Un **itérateur** est un élément (valeur) qui produit une séquence de valeurs, en général dans une boucle de répétition (**d'itération**). La librairie standard de **Rust** définit des **itérateurs** pour balayer des **vecteurs**, des **chaînes**, des **dictionnaires** (table de hachage) et autres collections, mais également pour générer des lignes de texte depuis un flux d'entrée, une connexion avec un serveur réseau, à partir des données reçues d'autres **exétrons** (**threads** concurrents) en passant par un canal, etc.

> *La boucle **for** de **Rust** propose une syntaxe tout à fait naturelle pour les **itérateurs**. Enfin, les **itérateurs** sont accompagnés de tout un groupe de méthodes pour établir des correspondances, pour filtrer, pour fusionner, pour collecter, etc.*

Premières expériences

Les **itérateurs** de **Rust** sont souples, expressifs et efficaces. Commençons par une fonction qui renvoie la somme des **n premiers entiers positifs**, ce qui correspond au calcul du nombre **triangulaire**.

Fonction triangle

```
fn triangle(n: u32) -> u32 {
  let mut somme=0;
  for i in 1..n+1 { somme += i; }
  somme
}

fn main() {
  println!("{}", triangle(5));
}
```

Résultat

```
15
```

> *Dans cet exemple, l'expression **1..n+1** est une valeur de type **Range<u32>** qui incarne un **itérateur** permettant de générer les valeurs entières depuis la borne inférieurs incluse jusqu'à la borne extérieure, mais cette borne est exclue. La valeur peut donc servir d'opérande d'une boucle **for** pour faire le total de **1** à **n**.*
>
> *Mais un **itérateur** dispose également d'une méthode nommée **fold()** qui permet de reformuler le même problème ainsi :*

Méthode fold()

```
fn triangle(n: u32) -> u32 {
  (1..n+1).fold(0, |somme, element| somme+element)
}

fn main() {
  println!("{}", triangle(5));
}
```

Résultat

```
15
```

> *En commençant à **0**, **fold()** utilise chacune des valeurs générées par la plage **1..n+1** en lui appliquant la clôture **|somme, element| somme+element** pour additionner le total courant à la nouvelle valeur. La valeur renvoyée par la **clôture** devient le nouveau total courant et la dernière valeur renvoyée est celle qu'incarne **fold()**, c'est-à-dire le total général.*
>
> *Cette approche est assez différente de celle utilisant une boucle **for** ou **while**. Une fois que vous y êtes habitué, **fold()** constitue une alternative compacte et très lisible. Voici un autre exemple avec la fonction **factoriel()** :*

Méthode fold()

```
fn triangle(n: u32) -> u32 {
  (1..n+1).fold(0, |somme, element| somme+element)
}

fn factoriel(n: u64) -> u64 {
  (1..n+1).fold(1, |produit, element| produit*element)
}

fn main() {
  println!("{}", triangle(5));
  println!("{}", factoriel(5));
}
```

Résultat

```
15
120
```

> ***Dans une construction pour déploiement du précédent exemple, Rust disposant de tous les détails de fold(), il peut l'insérer en ligne dans triangle() pour éviter un appel. Il insère ensuite en ligne la clôture. Enfin, Rust examine le résultat et détecte qu'il possède une solution plus simple pour totaliser cette plage de nombres : la somme est toujours égale à n*(n+1) / 2. De ce fait, Rust convertit tout le corps de triangle(), avec la boucle, la clôture, et tout le reste en une simple instruction de multiplication avec quelques retouches arithmétiques.***

Les traits Iterator et IntoIterator

Une valeur qui implémente le trait **std::iter::Iterator** devient un itérateur. Le type de la valeur que produit l'itérateur est **Item**. La méthode **next()** renvoie **Some(v)**, **v** étant la prochaine valeur de l'itérateur, soit **None** pour signifier la fin de la séquence. Nous avons omis les nombreuses méthodes par défaut de **Iterator** car nous les verrons individuellement dans la suite de cette étude.

Le trait Iterator

```
trait Iterator {
  type Item;

  fn next(&mut self) -> Option<Self::Item>;
  … // Autres méthodes prédéfinies (par défaut)
}
```

*Dès qu'il existe une solution pour balayer naturellement les valeurs d'un type, il peut implémenter **std::iter::IntoIterator**. Sa méthode **into_iter()** reçoit une valeur et renvoie un itérateur approprié :*

Le trait IntoIterator

```
trait IntoIterator where Self::IntoIter::Item == Self::Item {
  type Item;
  type IntoIter: Iterator;

  fn into_iter(self) -> Self::IntoIter;
}
```

*Le type de la valeur de l'itérateur est **IntoIter** et **Item** est le type de la valeur produite. Un type qui implémente **IntoIterator** est appelé itérable parce qu'il s'agit de quelque chose que vous pouvez balayer par itération.*

*Tous ces composants sont réunis élégamment dans une boucle **for** de **Rust**. Voici par exemple comment réaliser une itération parmi l'ensemble des éléments d'un vecteur.*

Parcourir un vecteur

```
fn main() {
  let metaux = vec!["Antimoine", "Arsenic", "Aluminium", "Sélénium"];

  for metal in &metaux {
    println!("{}", metal);
  }
}
```

Résultat

```
Antimoine
Arsenic
Aluminium
Sélénium
```

*En coulisses, la boucle **for** est en fait une abréviation d'une série d'appels aux méthodes de **IntoIterator** et de **Iterator** :*

Parcourir un vecteur

```
fn main() {
  let metaux = vec!["Antimoine", "Arsenic", "Aluminium", "Sélénium"];

  let mut iterateur = (&metaux).into_iter();
  while let Some(metal) = iterateur.next() {
    println!("{}", metal);
  }
}
```

Résultat

```
Antimoine
Arsenic
Aluminium
Sélénium
```

*La boucle **for** se sert de **IntoIterator::into_iter()** pour convertir l'opérande **&metaux** en itérateur, avant d'appeler à répétition **Iterator::next()**. Lorsque la valeur renvoyée est **Some(metal)**, la boucle exécute le corps ; si la valeur renvoyée est **None**, c'est la fin de la boucle.*

*La boucle **for** appelle toujours **into_iter()** sur son opérande, mais vous pouvez aussi transmettre directement un itérateur à une boucle **for**, ce qui est par exemple le cas lorsque vous rebouclez dans une plage **Range**. Tous les itérateurs implémentent d'office **IntoIterator**, et donc avec la méthode **into_iter()** qui renvoie l'itérateur.*

Voici quelques termes essentiels au sujet des itérateurs :

- *Comme déjà dit, un **itérateur** désigne un type qui implémente le trait **Iterator**.*

- *Un **itérable** est un type qui implémente **IntoIterator** et vous pouvez obtenir un **itérateur** correspondant en appelant sa méthode **into_iter()**. Dans cet exemple, c'est la référence du vecteur **&metaux** qui est **l'itérable**.*
- *Un **itérateur** produit un ensemble de valeurs.*
- *Les valeurs produites sont des **éléments**. Dans l'exemple, c'est **« Antimoine »** ou **« Arsenic »**, etc.*
- *Le code qui reçoit les éléments produits est le **consommateur**. Dans l'exemple, c'est la boucle **for** qui consomme les **éléments** de **l'itérateur**.*

Création d'un itérateur

La plupart des types de collections disposent systématiquement des méthodes **iter()** et **iter_mut()** qui renvoient les itérateurs naturels pour le type concerné en produisant une **référence partagée** ou **mutable** sur chaque élément. C'est également le cas des tranches telles que **&[T]** et **&str**.

