PYTHON DANS LE MONDE PROFESSIONNEL

Alexis RICHTER alr@atolcd.com

atol Conseils & Développements

SOMMAIRE

PYTHON DANS LE MONDE PROFESSIONNEL

VERSIONS

Date	Commentaire
29/06/2021	Version Initiale
16/07/2021	Corrections des coquilles et ajout de quelques précisions sur #Performance, #Haut-Niveau et #Témoignages

PRÉAMBULE

Ce document a été produit en tant qu'accompagnement d'une présentation effectuée à l'université de Bourgogne par l'entreprise Atol Conseils & Développements. Si vous souhaitez remonter une erreur, partager des points d'améliorations ou tout simplement nous faire part de vos commentaires, contactez communication@atolcd.com.

CHAPITRE 1

30 ANS DE PYTHON ET UNE POPULARITÉ CROISSANTE

PYTHON DANS LE MONDE PROFESSIONNEL

La première version de Python (0.9.0) fut déployée le 20 février 1991, en Open Source, même si le terme «Open Source» n'avait pas encore été défini à l'époque. Nous fêtons donc les 30 ans de Python.

Nommé d'après les Monty Python, et non pas d'après le serpent, Python fut créé par Guido van Rossum en tant que projet pour l'occuper autour des fêtes de Noël. Mais le projet prend vite de l'ampleur.

Une préoccupation majeure de Guido van Rossum dans le développement de Python, c'est de placer le développeur au centre de l'expérience.

> ***The mainframe is a machine that costs many millions of dollars, and the combined pay of all those programmers is peanuts compared to the cost of the mainframe, but as I experienced desktop workstations and PCs, I realized that a change of mindset about cost of the programmer's time versus cost of the computer's time was overdue.***
>
> DROPBOX BLOG, GUIDO VAN ROSSUM

Il restera Benevolent Dictator For Life (BDFL) jusqu'en 2018. L'ironie de quitter une position attribuée «For Life» n'échappe pas à Guido van Rossum comme on peut le voir dans cette vidéo où il retrace l'histoire de Python. Il ne nommera volontairement pas de successeur et préfèrera laisser le choix du système de gouvernance au noyau de l'équipe de développeurs.

> ***So what are you all going to do? Create a democracy? Anarchy? A dictatorship? A federation?***
>
> TRANSFER OF POWER, GUIDO VAN ROSSUM

Après réflexion, le Steering Council sera créé. Guido van Rossum accompagnera le Conseil en 2019, mais ne se représentera pas en 2020.

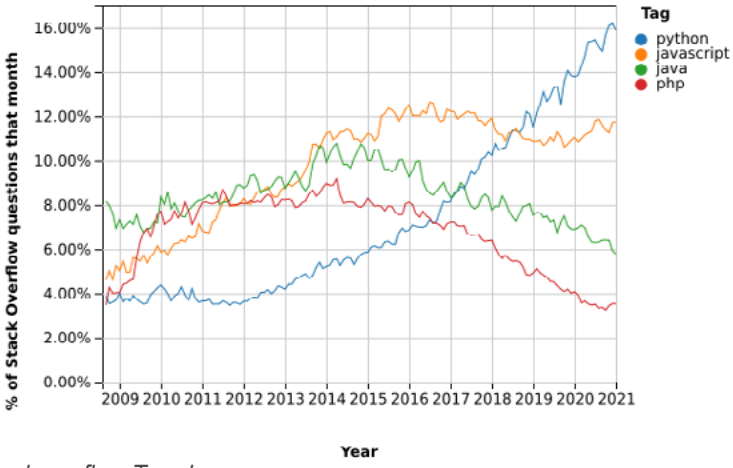
Ce qui est intriguant dans l'histoire de Python, c'est sa forte croissance en popularité. Et surtout ces dernières années. Car malgré une transition difficile, de la version 2 à la version 3 qui aurait pu compromettre l'avenir du langage, aujourd'hui, Python est placé parmi les langages les plus populaires. Notamment dans le classement RedMonk de juin 2020 :

> ***(...) Python is the first non-Java or JavaScript language ever to place in the top two of these rankings by itself, and would not have been the obvious choice for that distinction in years past. Underrated and often overlooked, the versatility of the language remains both its calling card and the basis for its continued strength. (...) As long as it remains a language of first resort, it will continue to perform well in these rankings.***
>
> REDMONK RANKING, JUNE 2020

C'est la première fois depuis que RedMonk génère ses statistiques (2012) qu'un langage autre que Java ou JavaScript prend la seconde place du classement. Bien sûr, ce type

de classement est fortement biaisé par les métriques utilisées. RedMonk, par exemple, se base sur le nombre de questions posées sur StackOverflow et le nombre de projets écrits sur Github dans un langage donné pour mesurer la popularité.

Mais, ce qu'il y a d'intéressant, c'est que ces données semblent être corroborées par d'autres sondages comme celui de StackOverflow. Déjà en 2017, David Robinson, ancien Data Scientist chez StackOverflow, se sentait dans l'obligation d'écrire un article sur l'incroyable croissance de Python. Et à en croire StackOverflow Trends, la popularité de Python n'a cessé de croître.



Stackoverflow Trends

En février 2021, l'indexation par TIOBE (basée sur le nombre de recherches sur une multitude de moteurs de recherche) place Python en troisième position derrière C et Java. Même si Python fait partie des top 10 depuis 2004, en 2020, le langage raflait le «Language of the year award» qui est attribué au langage avec le plus de croissance, pour la quatrième fois. Une première.
En France, en 2018, Python a été choisi comme le langage officiel dans l'enseignement secondaire et supérieur.

PYTHON DANS LE MONDE PROFESSIONNEL

> ***Un langage de programmation est nécessaire pour l'écriture des programmes: un langage simple d'usage, interprété, concis, libre et gratuit, multiplateforme, largement répandu, riche de bibliothèques adaptées aux thématiques étudiées et bénéficiant d'une vaste communauté d'auteurs dans le monde éducatif est nécessaire. Au moment de la conception de ce programme, le langage choisi est Python version 3 (ou supérieure).***
>
> SCIENCES NUMÉRIQUES ET TECHNOLOGIE, EDUSCOL

ET À ATOL ?

