurs, pas les variables, ont un type.
  - Le système contrôle les erreurs de types lors de l'exécution du code.
  - Plus souple mais moins efficace et plus sujet aux erreurs.

Le typage est fort si toutes les erreurs de type sont détectées (p.ex, Java), faible si des opérations peuvent être réalisées sur des valeurs de mauvais types (p.ex, C).

# Typage : exemple

En C :

```c
char *a = "coucou";
a = 4;                  // Génère un warning à la compilation
int c = b * a;          // Génère une erreur à la compilation
int d = b * (int)a;     // Pas de soucis
short e = d;            // Pas de soucis
```

En Java :

```java
int b = 4
short d;
d = b;                  // Génère une erreur à la compilation
```

En Python :

```python
a = 'CI'
b = 4
c = b * a           # c contient 'CICICICI'
d = 4 + 'coucou'    # génère une erreur à l'exécution
```

# Inférence de type

Certains langages à typage statique inférent le type des variables lors de la compilation. Le type des variables/arguments ne doit pas être précisé explicitement dans le code.

Exemple en OCaml :

```
let rec length = function
[] -> 0
| h::t -> 1+(length t);;

let myf() = length 4;;
```

Malgré le fait que les arguments des fonctions ne soient pas typés, ce code génèrera une erreur à la *compilation*.

# Gestion de la mémoire

Deux approches :

- Manuelle : l'utilisateur a la responsabilité d'allouer et de désallouer la mémoire
  - P.ex., `malloc` et `free` en C
  - Laborieux pour le programmeur mais il a un contrôle total
- Automatique : la mémoire doit toujours être allouée mais la désallocation est automatique.
  - L'algorithme qui s'occupe de désallouer la mémoire s'appelle le ramasse-miettes ("garbage collector").
  - Pas de fuite mémoire possible mais moins efficace et pas de contrôle de la mémoire par le programmeur

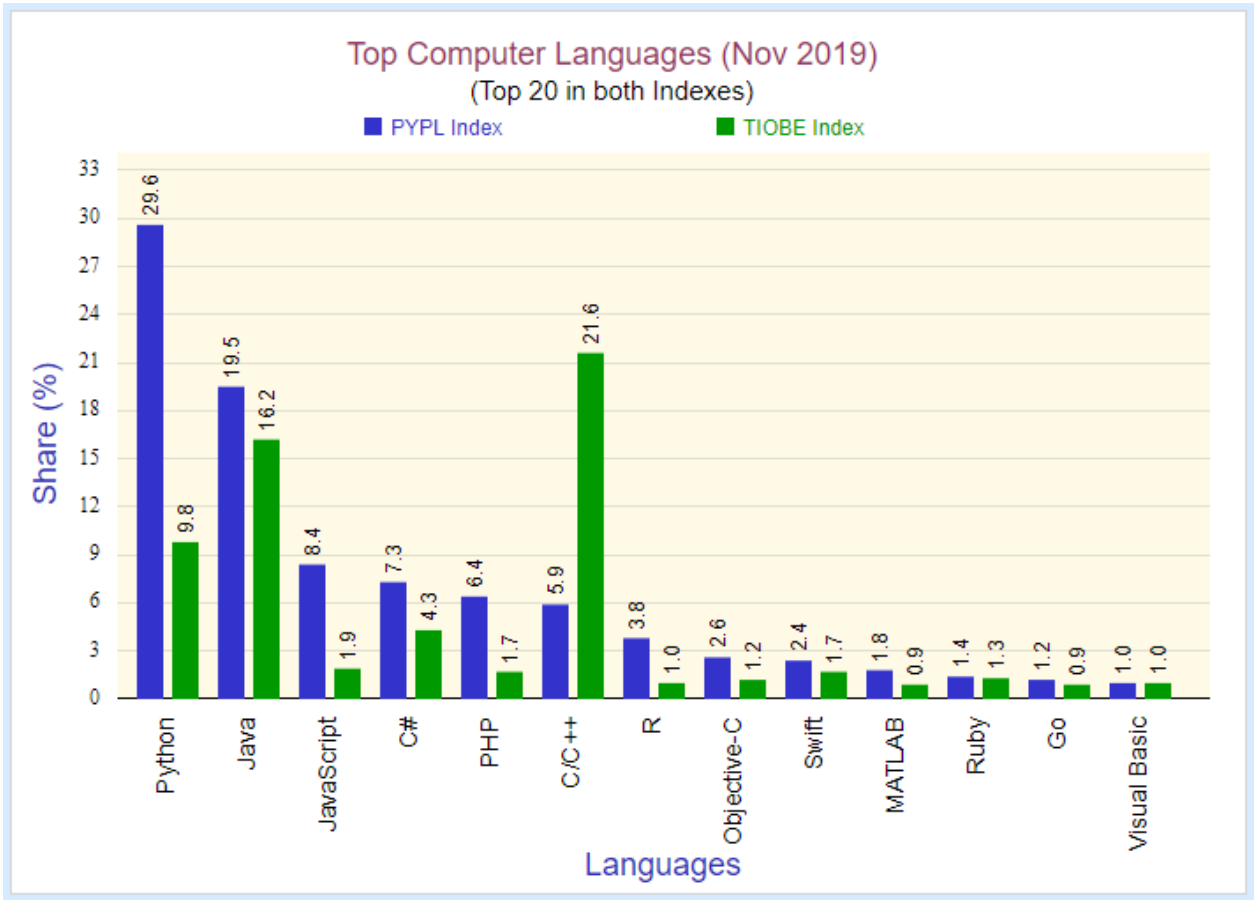
Exemple : réutilisation d'un nom de tableau, C versus Java :

```
double arr[] = malloc(5, sizeof(double));
...
free(arr);
arr = malloc(10, sizeof(double));
```

```
double[] arr = new double[5];
...
arr = new double[10];
```