*Les méthodes **iter()** et **iter_mut()** sont les plus utilisées pour obtenir un itérateur, lorsque vous ne voulez pas utiliser une boucle **for** pour effectuer le travail :*

Méthodes iter() et next()

```
fn main() {
  let entiers = vec![4, 20, 12, 8, 6];
  let mut iterateur = entiers.iter();

  while let Some(entier) = iterateur.next() {
    print!(".{}.", entier);
  }
}
```

Résultat

```
.4..20..12..8..6.
```

*Le type de l'élément de cet itérateur est **&i32** et chaque appel à **next()** produit une référence au prochain élément, jusqu'à atteindre la fin du vecteur.*

*Chaque type peut implémenter ces deux méthodes en fonction de ce qui paraît le plus utile. Par exemple, la méthode **iter()** appliquée à **std::path::Path** renvoie un itérateur capable de produire un composant de chemin d'accès à la fois :*

Chemin d'un système de fichier

```
use std::path::Path;

fn main() {
  let chemin = Path::new("/home/manu/Cours/Cours Rust");
  let mut iterateur = chemin.iter();

  while let Some(partie) = iterateur.next() {
    println!("{:?}", partie);
  }
}
```

Résultat

```
"/"
"home"
"manu"
"Cours"
"Cours Rust"
```

*Le type d'élément de cet itérateur est **&std::ffi::OsStr**, qui correspond à une tranche empruntée à une chaîne de caractères conforme à la syntaxe attendue par les appels au système d'exploitation.*

Implémentation de IntoIterator

Dès qu'un type implémente **IntoIterator**, vous pouvez appeler vous-même sa méthode nommée **into_iter()** comme le ferait une boucle **for**.

Gestion de téléphones

```
use std::collections::HashMap;

fn main() {
  let mut telephones = HashMap::new();
  telephones.insert("lui", "06-56-89-77-23");
  telephones.insert("elle", "04-89-56-33-78");
  telephones.insert("anonyme", "07-58-68-20-10");

  let mut iterateur = telephones.into_iter();
```

```
  while let Some((nom, numero)) = iterateur.next() {
    println!("nom='{}', numéro='{}'", nom, numero);
  }
}
```

Résultat

```
nom='anonyme', numéro='07-58-68-20-10'
nom='lui', numéro='06-56-89-77-23'
nom='elle', numéro='04-89-56-33-78'
```

Avec la boucle for

```
use std::collections::HashMap;

fn main() {
  let mut telephones = HashMap::new();
  telephones.insert("lui", "06-56-89-77-23");
  telephones.insert("elle", "04-89-56-33-78");
  telephones.insert("anonyme", "07-58-68-20-10");

  for (nom, numero) in telephones {
    println!("nom='{}', numéro='{}'", nom, numero);
  }
}
```

*La plupart des collections proposent plusieurs implémentations de **IntoIterator**, pour les références partagées, pour les références mutables et pour les transferts de possession (**move**) :*

- *À partir d'une **référence partagée** sur la collection, **into_iter()** renvoie un itérateur qui va produire des références partagées aux éléments. Dans l'exemple précédent **telephones.into_iter()** renvoie un itérateur dont le type **Item** est **(&str, &str)**.*
- *Avec une **référence mutable** sur la collection, **into_iter()** renvoie un itérateur qui produit des références mutables sur les éléments. Par exemple, si **vector** est de type **Vec<String>**, l'appel **(&mut vector).into_iter()** renvoie un itérateur dont le type **Item** vaut **&mut String**.*
- *Lorsque la collection est **fournie par valeur**, **into_iter()** renvoie un itérateur qui prend **possession** de la collection et renvoie les éléments par valeur. La possession des éléments est transféré de la collection vers le consommateur, et la collection de départ est bien sûr consommée pendant le traitement. Par exemple, l'appel **telephones.into_iter()** de l'exemple précédent renvoie un itérateur qui produit une valeur de type **tuple** et le consommateur devient le possesseur de chaque **tuple**. Lorsque l'itérateur sort de la portée, les éléments qui sont encore présents dans le **HashMap** sont largués eux aussi et la coquille dorénavant vide du **dictionnaire** est larguée à son tour.*

*Puisque la boucle **for** applique **IntoIterator::into_iter()** à son opérande, les trois implémentations présentées ci-dessus permettent de créer trois tournures pour balayer une collection avec des **références partagées** ou **mutables** ou pour **consommer** la collection en prenant **possession** des éléments :*

Références partagées, références mutables ou possession

```
fn main() {
  let mut nombres = vec![5, 18, -3, 12, 23, 55, 9];

  for nombre in &nombres { print!("..{}.", nombre);  }
  println!();
  for nombre in &mut nombres {
    *nombre += 10;
    print!("..{}.", nombre);
  }
  println!();
  for nombre in nombres { print!("..{}.", nombre);  }
  println!("{:?}", nombres);                          // Impossible, nombres n'existe plus
}
```

Résultat

```
..5...18...-3...12...23...55...9.
..15...28...7...22...33...65...19.
..15...28...7...22...33...65...19.
```

*Dans chacun des trois cas ci-dessus, vous obtenez un appel à l'une des implémentations de **IntoIterator**. Sachez que tous les types ne fournissent pas les trois. Par exemple, **HashSet**, **BtreeSet** et **BinaryHeap** ne proposent pas **IntoIterator** pour les **références mutables**.*
C'est logique car la modification des éléments va en général violer la règle d'invariance des types : la valeur modifiée risque d'avoir une valeur de hachage différente ou ne pas être triée de la même façon par rapport à ses voisins. La modification risque de mal la repositionner.

*D'autres types enfin supportent la mutation de façon partielle. Les types **HashMap** et **BtreeMap** produisent des références mutables sur leurs valeurs mais seulement des références partagées sur leurs clés, pour des raisons proches de celles que nous venons d'indiquer.*

Références partagées, références mutables ou possession

```
use std::collections::HashMap;

fn main() {
  let mut telephones = HashMap::new();
  telephones.insert("lui".to_string(), "04-71-89-77-23".to_string());
  telephones.insert("elle".to_string(), "04-71-56-33-78".to_string());
  telephones.insert("anonyme".to_string(), "04-71-68-20-10".to_string());

  for (nom, numero) in &telephones {
    println!("nom='{}', numéro='{}'", nom, numero);
  }
  for (nom, numero) in &mut telephones {
    // *nom = nom.to_uppercase();  (interdit, non mutable)
    *numero = numero[6..].to_string();
    println!("nom='{}', numéro='{}'", nom, numero);
  }
  for (nom, numero) in telephones {
    println!("nom='{}', numéro='{}'", nom, numero);
  }
}
```

Résultat

```
nom='elle', numéro='04-71-56-33-78'
nom='lui', numéro='04-71-89-77-23'
nom='anonyme', numéro='04-71-68-20-10'
nom='elle', numéro='56-33-78'
nom='lui', numéro='89-77-23'
nom='anonyme', numéro='68-20-10'
nom='elle', numéro='56-33-78'
nom='lui', numéro='89-77-23'
nom='anonyme', numéro='68-20-10'
```