Python a été introduit à Atol en 2016 avec l'utilisation du framework Django. Aujourd'hui, cinq projets ont été développés sous Python dont deux encore en développement actif:

- Application WebMDPH
- Limédia Sillon Lorrain

L'équipe Python est constituée de treize personnes dont huit développeurs et génère huit pourcent du chiffre d'affaires d'Atol.

Alors, Python à Atol CD ? Oui, une histoire récente mais qui semble être faite pour durer…

CHAPITRE 2

PARLONS PYTHON

PYTHON DANS LE MONDE PROFESSIONNEL

Mais pourquoi autant d'engouement pour Python ? À quoi est due cette popularité ? Pour répondre à cela, il nous faut d'abord faire un tour de Python et découvrir sa philosophie particulière.

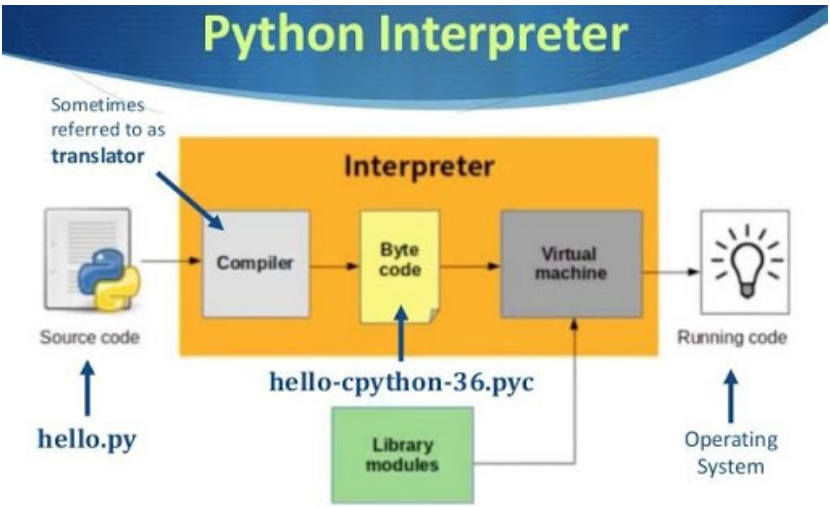
USAGE GÉNÉRAL (GENERAL PURPOSE)

Python est un langage à usage général (contrairement aux langages à usage spécifique, domain-specific languages) et peut donc être utilisé dans une grande variété de domaines :

- Applications Web
- Client Lourd (et oui, on peut faire des GUI sous Python)
- Analyse scientifique et économique (numpy, pandas)
- Administration Système (excellent langage de scripting)
- Machine Learning
- …

INTERPRÉTÉ (INTERPRETED)

Python est généralement défini comme langage interprété. Mais il est important de rappeler que «compilé» et «interprété» ne sont pas des propriétés du langage en lui-même, mais des propriétés d'implémentation. En réalité, le code source est compilé en Bytecode puis exécuté dans une machine virtuelle par l'interpréteur Python.



CSC121, Slideshare

Les langages interprétés souffrent souvent d'un problème de performance face aux langages compilés.

Alors, quels avantages pour le monde professionnel ?

- facilité d'utilisation et d'installation
- rapidité de développement (pas besoin de compiler le code avant de le tester)
- portabilité (il suffit de l'interprète)

La performance

Dans un interview avec Infoworld, Guido van Rossum s'exprime sur les problèmes de performance souvent reprochés à Python:

> ***At some point, you end up with one little piece of your system, as a whole, where you end up spending all your time. If you write that just as a sort of simple-minded Python loop, at some point you will see that that is the bottleneck in your system. It is usually much more effective to take that one piece and replace that one function or module with a little bit of code you wrote in C or C++ rather than rewriting your entire system in a faster language, because for most of what you're doing, the speed of the language is irrelevant.***
>
> GUIDO VAN ROSSUM, INFOWORLD

Comme d'habitude, le langage en lui-même n'est pas le seul responsable de la performance d'un programme. La structure et la qualité du code, ainsi que la maitrise de la technologie utilisée, sont des facteurs importants à prendre en considération.

En réalité, Python est bien assez rapide pour la majorité des applications.
Si les performances de Python s'avèrent insuffisantes pour un problème donné, il faudra:

- déterminer si le code problématique manque d'optimisation et le cas échéant, le modifier
- ou, comme le suggère Guido, réécrire le module problématique en C ou C++ plutôt que de réécrire le système complet dans un autre langage
- Suivre quelques recommandations et bonnes pratiques afin d'optimiser les performances

HAUT NIVEAU (HIGH-LEVEL)

Python est un langage haut niveau. C'est-à-dire que le niveau d'abstraction est élevé.

Concrètement, cela veut dire que le langage peut être plus facile à lire et à écrire et qu'il nécessite moins de connaissances détaillées du fonctionnement d'un ordinateur (memory allocation anyone ?). Théoriquement, cela permet d'écrire des programmes complexes plus rapidement tout en s'assurant que le niveau de lisibilité et de maintenabilité reste élevé.
Théoriquement.

Dans les faits, le code spaghetti, dispersé, illisible et incompréhensible existe tout autant sous Python que dans d'autres langages. Un langage haut niveau nécessite souvent plus de rigueur et un niveau d'exigence plus important (cf. Attention à la nomenclature).
Mais dans le monde professionnel, un langage haut niveau comme Python signifie que le niveau d'entrée sur les projets est faible car un développeur peut être opérationnel rapidement. Le recrutement et la formation sont donc plus faciles.

TYPAGE DYNAMIQUE, MAIS FORT (DYNAMIC BUT STRONG TYPES)

Python est un langage doté d'un typage dynamique.
Le type est vérifié à la volée lors de l'exécution du code.

