Rust et Python

Pourquoi et comment mettre du Rust dans votre code Python

Georges Racinet

Octobus, `https://octobus.net`

Wébinaire pour les dix ans d'Anybox – 4 Novembre 2020

Présentation G. Racinet

- Cofondateur d'Anybox ! (octobre 2010)
- Développeur chez Octobus `https://octobus.net`
- Contributeur cœur Mercurial, DVCS évolué en Python
- Committeur cœur rust-cpython
- Mainteneur Heptapod (GitLab + Mercurial)
  `https://heptapod.net`

Présentation Octobus

- developpement Mercurial (cœur ou extensions)
- Heptapod
- développement et accompagnement Rust (plusieurs experts)

Une souplesse incroyable :

- Langage interprété
- Typage dynamique, tout est objet ou presque
- Gestion complète de la mémoire
- Librairie standard riche : « les piles sont fournies ».

. . . à un prix (relatif):

- lenteur générale
- consommation mémoire
- *Global interpreter lock* (GIL), empêche vrai parallélisme à haut niveau

Améliorer les performances

- Traitements répétitifs: essayer PyPy et son JIT (compilateur à la volée)
- Réécrire tout en langage compilé (C, Rust. . .)
- Extensions CPython : réécrire les parties critiques seulement

Améliorer les performances

- Traitements répétitifs: essayer PyPy et son JIT (compilateur à la volée)
- Réécrire tout en langage compilé (C, Rust. . .)
- Extensions CPython : réécrire les parties critiques seulement

Utiliser Python d'abord pour prototyper rapidement puis comme assemblage a toujours été un cas d'utilisation majeur du langage

Question fondamentale pour un projet concret

Est-ce vraiment Python le problème ?

Python est lent, mais pas tant que ça.

- Quelle est vraiment la proportion de temps passé dans le code Python ?
- Avez-vous éliminé les traitements inutiles, externalisé ce qui devrait l'être ?
- Utilisez un profiler
- Si base de données riche (SQL), gros gains par amélioration requêtes.
- Un signe : si vous en êtes à tricher avec le langage

Extensions CPython

- CPython: implémentation de référence et la plus courante du *langage* Python, écrite en C.
- Autres implémentations:
  - PyPy (Python + JIT)
  - Jython (JVM, pas de GIL)
  - GraalPython (GraalVM)
  - RustPython (en Rust !).

Une *extension* Python est un module précompilé en langage machine, utilisant directement l'API interne de CPython.
L'intérêt tourne autour de la performance:

- Vitesse d'exécution
- Sobriété en termes de mémoire
- Vrai parallélisme: relâcher le GIL

Exemples d'extensions

Une bonne partie de la librairie standard:

- `datetime`, `decimal`
- `sqlite3`
- `zlib`, `hashlib`

Quelques classiques:

- `numpy`,`scipy`
- `yaml`
- `psycopg`

Tout ce qui est bien spécifié et a besoin d'être rapide.

Rust, langage de choix pour écrire des extensions

Traditionnellement, les extensions sont écrites en C mais c'est pénible et dangereux (sécurité).
Arguments en faveur de Rust:

- langage compilé, fortement optimisé
- Sûreté mémoire (fuites, sécurité)
- *Pas* de *Garbage collector* ni de comptage de références par défaut
- Emphase sur le parallélisme
- ABI compatbile avec C
- Librairie standard riche : les piles sont aussi livrées avec !

Rust: les règles mémoire

- Toute donnée a un propriétaire unique, identifié à la compilation
- Une donnée peut-être « prêtée » sous forme de références partagées (`&truc`) ou mutables (`&mut truc`)
- Impossible d'avoir une référence mutable en même temps qu'une autre référence

Intérêts :

- Sûreté / sécurité
- Absence de fuites mémoire
- Ce qu'il faut pour parallélisme sain avec sérénité
- Garantie à la compilation: optimisation agressive et meilleur passage à l'échelle des grands projets que C

Exemple Mercurial: itération sur les ancêtres

```
import heapq

def iter_ancestors(parentfn, initrevs):
  seen = {nullrev}
  seen.update(initrevs)
  visit = [-r for r in initrevs]
  heapq.heapify(visit)
  while visit:
      current = -visit[0]
      yield current
      p1, p2 = parentfn(current)
      if p1 not in seen:
          heapq.heapreplace(visit, -p1)
          seen.add(p1)
      else:
          heapq.heappop(visit)
      heapq.heappush(visit, -p2)
      seen.add(p2)
```

Sur-optimisation, un signal clair pour passer à autre chose

On était déjà au max. En vrai, le code commençait ainsi:

```
def iter_ancestors(parentfn, initrevs):
    seen = {nullrev}
    see = seen.update
    heappush = heapq.heappush
    ...
```

Le simple fait de placer `seen.update` et `heapq.heappush` en variables locales apporte un vrai gain dans le cas de Mercurial. À ce stade **on se bat contre le langage** .