# Plan

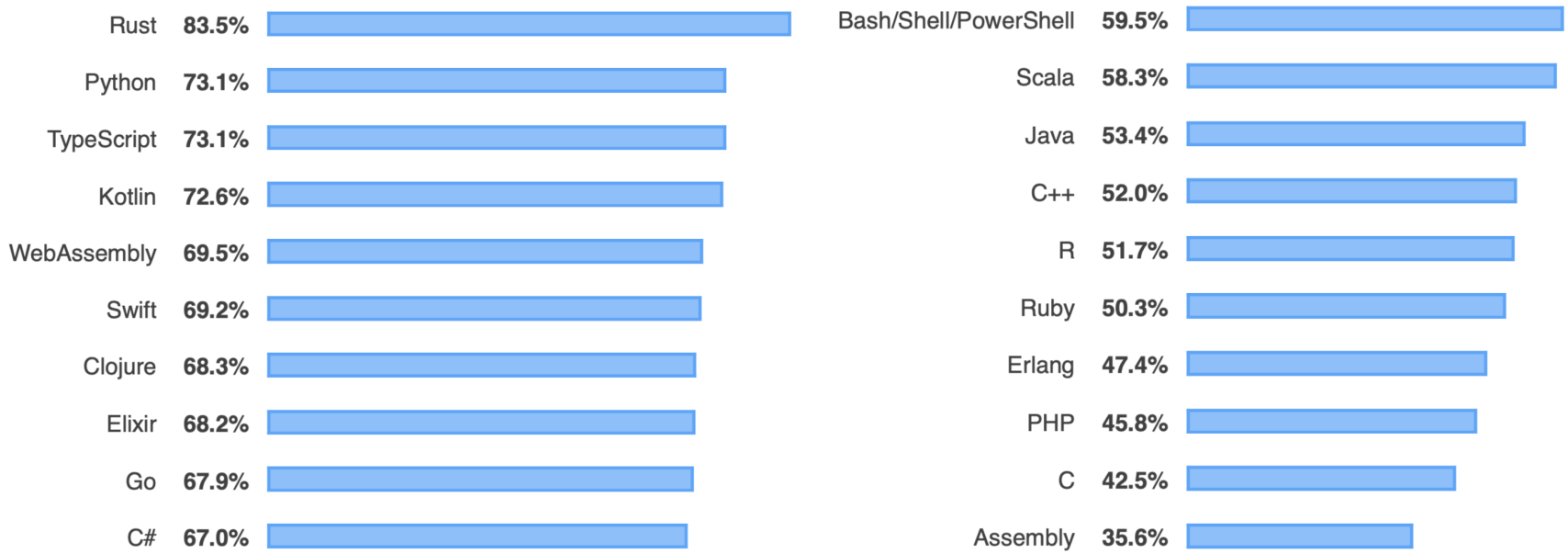
# Les langages les plus populaires




Source

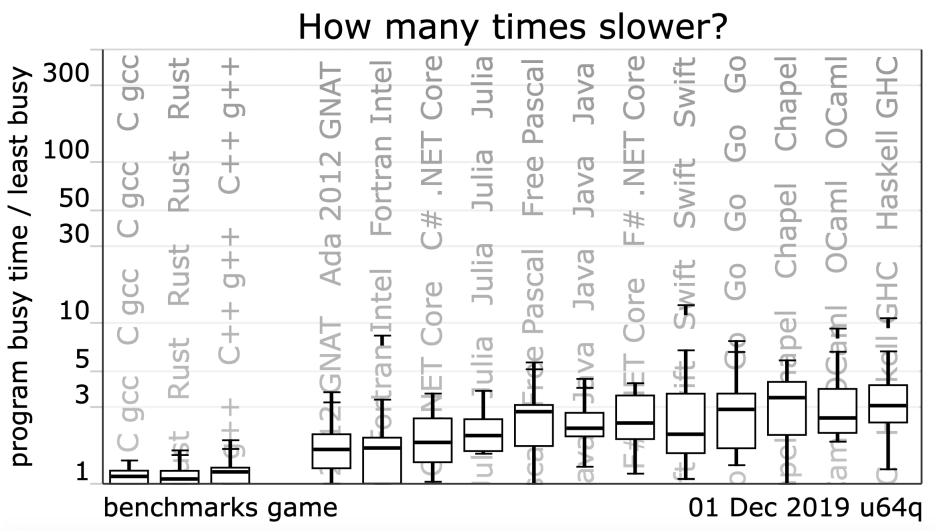
Deux indices de popularité : PYPL (recherche d'aide sur google) et TIOBE (basé sur plus de sources).