Type check at runtime

```
def a_simple_trap(a):
    if a > 0:
        print(a + 42)
    else:
        print(a + "42")
```

Le else n'est pas type checked

```
>>> a_simple_trap(1)
43
```

Le else est type checked (et le typage fort se fait ressentir)

```
>>> a_simple_trap(-1)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
   print(a + "42")
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Il n'y a aucune nécessité de déclarer les types sous Python et une variable peut changer de type au cours de son existence.

Comparaison Python, Java

Python

```
>>> un = 1
>>> type(un)
<class 'int'>
>>> un = "un"
>>> type(un)
<class 'str'>
```

Java

```
public class HelloTypes {
   public static void main(String[] args)
{
        int un;
        un = 1;
        un = "un";
        System.out.println(un);
   }
}

HelloTypes.java:5: error: incompatible
types: String cannot be converted to int
        un = "un";
             ^
1 error
```

Est-ce que le typage dynamique apporte un avantage dans le milieu professionnel ?

Pas vraiment. De manière générale, le typage dynamique sera plus flexible et permettra de coder sans se préoccuper du type de la variable. Le code sera donc souvent plus succinct et plus lisible. Le fait que la variable puisse changer de type au cours de son existence permet d'éviter certaines gymnastiques de casting qu'on peut trouver dans du code à typage statique.

Mais le typage statique, lui, permet d'éviter certains bugs comme celui évoqué ci-dessus.

Duck Typing

If it looks like a duck, swims like a duck and quacks like a duck, then it probably is a duck.

Le concept de «duck typing» est relié à celui du typage dynamique. Un objet ne sera donc pas identifié par la déclaration de son type - comme sous Java: Private String Example; - mais par son comportement, c'est-à-dire tous ses attributs et ses méthodes. Ainsi, un objet peut être considéré comme de type "A" pour certaines opérations et de type "B"

pour d'autres.

Type Hinting

La version 3.5 de Python introduit les type hints.

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

Et certaines bibliothèques comme mypy tentent d'implémenter l'équivalent d'une vérification d'un typage statique.

Mais Python restera toujours un langage à typage dynamique.

> ***It should also be emphasized that Python will remain a dynamically typed language, and the authors have no desire to ever make type hints mandatory, even by convention.***
>
> PEP 484, NO GOALS

ORIENTÉ OBJET (OBJECT ORIENTED)

Python est un langage orienté objet et tous ses éléments sont des objets.

la fonction hello() est une instance de la classe 'function'

```
def hello():
 print("world")

print(type(hello))
>>> <class 'function'>
```

Mais ce qui nous intéressera particulièrement ici, c'est les différences dans l'approche de la programmation orientée objet entre un langage comme Python et un langage comme Java.

<table>
<tr><th>python</th><th>java</th></tr>
<tr><td colspan="2">Construction</td></tr>
<tr><td>La classe peut être définie n'importe où, dans n'importe quel fichier

```python
class Book:
    def __init__(self, author, title, publication_year):
        self.author = author
        self.title = title
        self.publication_year = publication_year
```

</td><td>Doit être défini dans Book.java

```java
public class Book {
 private String author;
 private String title;
 private int publicationYear;

 public Book(String author, String title, int publicationYear) {
    this.author = author;
    this.title = title;
    this.publicationYear = publicationYear;
 }

 public String getAuthor() {
   return author;
 }

 public String getTitle() {
   return title;
 }

 public int getPublicationYear() {
   return publicationYear;
 }

 public void setAuthor(author) {
    this.author = author;
 }

 public void setTitle(title) {
    this.title = title;
 }

 public void setPublicationYear(publicationYear) {
    this.publicationYear = publicationYear;
 }
}
```

</td></tr>
</table>

- la class `init` correspond au *constructor* de Java.
- Les variables de classe sont déclarées à la racine de la classe
- Les variables d'instance sont déclarées dans le constructeur
- Pas de notion de `Private` ou `Public` (*pas comme on l'entend sous Java*)
- Contrairement au `this` de Java qui peut être omis dans certains cas, le `self` (qui fera référence à l'instance) est obligatoire sous Python
- Aucun besoin de `Setter` et `Getter`

Instantiation

Instanciation sous Python

```python
book = Book("Alexis", "Python", 2021)
print("author", book.author)
print("title", book.title)
print("publication_year",
book.publication_year)
```

Instanciation sous Java

```java
public static void main(String[] args) {
   Book myBook = new Book("Alexis",
"Java", 2021);


System.out.println(myBook.getAuthor());
   System.out.println(myBook.getTitle());

System.out.println(myBook.getPublicationY
ear());
 }
```

<table>
<tr><th colspan="2">Piège</th></tr>
<tr><td>AttributeError

```
book.publisher = "Atol"
print("publisher", book.publisher)
```

Il est possible de créer des variables d'instance en dehors du constructeur sous Python, mais cela est considéré comme une mauvaise pratique car on peut introduire des bugs difficilement trouvables.

```
print("pages", book.pages)
>>> Traceback (most recent call last):
>>> AttributeError: module
'python_code.book' has no attribute
'pages'
```

</td><td></td></tr>
<tr><th colspan="2">Public & Private</th></tr>
<tr><td>

```
class Book:
    def __init__(self, author, title,
publication_year):
        self.author = author
        self.title = title
        self.publication_year =
publication_year
        # this is supposed to be private
        self._format = "adoc"
        # this is really private
        self.__language = "frenglish"
```

</td><td></td></tr>
<tr><td>La notion de <code>non-public</code>, représenté par le préfix <code>_</code>, n'est qu'une convention de nommage. Elle permet d'indiquer aux développeurs que la méthode/variable/etc n'est pas censée être utilisée directement.

En revanche, avec l'utilisation du préfix <code>__</code>, Python change le nom de la variable en interne pour la rendre plus difficilement accessible directement.</td><td></td></tr>
</table>

<table>
<tr><th colspan="2">Access Control</th></tr>
<tr><td>Sous Python, les <code>Getter</code> et <code>Setter</code> sont optionnels, mais on peut contrôler l'accès aux données avec les properties.</td><td></td></tr>
<tr><td>

```python
    @property
    def format(self):
        return self._format

    @format.setter
    def format(self, format):
        print(f"You are now reading your
book in the '{format}' format.")
        self._format = format

    @format.deleter
    def format(self):
        print("No more reading for you.")
        del self._format
```

</td><td>

```java
private String format;

public String getFormat() {
  return format;
}

public void setFormat(String format) {
  this.format = format;
}
```

</td></tr>
<tr><td>

```python
print(f"Book format is: {book.format}")
book.format = "print"
print(f"Book format is: {book.format}")
del book.format
>>> Book format is: adoc
>>> You are now reading your book in the
'print' format.
>>> Book format is: print
>>> No more reading for you.
```

</td><td></td></tr>
</table>

Inheritance and Polymorphism

ring.py