*Les tranches implémentent deux des trois variantes de **IntoIterator** ; elles ne peuvent pas utiliser la troisième puisqu'elles ne **possèdent jamais** leurs éléments. La méthode **into_iter()** pour **&[T]** et **&mut [T]** renvoient un itérateur qui produit des **références partagées** et **mutables** sur les éléments.*

Références partagées, références mutables sans la possession

```
fn main() {
  let mut nombres = [5, 18, -3, 12, 23, 55, 9];

  for nombre in &nombres { print!("..{}.", nombre);  }
  println!();
  for nombre in &mut nombres {
    *nombre += 10;
    print!("..{}.", nombre);
  }
  println!();
  for nombre in nombres { print!("..{}.", nombre);  } // Impossible
  println!("{:?}", nombres);  // n'existe plus
}
```

Résultat

```
..5...18...-3...12...23...55...9.
..15...28...7...22...33...65...19.
```

*Vous aurez peut-être remarqué que les deux premières variantes pour les **références partagées** et **mutables** sont équivalente à des appels à **iter()** ou **iter_mut()** sur la cible. Pour quelles raisons **Rust** propose-t-il les deux approches ?*

*Le trait **IntoIterator** anime les boucles **for**, il est donc indispensable, mais si vous n'utilisez pas une boucle, **chemin.iter()** est beaucoup plus lisible que **(&chemin).into_iter()**. Vous aurez souvent besoin de réaliser des itérations par références **partagées**, **iter()** et **iter_mut()** restent donc intéressantes de par leur ergonomie.*

Les méthodes drain()

De nombreux types de collections disposent d'une méthode **drain()** qui part d'une **référence mutable** sur la collection pour renvoyer un itérateur qui **lègue** la **possession** de chaque élément à son **consommateur**. À la différence de la méthode **into_iter()** qui reçoit la collection par valeur puis la **consomme**, **drain() emprunte** une référence **mutable** sur la collection. Lorsque l'itérateur est **largué**, la méthode **supprime tous les éléments restants** dans la collection pour qu'elle soit vide.

*Pour les types qui peuvent être visités par un indice dans une plage, comme **String**, les vecteurs et **VecDeque**, la méthode **drain()** supprime une plage d'éléments au lieu de vider toute la séquence. Lorsque vous avez besoin de drainer toute la séquence, il suffit de spécifier la plage complète en paramètre, « **...** ».*

Méthode drain()

```
use std::iter::FromIterator;

fn main() {
 let mut exterieur = "TerrE".to_string();
 let interieur = String::from_iter(exterieur.drain(1..4));
 println!("extérieur=({}), intérieur=({})", exterieur, interieur);
}
```

Résultat

```
extérieur=(TE), intérieur=(err)
```

Autres sources d'itérateurs

Nous nous sommes surtout intéressé aux collections tels que les vecteurs et les tables **HashMap**. Bien d'autres types de la librairie standard permettent l'itération. Le tableau ci-dessous présente les plus intéressants, mais la liste n'est pas exhaustive. Nous reviendrons plus en détail sur les méthodes de certains types dans nos prochaines études.

Type ou Trait	Expression	Notes
std::ops::Range	**1..10**	Les bornes doivent être de type entier pour les itérables, la plage inclut la borne inférieure mais pas l'autre.
std::ops::RangeFrom	**1..**	Itération sans bornes. La valeur de départ doit être un entier. Panique ou débordement possible si les valeurs atteignent la limite du type.
Option<T>	**Some(10).iter()**	Similaire à un vecteur si la longueur est soit 0 (**None**) ou 1 (**Some(v)**).
Result<T, E>	**Ok(« bon »).iter()**	Similaire à **Option**, en produisant des valeurs **Ok**.
Vec<T>, &[T]	**v.windows(16)**	Produit toutes les tranches adjacentes de la longueur demandée de gauche à droite. Chevauchement de fenêtres.
	v.chunks(16)	Produit des tranches adjacentes sans chevauchement de la longueur fournie de gauche à droite.
	v.chunks_mut(102)	Comme **chunks()** avec des tranches mutables.
	v.split(|byte| byte&1!=0)	Produit des tranches séparées par des éléments qui comporte un prédicat.
	v.split_mut(...)	Comme le précédent avec des tranches mutables.
	v.rsplit(...)	Comme **split()**, mais tranches produites de droite à gauche.
	v.splitn(n, ...)	Comme **split()**, mais au maximum **n** tranches.
String, &str	**s.bytes()**	Produits des octets au format **UTF-8**.
	s.chars()	Produit des **char UTF-8**.
	s.split_whitespace()	Découpe une chaîne aux espaces et produit des tranches sans espace.
	s.lines()	Produit des tranches à partir des lignes de texte.
	s.split('/')	Découpe la chaîne d'après le motif en produisant des tranches entre les correspondances. Les motifs peuvent être : caractères, chaînes, clôtures.
	s.matches(char::is_ascii)	Produit des tranches satisfaisant le motif fourni.
std::collection::HashMap, std::collection::BTreeMap	**map.keys(), map.values()**	Produit une référence partagée sur les clés ou les valeurs de la carte.
	map.values_mut()	Produit des références mutables vers les valeurs des entrées.
std::collection::HashSet, std::collection::BTreeSet	**set1.union(set2)**	Produit des références partagées vers les éléments d'une union de **set1** et **set2**.
	set1.intersection(set2)	Produit des références partagées vers les éléments d'une intersection de **set1** et **set2**.

std::sync::mpsc::Receiver	**recv.iter()**	Produit des valeurs émises par un autre **exétron** par l'émetteur **Sender** correspondant.
std::io::Read	**stream.bytes()**	Produit des octets depuis le flux E/S.
	stream.chars()	Analyse un flux **UTF-8** pour produire des **char**.
std::io::BufRead	**bufstream.lines()**	Analyse un flux **UTF-8** pour produire des lignes de type **String**.
	bufstream.split(0)	Découpe un flux à l'octet indiqué en produisant des tampons **Vec<u8>** interoctet.
std::fs::ReadDir	**std::fs::read_dir(path)**	Produit des entrées de répertoires.
std::net::TcpListener	**listener.incoming()**	Produit des connexions réseau entrantes.
Fonctions libres	**std::iter::empty()**	Renvoie immédiatement **None**.
	std::iter::once(5)	Produit la valeur fournie puis s'arrête.
	std::iter::repeat(« #9 »)	Produit éternellement la valeur fournie.

Adaptateurs d'itérateurs – map et filter

Le trait fondamental **Iterator** propose un grand choix de méthodes d'adaptateurs, que nous appellerons simplement des adaptateurs. Leur principe est de consommer un itérateur pour en produire un nouveau avec le bon comportement. Partons des deux les plus utilisés pour comprendre le fonctionnement d'un adaptateur.