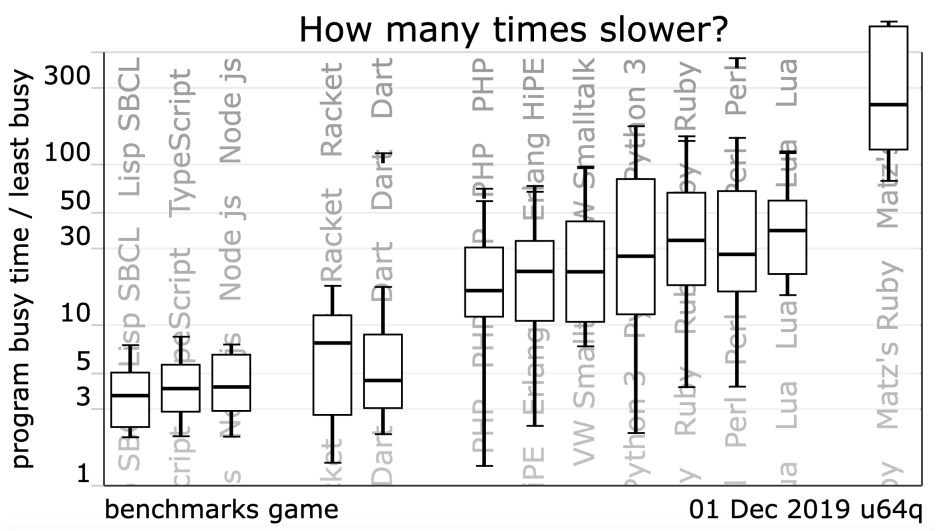
# Les langages les plus aimés




Sondage sur stackoverflow (en 2019) : pourcentage de développeurs utilisant le langage et ayant exprimé avoir l'intention de continuer de devélopper dans ce langage. (Seulement le top 10 et le bottom 10 du classement).

# Les langages les plus performants




Source : The computer langage benchmark games

- L'ordonnée montre combien de fois plus lente est la meilleure solution à chaque tâche en moyenne par rapport à la meilleur solution parmis tous les langages.
- Pas toujours le critère le plus pertinent.

# Quel langage et pour quoi ?

Parmi les langages généralistes, orientés applications scientifiques :

- C/C++ : Valeurs sûres. Efficace. C++ mieux adapté pour des gros projets.
- Java : Efficace. Très utilisé dans les entreprises de développement logiciel.
- Python : Très utilisé en recherche. Prototypage très facile, multi-paradigmes, interprété, énormément de bibliothèques externes...
- Matlab : Utilisé dans le monde académique et dans les sciences de l'ingénieur. Beaucoup de bibliothèques dédiées à l'ingénieurie. Gros défaut : payant (le seul de cette liste).
- R : Orienté sciences des données et statistiques. Interprété, beaucoup de bibliothèques et outils de visualisation.
- Julia : Orienté calcul scientifique (comme Matlab). Très efficace, gratuit. Manque de bibliothèques, en comparaison avec Python ou Matlab.

Pour les plus aventureux :

- Rust : Langage qui monte. Multi-paradigme, sécurisé, concurrent, très efficace et peu gourmand en mémoire. Manque pour l'instant de bibliothèques.
- Haskell, OCaml : Pour ceux qui veulent tester la programmation fonctionnelle.

# Programmations et langages dans les cours d'informatique

Bloc 2 :

- INFO0902 - Structures de données et algorithmes : Impérative procédurale en C.
- INFO0062 - Object-oriented programming : Orientée objet (en Java).

Bloc 3 :

- INFO0012 - Computation structures : langage d'assemblage, parallèle.
- INFO0004 - Projet de programmation orientée-objet : Orientée objet (en C++).
- INFO0054 - Programmation fonctionnelle : Fonctionnelle (en Scheme).
- INFO8006 - Introduction to artificial intelligence : Logique (un peu), projets en Python.
- INFO009 - Base de données : Langage d'interrogation de base de données (SQL).

# Dans les cours d'informatique

En master :

- INFO0016 - Introduction to the theory of computation : Théorie des langages et de la calculabilité.
- INFO0085 - Compilers : fonctionnement des compilateurs.
- INFO0049 - Knowledge representation : Programmation logique (en Prolog).
- INFO0050 - Constraint programming projects : Programmation logiques et sous contraintes.
- INFO0060 - Concurrent system verification and temporal logic : Assistants de preuves.
- ELEN0062/INFO8010 Machine an deep learning : Programmation automatique sur base d'exemples.
- ...

# Remerciements

Sources d'inspiration pour cette leçon : Cyril Soldani (présentation aux geeks anonymes), John R. Woodward, Adam Byrtek, Robert Sedgewick/Kevin Wayne.