```python
class Ring:
    def __init__(self, name):
        self.name = name
        print("(Ring.__init__)",
self.name)


class Power:
    def cast_magic(self, ability):
        print(ability)


class RingOfPower(Ring, Power):
    def __init__(self, name, owner):
        super().__init__(name)
        self.owner = owner
        print("(RingOfPower.__init__)",
self.owner)


class OneRing(RingOfPower):
    def __init__(self):
        super().__init__("The One",
"Sauron")

    def cast_magic(self):
        print(f"{self.rule()} and
{self.bind()}")

    def rule(self):
        return "rule them all"

    def bind(self):
        return "in the darkness bind them"
```

Ring.java

```java
public class Ring {
 private String name;

 public Ring(String name) {
   this.name = name;
   System.out.println("(public.Ring) " +
this.name);
 }
}
```

Power.java

```java
interface Power {
 void castMagic(String ability);
}
```

RingOfPower.java

```java
public class RingOfPower extends Ring
implements Power {
  private String owner;

 public RingOfPower(String name, String
owner) {
   super(name);
   this.owner = owner;

System.out.println("(public.RingOfPower)
" + this.owner);
 }

 public void castMagic(String ability) {
   System.out.println(ability);
 }

}
```

OneRing.java

```java
class OneRing extends RingOfPower
implements Power {

 public OneRing() {
   super("The One", "Sauron");
 }

 public void castMagic() {
   System.out.println(this.rule() + " and
" + this.bind());
 }

 public String rule() {
   return "rule them all";
 }
```

<table>
<tr><td></td><td>

```java
public String bind() {
  return "in the darkness bind them";
 }
}
```

</td></tr>
<tr><td>

Instanciation Python

```python
narya = RingOfPower("Narya", "Gandalf")
narya.cast_magic("inspire")

one = OneRing()
one.cast_magic()
```

</td><td>

Main.java (Instanciation Java)

```java
class Main {
 public static void main(String[] args) {
   RingOfPower narya = new
RingOfPower("Narya", "Gandalf");
   narya.castMagic("inspire");

   OneRing one = new OneRing();
   one.castMagic();
 }
}
```

</td></tr>
<tr><td>

Result Python

```
>>> (Ring.__init__) Narya
>>> (RingOfPower.__init__) Gandalf
>>> inspire

>>> (Ring.__init__) The One
>>> (RingOfPower.__init__) Sauron
>>> rule them all and in the darkness
bind them
```

</td><td>

Result Java

```
>>> (public.Ring) Narya
>>> (public.RingOfPower) Gandalf
>>> inspire

>>> (public.Ring) The One
>>> (public.RingOfPower) Sauron
>>> rule them all and in the darkness
bind them
```

</td></tr>
<tr><td>Python supporte l'héritage multiple.</td><td>Java ne supporte pas l'héritage multiple mais peut implémenter plusieurs interfaces.

Depuis Java 8, il est également possible d'avoir des implémentations de méthodes par défaut définies directement dans les interfaces.</td></tr>
</table>

PHILOSOPHIE

S'il y a quelque chose qu'on commence à discerner, c'est que Python n'est pas qu'un langage de programmation, c'est aussi une philosophie (avec un humour qui lui est propre).

PYTHON DANS LE MONDE PROFESSIONNEL

> ***In many ways, the design philosophy I used when creating Python is probably one of the main reasons for its ultimate success.***
>
> PYTHON-HISTORY, GUIDO VAN ROSSUM

Placer le développeur au centre de l'expérience de programmation, rendre le code plus lisible et offrir au développeur assez de liberté pour qu'il puisse exprimer sa créativité sans compromettre l'intégrité du code… Cette philosophie n'est pas là pour s'imposer à vous et lancer des débats "pour ou contre". Mais elle guide l'évolution de Python et son utilisation.

Dans le monde professionnel, on écrit avant tout pour le prochain développeur. La fonctionnalité n'est plus la seule préoccupation. D'autres éléments vont entrer en compte dans l'appréciation du code.

> ***"You primarily write your code to communicate with other coders, and, to a lesser extent, to impose your will on the computer."***
>
> DROPBOX INTERVIEW, GUIDO VAN ROSSUM

Considérations Professionnelles

Productivité

La rapidité de production est un enjeu majeur dans le monde professionnel. Python

permet de monter en compétence et de produire des fonctionnalités rapidement, même pour un novice.

> ***While this is hard to assess objectively, Python is considered a winner in coding time by most who have tried it.***
>
> FOREWORD FOR «PROGRAMMING PYTHON», GUIDO VAN ROSSUM

Maintenabilité

Il ne s'agit pas seulement de produire de la fonctionnalité. Le code fourni doit aussi être facilement maintenable, c'est-à-dire:

- être lisible et compréhensible pour le prochain développeur
- permettre un débogage efficace et précis (remontée d'erreurs, logging, …)
- (idéalement) être bien documenté

> ***This emphasis on readability is no accident. (…) Many of Python's features, in addition to its use of indentation, conspire to make Python code highly readable. (…) Readability is often enhanced by reducing unnecessary variability. When possible, there's a single, obvious way to code a particular construct. This reduces the number of choices facing the programmer who is writing the code, and increases the chance that will appear familiar to a second programmer reading it.***
>
> FOREWORD FOR «PROGRAMMING PYTHON», GUIDO VAN ROSSUM

Évolutivité

La limite entre maintenabilité et évolutivité n'est pas toujours évidente. Si la maintenabilité est souvent liée au code en lui-même, l'évolutivité, elle, est plutôt liée à la structure. Bien qu'il soit difficile de définir une structure qui garantit l'évolutivité du code, on sait souvent ce qu'il faut éviter: code spaghetti, duplication de code, héritage multiple mal maitrisé, import circulaire, globales qui ne sont pas thread-safe, …

Python n'offre pas plus de garanties qu'un autre langage. Ce sera au développeur de maitriser sa structure.

PEP (Python Enhancement Proposals)

Le développement de Python s'effectue surtout par le biais des «Python Enhancement Proposals» PEP, dont la plus connue est certainement PEP 8 — Style Guide for Python Code.

> ***One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code.***
>
> PEP 8

De manière générale, les PEP permettent de suivre de très près le langage ainsi que les recommandations accompagnant les nouvelles fonctionnalités. À un niveau plus avancé, elles permettent de participer à l'évolution de Python.

Dans le monde professionnel, les PEP permettent d'anticiper les nouvelles fonctionnalités de Python et leur incorporation dans le code (ou pas) et un guide tel que PEP 8 peut faciliter les décisions et réduire les débats chronophages et peu productifs.

Mais suivre les recommandations à la lettre serait ne pas comprendre la philosophie au cœur de Python.

> ***However, know when to be inconsistent — sometimes style guide recommendations just aren't applicable. When in doubt, use your best judgment. (…) Some other good reasons to ignore a particular guideline: (…) When applying the guideline would make the code less readable, even for someone who is used to reading code that follows this PEP.***
>
> FOOLISH CONSISTENCY, PEP 8

Zen of Python

Publié en 1999 par Tim Peters, nommé «[Python's] most prolific and tenacious core developer» par Guido lui-même, le «zen of python» représente un ensemble de 20 principes fondamentaux (même si le 20e n'a jamais été publié). Étudier chaque élément du zen de Python dépasse le cadre de ce document, mais la connaissance de ses éléments est une partie importante du développeur Python.

> ***The code is more what you'd call 'guidelines' than actual rules.***
>
> HECTOR BARBOSSA, PIRATE

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

La responsabilisation du développeur

> ***With great power comes great responsibility..***
>
> UNCLE BEN (OR NOT)

Principalement utilisé pour expliquer l'absence de notion Public et Private dans la programmation orientée objet de Python, l'adage «We are all consenting adults» ou «We are all responsible users» revient relativement souvent dans les discussions concernant Python.

Python préfèrera toujours laisser la liberté au développeur en le guidant avec des conventions qu'il pourra suivre (ou pas) plutôt que de l'enfermer dans des règles rigides et strictes.

Selon le profil du développeur (ou de l'équipe de développement), ceci est un bonheur ou une lourde charge. En effet, cette liberté peut être déconcertante pour un développeur qui a besoin d'un cadre un peu plus rigide pour évoluer sereinement dans son code.

Alors qu'en est-il dans le monde professionnel ? Si cette liberté n'est pas maitrisée et canalisée, on peut vite rentrer dans des problématiques de scalabilité avec un code qui va devenir de plus en plus compliqué (*«Simple is better than complex.»*, *«Complex is better than complicated.»*) et de moins en moins lisible (*«Readability counts.»*), difficilement maintenable et peu évolutif.

Comme avec n'importe quel autre langage, Python ne fait pas de miracle. La compétence et l'expérience comptent.

Il vaut mieux demander pardon que la permission

Sous Python, on préfèrera l'approche Easier to Ask Forgiveness than Permission (EAFP) plutôt que Leap Before You Leap (LBYL). C'est-à-dire, plutôt que de vérifier si mon objet peut faire quelque chose avant de le faire, je vais gérer le problème s'il ne sait pas faire quelque chose après avoir essayé.

Cette méthode est bien adaptée au duck typing qui ne se soucie pas du type de l'objet mais de son comportement.