*Parmi les nombreux adaptateurs du trait **Iterator**, **map()** sert à transformer un itérateur par application d'une clôture à ses éléments. Quant à l'adaptateur **filter()**, il permet d'éliminer des éléments d'un itérateur grâce à une clôture qui n'en sélectionne que certains. Vous pouvez choisir soit l'un soit l'autre soit les deux ensembles.*

Adaptateur de transformation et de filtre

```
fn main() {
  let texte = "  poneys  \n  girafes\niguanes  \ncalamars".to_string();
  let animaux : Vec<&str> = texte.lines().map(str::trim).filter(|s| *s != "iguanes").collect();
  println!("{:?}", animaux);

  let texte = "Bonjour à tout le monde".to_string();
  let mots: Vec<String> = texte.split_whitespace().map(|s| s.to_uppercase()).filter(|s| s.contains('O')).collect();
  println!("{:?}", mots);
}
```

Résultat

```
["poneys", "girafes", "calamars"]
["BONJOUR", "TOUT", "MONDE"]
```

*L'appel **texte.lines()** renvoie un itérateur qui génère des lignes de texte. En appelant **map()** sur cet itérateur , nous obtenons un autre itérateur qui applique ensuite **str::trim** à chaque ligne en produisant les résultats sous forme d'éléments. Ce que renvoie l'itérateur **map()** est bien sûr candidat à une adaptation ultérieure.*

*L'adaptateur **filter()** renvoie un troisième itérateur qui ne produit en sortie que les éléments de **map()** pour lesquels la clôture **|s| *s!= « iganes »** renvoie la valeur **true**. La méthode **collect()** rassemble tous ces éléments dans un vecteur.*

Cet enchaînement d'itérateurs ressemble beaucoup à un pipeline de commandes dans l'environnement Unix : chaque adaptateur n'a qu'un seul rôle et l'enchaînement des traitement reste très lisible de gauche à droite.

*L'appel **texte.split_whitespace()** renvoie un itérateur qui permet de séparer chacun des mots de la phrase. L'itérateur **map()** permet de mettre chacun des mots en majuscule. L'itérateur **filter()** permet de ne prendre en compte seulement les mots qui possèdent la lettre **O**. Voici ci-dessous un autre exemple d'itérateurs associés à la boucle **for**.*

Adaptateur de transformation et de filtre

```
fn main() {
  let texte = "Êtes vous aller au pré avec elle".to_string();
  for mot in texte.split_whitespace()
                  .filter(|s| s.is_ascii())
                  .map(|s| s.to_lowercase())
                  .filter(|s| s.contains('e'))
                  .map(|s| s.to_uppercase())  {
    print!("{} ", mot);
  }

  let notes = [12, 8, 10, 5, 13, 7, 15, 3, 18];
  let plus_de_dix: Vec<_> = notes.iter().filter(|n| **n >= 10).collect();
  println!("\nPlus de dix : {:?}", plus_de_dix);
  let pairs: Vec<_> = (3..18).filter(|x| x%2==0).collect();
  println!("Nombres pairs : {:?}", pairs);
}
```

Résultat

```
ALLER AVEC ELLE
Plus de dix : [12, 10, 13, 15, 18]
Nombres pairs : [4, 6, 8, 10, 12, 14, 16]
```

*Puisque nous pouvons enchaîner plusieurs adaptateurs, rien n'empêche de multiplier l'utilisation de **map()** ou de **filter()** sur une même séquence. Dans cet exemple, nous utilisons plusieurs filtres et plusieurs transformations dans une boucle **for**.*

L'itérateur map() transmet chaque élément à sa clôture par valeur et transfère la possession du résultat de la clôture à son consommateur. En revanche, l'itérateur filter() transmet chaque élément à sa clôture par référence partagée, et conserve la possession, pour le cas où l'élément serait sélectionné et devrait être transmis au consommateur.

*C'est pour cette raison que dans l'exemple précédent, dans la gestion des notes, nous devons déréférencer **n** deux fois pour faire la comparaison puisque le type du paramètre de la clôture est **&&i32**.*

Le fait d'appeler un adaptateur sur un itérateur ne consomme pas les éléments, mais renvoie un nouvel itérateur qui est prêt à produire ses propres éléments en utilisant le premier itérateur comme source pour l'alimenter. Dans une chaîne d'adaptateurs, le seul moyen de récupérer la nouvelle collection est de passer systématiquement par la méthode collect().

Adaptateurs d'itérateurs – filter_map et flat_map

L'adaptateur **map()** convient lorsqu'à chaque élément d'entrée correspond un élément de sortie. Mais vous aurez parfois besoin de supprimer des éléments au lieu de les traiter ou de remplacer un élément par zéro ou plusieurs. C'est l'objectif des deux adaptateurs **filter_map()** et **flat_map()**.

*L'adaptateur **filter_map()** ressemble à **map()** sauf qu'il permet à sa clôture soit de transformer l'élément en un autre, comme **map()**, soit de supprimer l'élément dans l'itération. C'est une sorte d'hybride entre **filter()** et **map()**.*

*La seule différence avec la signature de **map()** est que la clôture renvoie le type **Option<B>**, et non simplement **B**. Lorsqu'elle renvoie **None**, l'élément disparaît et lorsqu'elle renvoie **Some(b)**, alors **b** incarne le prochain élément généré par l'itérateur **filter_map()**.*

Adaptateur filter_map()

```rust
use std::str::FromStr;

fn main() {
  let texte ="1 bon .25 jour 289 salut 3.1415";
  for nombre in texte.split_whitespace().filter_map(|x| f64::from_str(x).ok()) {
    println!("{:.>7.2}", nombre.sqrt());
  }
}
```

Résultat

```
....1.00
....0.50
..17.00
....1.77
```

*La clôture de **filter_map()** cherche à analyser chacune des tranches séparées par des espaces en utilisant **f64::from_str()**. Ce qui permet de récupérer un **Result<f64, ParseFloatError>** qui est transformé par **ok()** en **Option<f64>**. En cas d'erreur, nous obtenons **None**, et en cas de réussite **Some(v)**. Les valeurs **None** sont abandonnées et l'itérateur produit la valeur **v** pour chaque **Some(v)**.*

L'adaptateur **flat_map()** est une sorte d'extension de **map()** combinée à **filter_map()** sauf que la clôture peut dorénavant renvoyer non seulement un élément unique ou bien zéro ou plusieurs éléments, mais également toute une séquence d'éléments. L'itérateur **flat_map()** génère la concaténation des séquences renvoyées par la clôture.

*La clôture que vous fournissez à **flat_map()** doit être capable de renvoyer un itérable, quel que soit son genre. (Du fait que **Option** est un itérable qui se comporte comme une séquence de zéro ou un élément, l'écriture **iterator.fliter_map(closure)** équivaut à **iterator.flat_map(closure)**, en supposant que **closure** renvoie **Option<T>**).*

Adaptateur flat_map()

```rust
use std::collections::HashMap;

fn main() {
  let mut metropoles= HashMap::new();

  metropoles.insert("France", vec!["Paris", "Lyon", "Marseille"]);
  metropoles.insert("Japon", vec!["Tokyo", "Kyoto"]);
  metropoles.insert("USA", vec!["Newyork", "Washington"]);

  for &ville in metropoles.iter().flat_map(|pays| pays.1) {
    print!("{} ", ville.to_uppercase());
  }
}
```

Résultat

```
TOKYO KYOTO NEWYORK WASHINGTON PARIS LYON MARSEILLE
```

Dans cet exemple, pour chaque pays, nous récupérons le vecteur contenant ses métropoles en regroupant tous les vecteurs en une seule et unique séquence et nous affichons le résultat.