Traduction en Rust

Type et constructeur

```
pub struct AncestorsIterator<G: Graph> {
    graph: G,
    visit: BinaryHeap<Revision>,
    seen: HashSet<Revision>,
}

impl AncestorsIterator<G: Graph> {
  pub fn new(
      graph: G,
      initrevs: impl IntoIterator<Item = Revision>
  ) -> Self {
        let visit: BinaryHeap<Revision> = initrevs.collect();
        let seen = visit.iter().cloned().collect();
        seen.insert(NULL_REVISION);
        AncestorsIterator {visit: visit, seen: seen};
    }
}
```

Traduction en Rust

Itération

```rust
impl<G: Graph> Iterator for AncestorsIterator<G> {
  type Item = Result<Revision, GraphError>;

  fn next(&mut self) -> Option<Self::Item> {
      let current = match self.visit.peek() {
          None => { return None; }
          Some(c) => *c,
      };
      let [p1, p2] = self.graph.parents(current)
          .map_err(|e| Some(e))?;
      if self.seen.insert(p1) {
          *(self.visit.peek_mut().unwrap()) = p1;
      } else {
          self.visit.pop();
      }
      if self.seen.insert(p2) {
          self.visit.push(p2);
      }
      Some(Ok(current))
  }
```

Réimplémentation Rust pur: conclusion

Bas-niveau dans le résultat, haut-niveau dans l'expressivité

- abstractions à coût d'exécution nul :
  - généricité (paramètres de types comme `<G>`)
  - interfaces standard: les *traits* comme `Iterator`
- librairie standard riche :
  - `set` → `HashSet`
  - `heapq` → `BinaryHeap`
- code écrit chez Anybox (octobre 2018) ;-)

Maintenant, comment brancher tout ceci à Python ?

Options FFI pour les extensions Rust

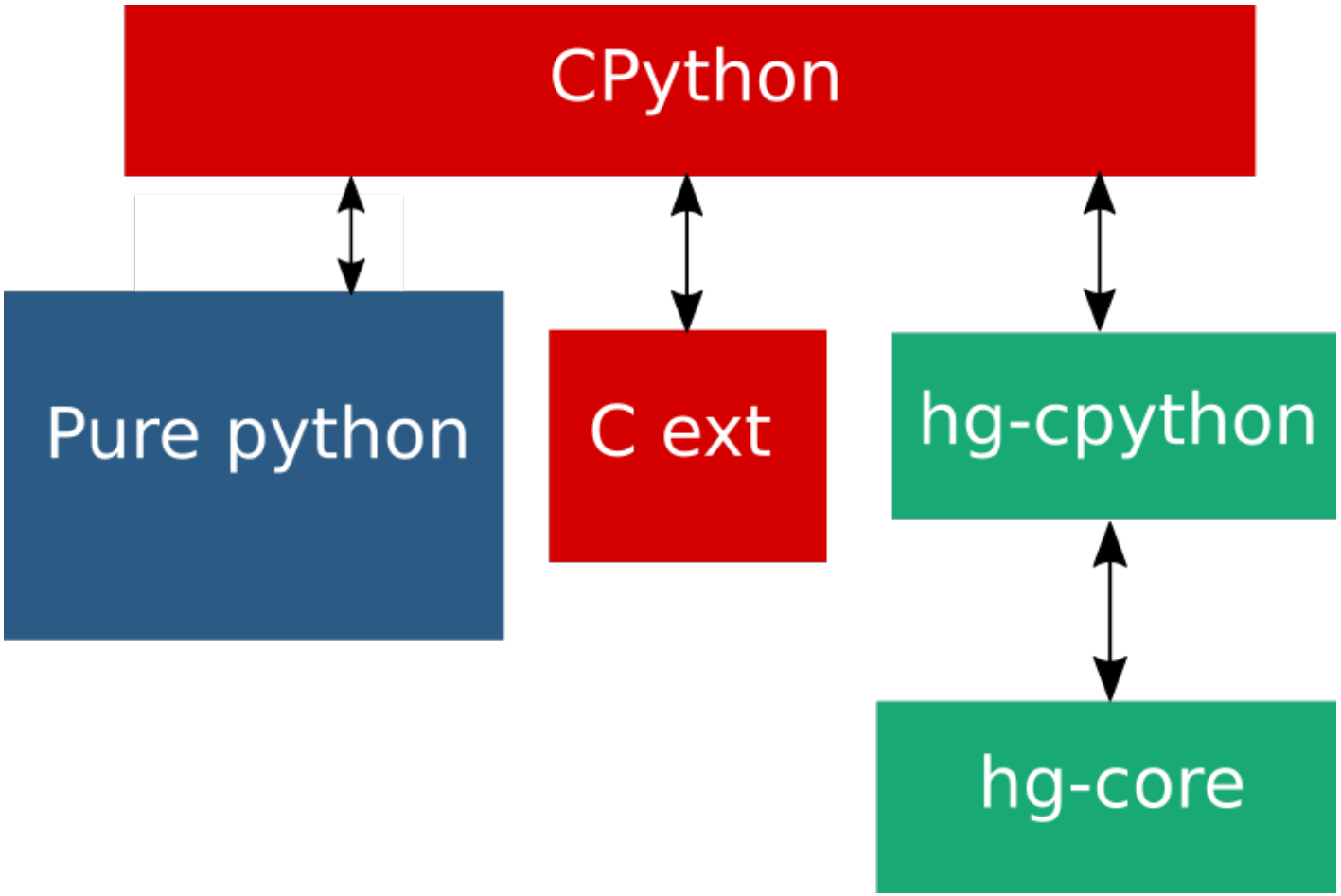
On a le choix !