```
class NotADuck(Exception):
    # custom exception
    def __init__(self, message="This is not a duck"):
        self.message = message
        super().__init__(self.message)


class Person:
    def not_a_duck(self):
        print("I do not quack and swim")


class Duck:
    def quack(self):
        print("I quack like a duck.")

    def swim(self):
        print("I swim like a duck.")
```

```
def lbyl(duck):
    # considered non-pythonic
    print("\nLBYL")
    if hasattr(duck, "quack") and callable(duck.quack):
        duck.quack()
    if hasattr(duck, "swim") and callable(duck.swim):
        duck.swim()


def eafp(duck):
    # considered pythonic
    print("\nEAFP")
    try:
        duck.quack()
        duck.swim()
        # duck.bark()  # would raise exception as well
    except AttributeError:
        raise NotADuck()


d = Duck()
lbyl(d)
eafp(d)

p = Person()
lbyl(p)
eafp(p)
LBYL
I quack like a duck.
I swim like a duck.

EAFP
I quack like a duck.
I swim like a duck.

LBYL

EAFP
Traceback (most recent call last):
 File "eafp.py", line 35, in eafp
   duck.quack()
AttributeError: 'Person' object has no attribute 'quack'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "eafp.py", line 48, in <module>
   eafp(p)
 File "eafp.py", line 39, in eafp
   raise NotADuck()
eafp.NotADuck: This is not a duck
```

Quels sont les avantages de l'approche EAFP ? Généralement, on mettra en avant les points suivants:

Plus lisible:

- le chemin «qui doit fonctionner» est plus clair
- parfois moins de vérification à faire (on ne peut pas toujours anticiper tous les cas avec l'approche LBYL)

Plus performant:

- Contrairement à l'approche LBYL où je vais faire appel deux fois à mon traitement (une fois pour le Look et une fois pour le Leap), EAFP tente le traitement directement
- Il est vrai que les exceptions sont plus coûteuses que les if mais comme «l'exception» n'est appelée qu'en cas d'erreur, ce coût est moins fréquent.

Il est vrai que les exceptions sont plus coûteuses que les if mais l'exception n'est appelée qu'en cas d'erreur.

En réalité, un code rempli de try\except peut être difficilement lisible et la gestion de remontée d'erreur pour être gérée par un traitement parent peut être difficilement maintenable.

On recommande généralement l'approche EAFP pour Python, mais on est aussi en droit de se demander s'il ne s'agit pas simplement d'un style, d'une façon de coder dont l'utilisation (ou pas) est complètement à l'appréciation de l'équipe de développement.

Dans le monde professionnel, la cohérence et la persistance font foi.

Fait le café (batteries included)

Python inclut dans sa librairie standard une multitude de paquets ayant pour but d'aider le développeur à solutionner des problèmes divers sans pour autant faire appel à des

librairies externes.

Dans le monde professionnel, ceci peut être un aspect attrayant car il n'y a pas forcément besoin de se soucier de la maintenance et de l' intercompatibilité des versions des différents paquets.

Dans la réalité, la plupart des projets professionnels vont inclure des librairies externes. Mais de manière générale, on cherchera toujours à résoudre un problème avec la librairie standard avant de se tourner vers des modules externes.

> ***The Python source distribution has long maintained the philosophy of «batteries included» — having a rich and versatile standard library which is immediately available, without making the user download separate packages.***
>
> PEP 206 — PYTHON ADVANCED LIBRARY

Recommandation de librairie

Logging (SL)

Le module logging est inclus dans la librairie standard. Activement utilisé sur nos projets, il permet de facilement gérer les logs d'une application:

- Log Rotate
- Log Level
- Log to (multiple) files
- Format message
- ...