Les itérateurs sont paresseux : le travail n'est réalisé qu'à partir des appels dans la boucle for à la méthode flat_map() nommée next(). La séquence complète n'est jamais implémentée en mémoire. Nous utilisons en fait un petit automate à états finis qui s'alimente à partir de l'itérateur des villes, un élément à la fois, jusqu'à ce qu'il n'y en ait plus.

Ce n'est qu'ensuite que nous produisons un nouvel itérateur pour le pays suivant. Cela équivaux à une imbrication de boucles, correctement conditionnée pour que cela soit utilisable comme itérateur.

Alternative au programme précédent

```
use std::collections::HashMap;

fn main() {
  let mut metropoles= HashMap::new();
  metropoles.insert("France", vec!["Paris", "Lyon", "Marseille"]);
  metropoles.insert("Japon", vec!["Tokyo", "Kyoto"]);
  metropoles.insert("USA", vec!["Newyork", "Washington"]);

  for ville in metropoles.iter().flat_map(|pays| pays.1).map(|ville| ville.to_uppercase()) {
    print!("{} ", ville);
  }
}
```

Résultat

```
PARIS LYON MARSEILLE NEWYORK WASHINGTON TOKYO KYOTO
```

Adaptateurs d'itérateurs – scan

Cet adaptateur ressemble à **map()** sauf que sa clôture obtient une valeur mutable qu'elle peut analyser, ce qui permet de décider d'avorter l'itération. En partant d'une valeur initiale et d'une clôture, pour chaque élément de l'itérateur sous-jacent, **scan()** transmet à la clôture une référence mutable sur la valeur d'état et l'élément.

*La clôture doit renvoyer une **Option** : si elle vaut **None**, l'itération est terminée ; si elle vaut **Some(v)**, alors **v** est la prochaine valeur produite par l'itérateur **scan()**. La séquence d'itérateurs suivante calcule le carré de l'élément reçu et arrête l'itération dès que le total dépasse **10**.*

Adaptateur scan()

```
fn main() {
  let carres : Vec<_> = (0..10).scan(0, |somme, item| {
      *somme += item;
       if *somme > 10 { None }
       else { Some(item*item) }
    }).collect();
  println!("{:?}", carres);
}
```

Résultat

```
[0, 1, 4, 9, 16]
```

*Le paramètre de somme de la clôture est une référence mutable sur une valeur interne à l'itérateur, valeur qui est initialisée avec le premier paramètre de **scan()** qui ici prend la valeur **0**. La clôture actualise ***some** puis vérifie que la limite n'est pas atteinte et renvoie le prochain résultat de l'itérateur.*

Adaptateurs d'itérateurs – take et take_while

Les deux adaptateurs **take()** et **take_while()** permettent d'arrêter d'itérer après un certain nombre d'éléments ou bien lorsque la clôture en décide ainsi. Tous deux prennent possession d'un itérateur pour en renvoyer un autre qui retransmet les éléments du premier, en avortant la séquence éventuellement.

*L'itérateur **take()** renvoie **None** après avoir produit au plus **n** éléments alors que l'autre applique un prédicat à chaque élément pour renvoyer **None** à la place du premier élément pour lequel ce renvoie **false** ainsi que dans tous les appels suivants à **next()**. Dans l'exemple qui suit, l'adaptateur **take_while()** abandonne dès qu'il tombe sur une ligne vide et le résultat ne comporte donc que deux lignes.*

Adaptateur take_while()

```
fn main() {
  let message = "To: moi\r\n\
                 From: lui <lui@mail.com>\r\n\
                 \r\n\
                 Contenu du mail.\r\n";
  for entete in message.lines().take_while(|ligne| !ligne.is_empty()) {
```

```
    println!("{}", entete);
  }
}
```

Résultat

```
To: moi
From: lui <lui@mail.com>
```

Adaptateurs d'itérateurs – skip et skip_while

Ces deux méthodes sont complémentaires de **take()** et **take_while()**. Elles ignorent un certain nombre d'éléments à partir du début ou jusqu'à ce qu'une clôture trouve un élément acceptable. Elles transmettent ensuite les éléments sélectionnés sans les modifier.

*L'itérateur **skip()** sert particulièrement à filtrer les paramètres d'une ligne de commande de lancement de programme, pour ignorer le nom de ce programme. L'autre adaptateur, **skip_while()**, se fonde sur une clôture pour décider des éléments à oublier à partir du début de la séquence.*

Adaptateur skip_while()

```
fn main() {
  let message = "To: moi\r\n\
                 From: lui <lui@mail.com>\r\n\
                 \r\n\
                 Contenu du mail\r\n\
                 sur deux lignes.";
  for entete in message.lines().skip_while(|ligne| !ligne.is_empty()).skip(1) {
    println!("{}", entete);
  }
}
```

Résultat

```
Contenu du mail
sur deux lignes.
```

*Ici, **skip_while()** nous permet de scruter les lignes du corps du message de l'exemple de messagerie. Nous ignorons donc les lignes précédentes, mais l'itérateur produit la ligne vide puisque la clôture renvoie **false** pour celle-ci. C'est pourquoi nous utilisons ensuite **skip()** pour supprimer cette première ligne inutile afin de ne restituer que le corps du message.*

Adaptateurs d'itérateurs – Peekable (consultable)

Un itérateur consultable permet de jeter un œil sur le prochain élément qu'il va produire sans le consommer pour autant. Quasiment tous les itérateurs peuvent être dotés de cette capacité en appelant la méthode du trait **Iterator** nommée **peekable()**.

*Un itérateur consultable est doté d'une méthode nommée **peek()** en plus d'autres, cette méthode renvoyant un **Option<&Item>** qui vaut **None** si l'itérateur lié a terminé ou bien **Some(r)** avec **r** une référence partagée au prochain élément. (Si le type de l'élément est déjà une référence, nous obtenons une référence sur une référence).*

*Le fait d'appeler **peek()** tente d'extraire le prochain élément de l'itérateur et le met en cache s'il existe jusqu'au prochain appel à la méthode **next()**. Toutes les autres méthodes du trait **Iterator** une fois celui-ci rendu consultable connaissent l'existence du cache.*

*Par exemple, **Iter.last()** sur un itérateur consultable **iter()** sait qu'il doit lire le cache après avoir fini de scruter l'itérateur qui lui sert de source.*

Discriminer un nombre en base 10

```
fn main() {
  let message = "123AF";
  let mut nombre = 0;
  let mut suite = message.chars().peekable();
  loop {
    match suite.peek() {
      Some(lettre) if lettre.is_digit(10) => {
        nombre = nombre*10 + lettre.to_digit(10).unwrap();
      }
      _ => break
    }
    suite.next();
  }
  println!("Nombre = {}", nombre);
}
```

Résultat

```
Nombre = 123
```

Discriminer un nombre en base 16 et donner le résultat en base 10

```
fn main() {
  let message = "123AF";
  let mut nombre = 0;
  let mut suite = message.chars().peekable();
  loop {
    match suite.peek() {
      Some(lettre) if lettre.is_digit(16) => {
        nombre = nombre*16 + lettre.to_digit(16).unwrap();
      }
      _ => break
    }
    suite.next();
  }
  println!("Nombre = {}", nombre);
}
```

Résultat

```
Nombre = 74671
```

Les itérateurs consultables sont essentiels dès que vous avez besoin d'évaluer le nombre d'éléments à consommer sans aller trop loin. Si vous analyser les nombres qui proviennent d'un flux de caractères, vous ne pouvez pas savoir à l'avance quand vous en aurez terminé, jusqu'au moment où vous allez détecter le premier caractère différent d'un nombre.