- écrire une extension C qui fait le pont
- rust-cpython
- PyO3 : *fork* de rust-cpython, a beaucoup évolué depuis
- cffi
- HPy (expérimental) : le futur !

Dans tout les cas, séparer en deux couches:

- librairie (*crate*) en Rust pure
- partie (autre *crate*) assurant l'intéropérabilité.

Options FFI : tableau complet pour Mercurial



Les échanges directs entre Rust et C sont également souhaitables et possibles, mais ne seront pas abordés ici (`PyCapsule`)

rust-cpython

- expose les object CPython en structures Rust
- traduction automatique des types de base (`int`, `set`)
- des macros pour définir directement classes et modules Python
- acquisition du GIL garantie statiquement à la compilation avec un pseudo-object marqueur grâce aux règles du langage (!)

Exemple:

```
use cpython::{Python, PyResult, PyDict, py_class};

py_class!(class MyType |py| {
    data number: i32;
    def __new__(_cls, arg: i32) -> PyResult<MyType> {
        MyType::create_instance(py, arg)
    }
    def half(&self) -> PyResult<i32> {
        Ok(self.number(py) / 2)
    }
});
```

Défi: le flux de données

Comment gérer les grands transferts de données entre les parties Rust et Python de l'application ?

Le risque est de reperdre à l'interface tout ce que l'on a gagné en écrivant du Rust: que tout le temps soit passé à créer des objects Python.

Flux de données: conversions

- Exemple : `set` ↔ `HashSet`
- C'est en fait une copie
- Facile à implémenter et à manipuler
- Peut s'avérer très inefficace.
  Imaginer un traitement dont le résultat est d'ajouter un élément à un ensemble de taille un million...
- Règle grossière: acceptable comme étape intermédiaire et pour les petites données.

Flux de données: réduction au maximum

Exemple:
« Quel est l'âge du capitaine ?
– (lourd traitement) 42 ! »

- la solution ultime
- peut être contradictoire avec le problème à résoudre.
- suppose d'avoir traduit suffisamment de processus complexes, on a souvent besoin d'étapes intermédiaires raisonnables.

Flux de données: encapsulation

- Nul besoin de convertir/copier d'énormes objets
- Réacquérir le GIL à chaque utilisation (lent) ou le garder (tue le parallélisme)
- Nécessite de définir des abstractions compliquées pour ne pas polluer tout le code avec du spécifique Python (GIL encore)

Flux de données: c'est Rust qui gère

Les objets complexes traversant l'interface sont entièrements implémentés en Rust. Exemple : on fournit à Python une classe `RustSet` suffisamment compatible avec `set`

- La partie Python n'utiliserait (quasi) plus que des `RustSet`.
- Le GIL ne sert plus que pour l'objet entier, à l'interface.
- Effort supplémentaire, rebutant si peu de valeur ajoutée
- Problème: les règles de Rust interdisent d'utiliser le même objet deux fois...

Références partagées entre Rust et Python

- problème: si on a une référence permettant la mutation d'un objet Rust, toute autre référence est interdite.
- exemple : si l'on implémente un `RustSet` visible depuis Python, impossible de donner à Python des itérateurs dessus.
- solution: rust-cpython fournit la notion de `shared_data`.
- historique:
  - premier prototype G. Racinet (juin 2019)
  - améliorations R. Gomès (été 2019)
  - cas général avec compteur générationnel et empoisonnement de références M. Thomas, Y. Nishihara (octobre 2019).

Un avantage substantiel de `rust-cpython` sur les autres solutions

- Permettre aux extensions de ne plus manipuler directement des pointeurs `*PyObject` mais des « *handles* » abstraits.
- Poussé par développeurs cœur CPython et par communauté PyPy
- Exemple : PyPy et son JIT plus extensions natives (déjà possible mais très compliqué)
- Le rêve : PyPy + extensions en Rust (essais préliminaires par G. Racinet au sprint PyPy 2020)

Références

- Extensions CPython: https://docs.python.org/3/extending/index.html
- rust-cpython: https://crates.io/crates/cpython
- Mercurial: https://www.mercurial-scm.org
- PyPy: https://pypy.org
- HPy: https://github.com/hpyproject/hpy
- Sprint PyPy (HPy) 2020 https://morepypy.blogspot.com/2020/03/leysin-2020-sprint-report.html
- Références partagées entre Rust et CPython: https://octobus.net/blog/2019-07-25-rust-cpython-shared-ref.html
- Présentations Rust et Python par R. Gomès (Octobus) à FOSDEM 2020: https://archive.fosdem.org/2020/schedule/speaker/raphael_gomes/

Fin

À vous pour les questions