```
import logging
logging.warning('Watch out!')  # will print a message to the console
logging.info('I told you so')  # will not print anything
```

Black (Ext)

Un même code peut généralement être écrit de plusieurs manières. Pour réduire le temps attribué au débat inutile sur «la meilleure syntaxe», nous utilisons un formateur automatique de code. Plusieurs possibilités existent pour Python. Nous avons choisi Black.

Sans BLACK

```
migrations.CreateModel(
            name='SentElement',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True,
serialize=False, verbose_name='ID')),
                ('exchange_generated', models.BooleanField(null=True,
verbose_name='Échange généré')),
                ('decision', models.ForeignKey(null=True,
on_delete=django.db.models.deletion.CASCADE, to='decisions.Decision')),
                ('exchange_history',
models.ForeignKey(on_delete=django.db.models.deletion.CASCADE,
to='exchanges.ExchangesHistorization', verbose_name="Historique de l'échange")),
            ],
        ),
```

Avec BLACK

```
migrations.CreateModel(
            name="SentElement",
            fields=[
                (
                    "id",
                    models.AutoField(
                        auto_created=True,
                        primary_key=True,
                        serialize=False,
                        verbose_name="ID",
                    ),
                ),
                (
                    "exchange_generated",
                    models.BooleanField(null=True, verbose_name="Échange généré"),
                ),
                (
                    "decision",
                    models.ForeignKey(
                        null=True,
                        on_delete=django.db.models.deletion.CASCADE,
                        to="decisions.Decision",
                    ),
```

```
            ),
            (
                "exchange_history",
                models.ForeignKey(
                    on_delete=django.db.models.deletion.CASCADE,
                    to="exchanges.ExchangesHistorization",
                    verbose_name="Historique de l'échange",
                ),
            ),
        ],
    ),
```

Pipenv (Ext)

Toujours (oui, toujours) utiliser un environnement virtuel pour le développement Python.

Le stack traditionnel pour gérer l'environnement Python est:

- pip : le gestionnaire de paquets standard
- virtuelenv : le gestionnaire d'environnements virtuels standard
- requirements.txt : le gestionnaire de dépendances standard

Mais je ne saurais assez recommander Pipenv: un (le) gestionnaire de paquet et d'environnement Python.

La gestion de multiples versions de Python sur un même système et la compatibilité des librairies sur plusieurs programmes (et version Python du coup) peut être un casse-tête et une perte de temps inutile. Pipenv permet de solutionner ce problème de manière intuitive et cohérente sans pour autant perdre la compatibilité avec les standards historiques en permettant d'importer et d'exporter requirements.txt.

Requests (Ext)

Nos logiciels ont parfois besoin de communiquer avec d'autres API. Requests est la librairie HTTP pour toutes nos requêtes HTTP.

Extrait d'un code en production avec des commentaires explicatifs

```python
import logging

from django.conf import settings

import requests
from cnsa_service.mixins.service_disruption_mixins import ServiceDisruptionMixin
from requests.adapters import HTTPAdapter
from requests.packages.urllib3.util.retry import Retry


# SPECIFIC LOGGER IS DEFINED
logger = logging.getLogger("ras")


# CLASS INHERITS FROM MIXIN
class CnsaService(ServiceDisruptionMixin):
    def __init__(self):
        # ENVIRONMENT VARIABLES
        self.BASE_URL = settings.CNSA_REMOTE_ACCESS_SERVICE_BASE_URL
        self.HEADERS = {"X-Gravitee-Api-Key": settings.CNSA_REMOTE_ACCESS_SERVICE_KEY}

        # THE REQUEST SESSION IS INSTANTIATED
        self.session = requests.Session()

        # RETRY POLICY
        retries = Retry(
            total=1,
            backoff_factor=1,
            status_forcelist=[
                429,
                500,
                503,
                504,
            ],
        )
        self.session.mount("http://", HTTPAdapter(max_retries=retries))
        self.session.mount("https://", HTTPAdapter(max_retries=retries))

        # SET THE SESSION HEADERS
        self.session.headers.update(self.HEADERS)

    # 'PRIVATE' METHOD calls external API
    # TYPEHINTS are defined and DOCSTRING generated
    def _call_flow_1_5(self, ras_folder_id: str) -> int:
        """Flow 1.5
        Acquit a folder

        {Verb: Expected Status}: {"PATCH": 200 OK}

        Args:
            ras_folder_id (str): the id of the folder to be acquitted
```

```
        Returns:
            int: HTTP status code
        """
        url = f"{self.BASE_URL}/QuestionnaireResponse/{ras_folder_id}"

        # LOG THE CALL
        logger.info(f"call: _call_flow_1_5 -> {url}")

        # SOME SPECIFIC json data TO BE SEND TO THE API
        data = [{"op": "add", "path": "/tag", "value": "read"}]

        # SOME SPECIFIC headers FOR THIS CALL (default headers are send as well)
        headers = {"Content-type": "application/json-patch+json"}

        # API PATCH CALL
        response = self.session.patch(url, headers=headers, json=data)

        # INTERNAL response check POLICY
        response = self._check_response(response, 200)

        return response.status_code
```

Pytest (Ext)

Que votre logiciel inclue ou pas des tests unitaires, que vous suiviez la méthodologie du Test Driven Development ou pas, s'il y a une librairie à recommander pour exécuter les tests sous Python c'est sans aucun doute Pytest.

Pathlib (SL)

Inclus dans la librairie standard depuis Python 3.6, Pathlib est un excellent module de gestion des File Paths. Sa particularité ? Les chemins sont des objets et une gestion sans maux de tête des chemins unix vs windows.