Adaptateurs d'itérateurs – itérateurs réversibles - rev

Certains itérateurs sont capable de récupérer des éléments en puisant dans les deux extrémités de la séquence d'entrée. Vous pouvez inverser l'ordre de lecture au moyen de l'adaptateur **rev**. Vous pouvez ainsi faire récupérer les éléments en commençant par la fin d'un vecteur au lieu du début. Ce genre d'itérateur peut implémenter le trait **DoubleEndedIterator** qui est une extension de **Iterator**.

Ce genre d'itérateur se comporte comme s'il disposait de deux indices vers le début et la fin de la séquence. En récupérant un élément d'un côté ou de l'autre, vous faites progresser l'indice vers le centre et l'itération se termine lorsque les deux indices se rejoignent.

*La structure d'un tel itérateur appliqué à une tranche est facile à implémenter puisqu'il s'agit d'une paire de pointeurs vers le début et la plage d'éléments qui n'a pas encore été produite. Les méthodes **next()** et **next_back()** récupèrent un élément d'un côté ou de l'autre. Les itérateurs des collections ordonnées comme **BtreeSet** et **BtreeMap** utilisent aussi cette double entrée ; leur méthode **next_back()** récupère le plus grand élément d'abord.*

Parcourir une séquence dans les deux sens

```
fn main() {
  let corps = ["tête", "torax", "abdomen", "jambes", "pieds"];
  let mut sequence = corps.iter();
  print!("{:?} ", sequence.next());
  print!("{:?} ", sequence.next_back());
  print!("{:?} ", sequence.next());
  print!("{:?} ", sequence.next_back());
  print!("{:?} ", sequence.next());
  print!("{:?} ", sequence.next_back());
}
```

Résultat

```
Some("tête") Some("pieds") Some("torax") Some("jambes") Some("abdomen") None
```

*Dès qu'un itérateur est à double entrée, vous pouvez inverser son exploitation au moyen de l'adaptateur **rev**. L'itérateur qui est renvoyé est lui aussi à double entrée : simplement, ses deux méthodes **next()** et **next_back()** sont permutées.*

Parcourir une séquence dans les deux sens avec inversion de lecture

```
fn main() {
  let corps = ["tête", "torax", "abdomen", "jambes", "pieds"];
  let mut sequence = corps.iter().rev();
  print!("{:?} ", sequence.next().unwrap());
  print!("{:?} ", sequence.next_back().unwrap());
  print!("{:?} ", sequence.next().unwrap());
  print!("{:?} ", sequence.next_back().unwrap());
}
```

Résultat

```
"pieds" "tête" "jambes" "torax"
```

*La plupart des adaptateurs renvoient un itérateur réversible lorsque vous les appliquez à un itérateur réversible. Par exemple, **map()** et **filter()** préservent cette réversibilité.*

Parcourir une séquence avec inversion de lecture

```
fn main() {
  let corps = ["tête", "torax", "abdomen", "jambes", "pieds"];
  for partie in corps.iter().filter(|partie| partie.contains('a')).rev() {
    print!("{} ", partie);
  }
}
```

Résultat

```
jambes abdomen torax
```

Adaptateurs d'itérateurs – inspect

Cet adaptateur est utilisé pour mettre au point des pipelines d'adaptateurs, bien qu'ils soient rarement utilisés dans du code à déployer. Il applique une clôture à une référence partagée sur chaque élément puis retransmet l'élément. La clôture ne peut pas modifier les éléments mais elle peut en profiter pour afficher leurs valeurs ou lancer des assertions à leur propos.

Inspection d'une séquence

```
fn main() {
  let message : String = "salut".chars()
    .inspect(|c| print!("avant:{:?}  ", c))
    .flat_map(|c| c.to_uppercase())
    .inspect(|c| print!("après:{:?}  ", c))
    .collect();
  println!("\n{}", message);
}
```

Résultat

```
avant:'s'  après:'S'  avant:'a'  après:'A'  avant:'l'  après:'L'  avant:'u'  après:'U'  avant:'t'  après:'T'
SALUT
```

> *La méthode de forçage en majuscule **to_uppercase()** renvoie un itérateur sur des caractères et non le caractère de remplacement. L'exemple précédent utilise **flat_map()** pour réunir toutes les séquences renvoyées par **to_uppercase()** afin de former une seule chaîne **String**, tout en affichant les différentes étapes.*

Adaptateurs d'itérateurs – chain

Cet adaptateur branche un itérateur à la suite d'un autre. Autrement dit, **i1.chain(i2)** renvoie un itérateur qui récupère les éléments de **i1** jusqu'au dernier puis récupère ceux de **i2**. Vous pouvez rabouter un itérateur à la suite d'un itérable produisant le même type d'élément. Notez que l'itérateur **chain()** est réversible, à condition que les deux itérateurs le soient.

Fusion et inversion de deux séquences

```
fn main() {
  let entiers : Vec<_> = (1..8).chain(vec![10, 20, 30]).rev().collect();
  println!("{:?}", entiers);
}
```

Résultat

```
[30, 20, 10, 7, 6, 5, 4, 3, 2, 1]
```

> *Cet itérateur garde la trace de l'endroit à partir duquel chacun des deux itérateurs utilisés a renvoyé **None**, ce qui lui permet de rediriger **next()** et **next_back()** vers l'un ou l'autre en fonction des résultats.*

Adaptateurs d'itérateurs – enumerate

Le trait **Iterator** propose l'adaptateur **enumerate()** pour associer un index mobile à une séquence, en partant un itérateur qui produit des éléments **A, B, C, …** et en renvoyant un itérateur qui produit les paires **(0, A), (1, B), (2, C), …** Cette opération semble très bénigne, mais elle sert relativement souvent.

> *Le consommateur de la séquence peuvent se servir de l'indice pour distinguer les éléments et constituer un contexte pour traiter chacun d'eux.*

Énumérer les élemnts d'une séquence

```
fn main() {
  let notes = [12, 8, 15];
  for (numero, note) in notes.iter().enumerate() {
    println!("Note n°{} : {}", numero+1, note);
  }
}
```

Résultat

```
Note n°1 : 12
```

```
Note n°2 : 8
Note n°3 : 15
```

Adaptateurs d'itérateurs – zip

Le nom anglais de cet adaptateur, **zip()**, suggère l'idée d'une fermeture Éclair (pas de notion de compression ici). Il combine effectivement deux itérateurs en un seul afin de produire des **paires** à partir d'une valeur de chacun d'eux. Cet itérateur s'arrête dès qu'une des deux sources n'a plus rien à offrir.

*Finalement, vous pouvez obtenir le même effet qu'avec **enumerate()** mais en choisissant éventuellement le début de la valeur d'index, sachant que nous pouvons avoir un autre type qu'un entier sur cette notion d'index.*

Énumérer les éléments d'une séquence

```
fn main() {
  let notes = [12, 8, 15];
  for (numero, note) in (1..).zip(&notes) {
    println!("Note n°{} : {}", numero, note);
  }
}
```

Résultat

```
Note n°1 : 12
Note n°2 : 8
Note n°3 : 15
```

*D'une certaine façon, **zip()** est une généralisation de **enumerate()** : alors que **enumerate()** associe un indice aux éléments ; **zip()** réunit n'importe quels éléments d'itérateurs. L'adaptateur **enumerate()** permet de constituer un contexte de traitement des éléments ; l'adaptateur **zip()** offre une solution plus souple pour arriver au même résultat.*

Adaptateurs d'itérateurs – by_ref

Tous les adaptateurs vus jusqu'ici sont reliés à un itérateur et une fois cette liaison établie, il n'est pas possible en général de la défaire. Les adaptateurs prennent possession de l'itérateur sous-jacent et ne disposent d'aucune méthode pour le restituer.

*La méthode **by_ref()** d'un itérateur permet **d'emprunter une référence mutable** à l'itérateur, ce qui permet ensuite d'appliquer l'adaptateur à cette référence. Une fois que vous avez fini d'exploiter les éléments, vous pouvez les rendre, ce qui **éteint l'emprunt** et vous **redonne accès à l'itérateur de départ.***

Récupérer la référence d'un itérateur

```
fn main() {
  let message = "To: moi\r\n\
                 From: lui <lui@mail.com>\r\n\
                 \r\n\
                 Contenu du mail.\r\n\
                 sur deux lignes.";
  let mut lignes = message.lines();

  println!("En-têtes:");
  for entete in lignes.by_ref().take_while(|ligne| !ligne.is_empty()) {
    println!("{}", entete);
  }

  println!("Corps:");
  for corps in lignes {
    println!("{}", corps);
  }
}
```

Résultat

```
En-têtes:
To: moi
From: lui <lui@mail.com>
Corps:
Contenu du mail.
sur deux lignes.
```

*L'appel **lignes.by_ref()** emprunte une référence mutable à l'itérateur et c'est cette référence dont prend possession l'itérateur **take_while()**. À la fin de la première boucle **for**, cet itérateur disparaît, ce qui éteint l'emprunt et permet d'utiliser à nouveau **lignes** dans la seconde boucle **for**.*

Adaptateurs d'itérateurs – cycle

L'itérateur que renvoie **cycle()** répète indéfiniment la séquence produite par l'itérateur sous-jacent, ce dernier devant implémenter **Clone** pour que cycle() puisse sauvegarder l'état initial afin de le réutiliser à chaque nouveau début de boucle.

Répétition indéfinie

```
fn main() {
  let orientation = ["Nord", "Est", "Sud", "Ouest"];
  let mut suite = orientation.iter().cycle();
  for occurence in 1..8 {
    print!("{:?} ", suite.next().unwrap());
  }
}
```

Résultat

```
"Nord" "Est" "Sud" "Ouest" "Nord" "Est" "Sud"
```

*L'exemple suivant propose une suite de nombres de **1** à **31** en affichant le nombre correspondant à l'occurrence. Dès que le nombre proposé est divisible par **3**, c'est le mot « **trois** » qui est alors affiché, et si le nombre est divisible par **5**, c'est le mot « **cinq** » qui est affiché. Enfin, les nombre divisible à la fois par **3** et par **5**, c'est le mot « **troiscinq** » qui est affiché.*

Répétition indéfinie

```
use std::iter::{once, repeat};

fn main() {
  let trois = repeat("").take(2).chain(once("trois")).cycle();
  let cinq = repeat("").take(4).chain(once("cinq")).cycle();
  let trois_cinq = trois.zip(cinq);
  let suite = (1..32).zip(trois_cinq)
    .map(|tuple| match tuple {
      (i, ("", "")) => i.to_string(),
      (_, (t, c)) => format!("{}{}", t, c)
    });

  for ligne in suite {
    print!("{} ", ligne);
  }
}
```

Résultat

```
1 2 trois 4 cinq trois 7 8 trois cinq 11 trois 13 14 troiscinq 16 17 trois 19 cinq trois 22 23 trois cinq 26 trois 28 29 troiscinq 31
```

Consommation des itérateurs – accumulation simple : count, sum, product

Nous venons de voir comment créer des itérateurs puis comment les adapter pour en produire de nouveaux. Il nous reste maintenant à découvrir comment exploiter ou consommer ces itérateurs. Vous pouvez évidemment consommer un itérateur dans une boucle **for** ou bien appeler **next()** de façon explicite, mais il existe un nombre de tâches banales qu'il serait intéressant de ne pas devoir répéter dans nos codes sources. Le trait **Iterator** offre tout une gamme de méthodes pour répondre à nos différents besoins.

*La méthode **count()** extrait des éléments d'un itérateur jusqu'à épuisement (renvoi de **None**) et indique combien ont été reçus. Le bref programme suivant compte le nombre de caractères constituant une chaîne de caractères.*

Récupérer la référence d'un itérateur

```
fn main() {
  let lettres = "Bonjour";
  println!("longueur = {}", lettres.chars().count());
}
```

Résultat

```
longueur = 7
```

*Les méthode **sum()** et **product()** produisent respectivement des éléments trouvés dans l'itérateur. Par contre, il doit s'agir de valeurs entières ou à virgule flottante.*

Consommation des itérateurs avec la somme et le produit de nombres entiers

```
fn main() {
  let n = 5;
  let triangle : u32 = (1..n+1).sum();
  let factoriel : u32 = (1..n+1).product();
  println!("Somme={}, {}!={}", triangle, n, factoriel);
}
```

Résultat

```
Somme=15, 5!=120
```

Consommation des itérateurs – min et max

Ces deux méthodes de **Iterator** renvoient le plus petit ou le plus grand élément obtenu dans l'itérateur. Le type des éléments d'entrée doit implémenter **std::cmp::Ord** de sorte que les éléments puissent être comparés les uns aux autres.

Retrouver la note mini et maxi de nombre entiers

```
fn main() {
  let notes= [12, 8, 15, 9, 7, 18, 15];
  let plus_petite = notes.iter().min().unwrap();
  let plus_haute = notes.iter().max().unwrap();
  println!("Notes = {:?}, Plus petite = {}, Plus haute = {}", notes, plus_petite, plus_haute);
}
```

Résultat

```
Notes = [12, 8, 15, 9, 7, 18, 15], Plus petite = 7, Plus haute = 18
```

Consommation des itérateurs – min_by et max_by

Ces deux méthodes renvoient le minimum et le maximum parmi les éléments produits par l'itérateur, en se basant sur une fonction de comparaison que vous devez fournir en argument

Retrouver la note mini et maxi avec des nombres réels

```
fn main() {
  let notes= [12., 8.5, 15.5, 9., 6.5, 17.5];
  let compare = |a: &&f64, b: &&f64| a.partial_cmp(b).unwrap();
  let plus_petite = notes.iter().min_by(compare).unwrap();
  let plus_haute = notes.iter().max_by(compare).unwrap();
  println!("Notes = {:?}, Plus petite = {}, Plus haute = {}", notes, plus_petite, plus_haute);
}
```

Résultat

```
Notes = [12.0, 8.5, 15.5, 9.0, 6.5, 17.5], Plus petite = 6.5, Plus haute = 17.5
```

*Les doubles références dans les paramètres de **compare()** sont liés au fait que **min_by()** et **max_by()** transmettent la fonction de comparaison des références aux éléments de l'itérateur pour qu'ils puissent servir efficacement avec n'importe quel genre d'itérateur. Dans ce cas précis, ces éléments sont eux-mêmes des références, puisque **notes.iter()** produit des références sur les éléments du tableau.*

Consommation des itérateurs – min_by_key et max_by_key

Ces deux méthodes de **Iterator** permettent de choisir quel est l'élément minimal ou maximal en fonction d'une clôture que vous appliquez aux éléments. Cette clôture peut sélectionner un champ d'un élément ou réaliser un calcul sur eux. Ces deux fonctions sont donc plus souvent utiles que les versions de base **min()** et **max()** parce que vous aurez souvent besoin des données associées à une valeur minimale ou maximale et pas seulement à cette valeur limite.