```
>>> path = Path('/home/alr/Notes/Python/presentation.adoc')
>>> path
PosixPath('/home/alr/Notes/Python/presentation.adoc')
>>> path.name
'presentation.adoc'
>>> path.stem
'presentation'
>>> path.suffix
'.adoc'
>>> path.parent
PosixPath('/home/alr/Notes/Python')
>>> path.parent.parent
PosixPath('/home/alr/Notes')
>>> path.anchor
'/'
```

Celery

Pour la gestion des files d'attente en asynchrone, le standard de l'industrie est Celery. Les retours de la communauté sur Celery sont excellents, même si on lui reproche parfois d'être un peu lourd à la mise en place.

Celery est activement utilisé sur des projets à Atol.

Web Frameworks

- Django : «LE» Framework, utilisé chez Atol pour tous les projets WEB Python
- Flask : Le micro-framework Python, sa scalabilité est légendaire
- FastAPI : Le petit dernier de la bande, Async, Type Hints, … à surveiller de très près

CHAPITRE 3

CE QU'IL NE FAUT PAS FAIRE

ATTENTION À LA NOMENCLATURE

Avec Python, on peut parfois avoir l'impression que tout est possible. Mais parfois, le côté permissif et ce «trop de liberté» peut avoir des conséquences négatives sur la lisibilité du code. Dans le feu de l'action et sous le stress des contraintes de production, un code sur lequel plusieurs personnes travaillent à des périodes différentes et espacées peut vite perdre de son sens.

Une nomenclature hasardeuse utilisant le même nom de variable pour des objets différents dans la même méthode peut rendre le debuggage compliqué.

```python
class SendInEvaluationAssignCreateView(generics.CreateAPIView):
    permission_classes = (permissions.IsAdminUser,)
    serializer_class = SendInEvaluationRequestSerializer

    def post(self, request, *args, **kwargs):
        # assign Team to Requests
        requests_ids = self.request.data["requests"]
        folders_id = self.request.data["folders"]
        requests = Request.objects.filter(id__in=requests_ids)
        folders = Folder.objects.filter(id__in=folders_id)
        team_id = self.request.data["team"]
        try:
            requests.update(team_id=team_id)
        except utils.IntegrityError:
            return response.Response(
                {"message": "Equipe non trouvée."}, status=status.HTTP_404_NOT_FOUND
            )

        # set 'Being evaluated' to Requests
        for request in requests:
            request.set_being_evaluated_status()

        for folder in folders:
            folder.set_being_evaluated_status()
            # sending update_team_event to be logged
            update_team_event.send(
                sender=self.__class__,
                folder=folder,
                label_prefix=False,
                team_id=team_id,
            )

        # get Requests formatted for serializer
        requests_data = requests.values(
            "id",
            "folder_id",
            "demand__name",
            "nature__name",
            "status__name",
            "team__name",
        )

        # return serialized data
        return response.Response(
            SendInEvaluationRequestSerializer(
                [request for request in requests_data], many=True
            ).data,
            status=status.HTTP_200_OK,
        )
```

Il ne manque plus qu'un appel à une API avec le module requests…

LES ORM

Considéré par certains comme le vietnam de l'industrie informatique mais adoré par d'autres, le mapping objet-relationnel tente de répondre à un problème pour lequel il ne semble pas avoir de bonne solution: s'interfacer entre le logiciel et sa base de données relationnelle pour simuler une base orientée objet. La facilité et la rapidité d'implémentation d'un ORM peuvent être séduisantes pour démarrer, mais l'utilisation d'un ORM est souvent coûteuse. Nous ne voulons pas entrer dans le débat qui divise la communauté depuis des années, mais nous pouvons témoigner de quelques problématiques liées à l'utilisation d'un ORM:

- Problème de performance
- Requête illisible dès qu'elle dépasse un certain niveau de complexité
- Une scalabilité désastreuse (plus l'application grandie, plus l'ORM devient un problème)
- Erreur silencieuse (spécifique Django-ORM)
- …

En réalité, un ORM peut sensiblement augmenter la rapidité de développement mais si son utilisation n'est pas cadrée par des règles précises, adaptées au développement dans le monde professionnel, celui-ci pourra s'avérer contre-productif. Comme avec tout outil puissant dont l'utilisation est trompeusement facile, une bonne connaissance des limites est fortement recommandée.

ARGUMENT PAR DÉFAUT MUTABLE

Un des exemples les plus répandus des pièges à éviter:

```
def append_to(element, to=[]):
    to.append(element)
    return to

my_list = append_to(12)
print(my_list)

my_other_list = append_to(42)
print(my_other_list)
```

Ici, on s'attend à un retour de [12] et [42] mais les arguments par défaut sont évalués qu'une seule fois à la définition de la fonction, si bien que le code ci-dessus retourne:

```
[12]
[12, 42]
```

car une fois muté, la liste to reste dans le nouvel état.

VARIABLES GLOBALES

L'utilisation de variables globales est fortement déconseillée. C'est tout à fait possible d'en déclarer, mais celles-ci peuvent avoir des effets de bord difficile à anticiper. Sous Django par exemple, les variables globales ne sont pas thread safe et leurs valeurs ne sont pas robustes.

WALRUS OPERATOR

Introduit dans Python 3.8 (et la raison du départ de Guido van Rossum), le scope du walrus operator peut être problématique.

```
>>> def is_false():
...     return False
...
>>> if assigned := is_false() is False:
...     # `assigned` has the result of `is_false() is False` -> True
...     print(assigned)
True
>>> if (assigned := is_false()) is False:
...     # `assigned` has the result of `is_false()` -> False
...     print(assigned)
False
```

INDENTATION

L'indentation compte sous Python. Et parfois, même si le code est syntaxiquement correct, il ne l'est pas sémantiquement.

```python
def an_indentation_trap(traps):
    for trap in traps:
        if trap_has_snapped(trap):
            return True
        return False  # will return False at the first trap that hasn't snapped

def an_indentation_trap(traps):
    for trap in traps:
        if trap_has_snapped(trap):
            return True
    return False  # will return False only if no trap has snapped
```

ET COMMENT JE DEBUG ?

Python génère des traceback lors d'une erreur dans le code.

```
ERROR 2021-03-25 18:45:54,286 log 205 139627925956352 Internal Server Error: /api/exchanges/sngi/1/validate/
Traceback (most recent call last):
  File "/usr/local/lib/python3.8/site-packages/django/core/handlers/exception.py", line 34, in inner
    response = get_response(request)
  File "/usr/local/lib/python3.8/site-packages/django/core/handlers/base.py", line 115, in _get_response
    response = self.process_exception_by_middleware(e, request)
  File "/usr/local/lib/python3.8/site-packages/django/core/handlers/base.py", line 113, in _get_response
    response = wrapped_callback(request, *callback_args, **callback_kwargs)
  File "/usr/local/lib/python3.8/site-packages/django/views/decorators/csrf.py", line 54, in wrapped_view
    return view_func(*args, **kwargs)
  File "/usr/local/lib/python3.8/site-packages/django/views/generic/base.py", line 71, in view
    return self.dispatch(request, *args, **kwargs)
  File "/usr/local/lib/python3.8/site-packages/rest_framework/views.py", line 505, in dispatch
    response = self.handle_exception(exc)
  File "/usr/local/lib/python3.8/site-packages/rest_framework/views.py", line 465, in handle_exception
    self.raise_uncaught_exception(exc)
  File "/usr/local/lib/python3.8/site-packages/rest_framework/views.py", line 476, in raise_uncaught_exception
    raise exc
  File "/usr/local/lib/python3.8/site-packages/rest_framework/views.py", line 502, in dispatch
    response = handler(request, *args, **kwargs)
  File "/app/exchanges/views.py", line 445, in get
    response = TokenSuite().sngi_validation(person_id)
  File "/app/exchanges/token_suite.py", line 29, in __init__
    self.this_is_a_bug = this_is_a_bug()
NameError: name 'this_is_a_bug' is not defined
[25/Mar/2021 18:45:54] "GET /api/exchanges/sngi/1/validate/ HTTP/1.1" 500 17247
```

Une analyse directe du traceback peut suffir pour déterminer la source du problème et nombreux sont les développeurs qui se contentent de simples print() stratégiquement placés pour debugger leur code.

Bien sûr, il existe des outils - inclus dans la librairie standard - pour faciliter le processus. The Python Debugger (pdb) permet de placer des breakpoints à l'intérieur du code et d'ouvrir une console interactive dans le traceback du bug afin d'analyser le contenu de l'environnement en erreur et même de les modifier à la volée pour tenter de corriger le problème en direct.

```
>>> import pdb
>>>
>>> a = None
>>> for i in range(10):
...     if i == 4:
...         a = "Hello"
...     elif i == 5:
...         pdb.set_trace()
...
> <stdin>(1)<module>()
(Pdb) print(a)
Hello
(Pdb) print(i)
5
(Pdb)
```

Pour simplifier le processus, depuis Python 3.7 PEP-553 on peut utiliser la commande breakpoint() qui se charge de monter pdb et de l'appeler.

```
>>> a = None
>>> for i in range(10):
...     if i == 4:
...         a = "Hello"
...     elif i == 5:
...         breakpoint()
```

CHAPITRE 4

TÉMOIGNAGES

Ces témoignages ont été recueillis au sein de l'équipe Python d'Atol Conseils & Développements. Les personnes concernées n'étaient pas au courant de la finalité du sondage afin de ne pas les influencer.

PYTHON DANS LE MONDE PROFESSIONNEL

Ce que j'aime, c'est la lisibilité du code et l'héritage multiple: ça se fait tout seul, c'est naturel.

JULIEN 3 ANNÉES D'EXP. PYTHON

J'ai du mal avec le fait qu'il y ait plusieurs classes dans un même modèle. Et il y a un peu trop de liberté sous Python. Par contre, j'apprécie beaucoup la lisibilité: j'ai l'impression d'écrire du Python comme je parle. Et Python, c'est facile à apprendre: en 1 mois, tu peux commencer à bosser.

MAXIME 5 MOIS D'EXP. PYTHON

Ce que j'aime, c'est la structure du langage et la cohérence du code: peu de dette technique ressentie. C'est bien pensé et l'écosystème de Python est très riche. Les PEP m'aident à bien coder, c'est facile de trouver des conseils et de se renseigner sur les bonnes pratiques. La communauté et «l'ambiance» Python sont aussi importantes pour moi.

ROMAIN 2.5 ANNÉES D'EXP. PYTHON

C'est ludique et facile: je comprends ce que je fais.
Je trouve de l'aide sur tout et les exemples sont clairs.

FLAVIEN 4.5 ANNÉES D'EXP. PYTHON

La rigueur de la syntaxe Python est pour moi autant un point faible qu'un point fort: j'aime pouvoir structurer mon code comme je l'entends. Au début, ça me gênait, mais depuis qu'on a implémenté black, c'est bon, je ne m'en soucie plus. Sinon, ce que j'apprécie, c'est la rapidité de montée en compétence et la lisibilité du code: on comprend très vite du code Python.

VALENTIN 6 MOIS D'EXP. PYTHON

Ce que j'aime, c'est la lisibilité: la syntaxe est propre et exige une certaine rigueur. C'est un excellent langage pour apprendre les bases de la programmation. J'apprécie beaucoup sa versatilité et sa portabilité, en bref: si on me demande de faire quelque chose, je sais que je peux le faire sous Python.

NATHAN 1 ANNÉE D'EXP. PYTHON

CONCLUSION

PYTHON DANS LE MONDE PROFESSIONNEL

> ***(…)many consider using Python a pleasure — a better recommendation is hard to imagine.***
>
> FOREWORD FOR «PROGRAMMING PYTHON», GUIDO VAN ROSSUM

Une chose importante n'est jamais assez abordée quand on parle de Python: **pas de «;» en fin de ligne !** Ça change tout.

Les problèmes que nous sommes amenés à résoudre en tant que développeur peuvent être complexes et quand je code sous Python, j'ai l'impression que le langage est à mon service. Ce n'est pas une barrière supplémentaire que je dois surmonter pour transmuter ma pensée en code, mais un outil à ma disposition qui fait son possible pour ne pas se mettre en travers de mon chemin.

Est-ce que Python est le langage parfait ? Non. Et je ne fais certainement pas partie des développeurs qui ne sauront recommander d'autres langages (parfois plus appropriés). Est-ce que c'est un plaisir de travailler sous Python ? Oui, certainement.

Python a entièrement sa place dans le milieu professionnel, cela ne fait aucun doute, et une entreprise peut largement bénéficier d'avoir des développeurs Python qui apporteront bien plus que des pures compétences techniques.

Et en tout cas, aujourd'hui, si je peux affirmer que j'aime mon métier, c'est aussi grâce à Python.


VISITEZ NOTRE SITE