*Il faut les comprendre de la façon suivante : en partant d'une clôture qui reçoit un élément et renvoie un type ordonné **B**, nous pouvons renvoyer l'élément pour lequel la clôture a trouvé le **B** minimal ou maximal ou bien **None** si aucun élément n'a été produit.*

Récupérer la référence d'un itérateur

```
use std::collections::HashMap;

fn main() {
  let mut villes= HashMap::new();
  villes.insert("Lyon", 518_635);
  villes.insert("Paris", 2_175_601);
  villes.insert("Marseille", 873_716);
  println!("La plus grande ville {:?}", villes.iter().max_by_key(|&(_ville, habitants)| habitants).unwrap());
}
```

Résultat

```
La plus grande ville ("Paris", 2175601)
```

La clôture ci-dessus est appliquée à chacun des éléments produits par l'itérateur pour renvoyer une valeur à comparer, donc ici, le nombre d'habitants. Cette valeur constitue la totalité de l'élément et non seulement la valeur renvoyée par la clôture.

Comparaison de séquences d'éléments

Les opérateurs **<** et **==** permettent de comparer des chaînes, des vecteurs et des tranches, en supposant que les éléments sont comparables. Les itérateurs ne reconnaissent pas les opérateurs de comparaison de **Rust**, mais ils proposent des méthodes comme **eq()** et **lt()** qui permettent d'atteindre le même objectif, en extrayant des paires d'élément depuis les itérateurs et en les comparant jusqu'à la prise de décision ultérieure.

Comparaison de textes

```
fn main() {
  let normal = "Bienvenue tout le monde !";
  let espaces = "Bienvenue   tout   le   monde !";
  let salutation = "Salut à tous !";

  println!("Chaînes égales : {}", normal == espaces);
  println!("Mêmes mots : {}", normal.split_whitespace().eq(espaces.split_whitespace()));

  println!("'Bienvenue' placée avant 'Salut' : {}", normal < salutation);
  println!("espace > salutation  : {}", espaces.split_whitespace().gt(salutation.split_whitespace()));
}
```

Résultat

```
Chaînes égales : false
Mêmes mots : true
'Bienvenue' placée avant 'Salut' : true
espace > salutation  : false
```

*Les appels successifs à **split_whitespace()** renvoient à chaque fois un itérateur pour les mots séparés par des espaces dans les chaînes. En appliquant les méthodes **eq()** et **gt()** sur ces itérateurs, vous réalisez une comparaison mot à mot et non caractère par caractère. Cela est possible parce que **&str** implémente **PartialOrd** et **PartialEq**.*

Pour l'égalité et la différence, les itérateurs proposent les méthodes eq() et ne(). Pour les comparaisons inférieur et supérieur, vous disposez de lt(), le(), gt() et ge(). Les deux méthodes cmp() et partial_cmp() fonctionnent de la même façon que les méthodes correspondantes des traits Ord et PartialOrd.

Comparaison de séquences d'éléments – any et all

Les méthodes **any()** et **all()** appliquent une clôture à chaque élément généré par l'itérateur. La première renvoie **true** si la clôture renvoie **true** pour au moins un élément, et l'autre si elle renvoie **true** pour tous les éléments.

Recherche de majuscules

```
fn main() {
  let message = "Bienvenue!";

  println!("({}) Au moins une majuscule : {}", message, message.chars().any(char::is_uppercase));
  println!("({}) Tout en majuscule : {}", message, message.chars().all(char::is_uppercase));
}
```

Résultat

```
(Bienvenue!) Au moins une majuscule : true
(Bienvenue!) Tout en majuscule : false
```

*Les deux méthodes ne consomment qu'autant d'éléments qu'il en faut pour déterminer la réponse. Si la clôture renvoie **true** pour un élément, **any()** renvoie **true** immédiatement sans extraire d'autres éléments de l'itérateur.*

Comparaison de séquences d'éléments – position, rposition et ExactSizeIterator

La méthode **position()** applique une clôture à chaque élément de l'itérateur pour renvoyer l'indice du premier élément pour lequel la clôture valide le critère de sélection. Elle renvoie **Option** pour l'indice, ou **None** si la clôture n'a jamais renvoyé **true**. L'extraction s'arrête dès que la clôture répond **true**. La méthode **rposition()** est son complément puisqu'elle travaille à partir de la fin de la séquence.

Recherche de position d'un caractère

```
fn main() {
  let message = "Bienvenue";

  println!("Position de la lettre 'e' : {:?}", message.chars().position(|c| c=='e'));
  println!("'e' en partant de la fin : {:?}", message.bytes().rposition(|c| c==b'e'));
  println!("Position de la lettre 'z' : {:?}", message.chars().position(|c| c=='z'));
}
```

Résultat

```
Position de la lettre 'e' : Some(2)
'e' en partant de la fin : Some(8)
Position de la lettre 'z' : None
```

*La méthode **rposition()** demande que l'itérateur soit réversible, puisqu'elle a besoin de commencer par le côté droit de la séquence. L'itérateur doit en outre être à taille exacte pour que les indices coïncides avec ceux de **position()**, en comptant **0** pour l'indice du premier élément à gauche.*

*Les itérateurs ne savent pas toujours combien d'éléments ils vont produire et la première évaluation de l'exemple ci-dessus utilise un itérateur **chars** sur **&str** qui pose problème puisque **UTF-8** est un encodage de caractères à largeur variable.*

*Vous ne pouvez donc pas utiliser **rposition()** sur une chaîne. En revanche, le balayage d'un tableau d'octets est à longueur prévisible et peut donc implémenter **ExactSizeExtrator**.*

Comparaison de séquences d'éléments – fold

Cette méthode **fold()** constitue un outil à usage général pour accumuler un résultat par rapport à toute une séquence d'éléments d'un itérateur. Vous lui fournissez une valeur initiale qui incarne l'accumulateur et une clôture ; la méthode applique la clôture à cet accumulateur avec le prochain élément de l'itérateur et la valeur résultant de la clôture devient le nouvel accumulateur qui est retransmis à la clôture au prochain tour.

*Au final, l'accumulateur contient ce qu'à renvoyé **fold()**. Dans le cas où la séquence est vide, la méthode renvoie l'accumulateur de départ. De nombreuses autres méthodes qui consomment les valeurs d'un itérateur peuvent être reformulées comme des applications de **fold()**.*

Multi-évaluation

```
use std::cmp::max;

fn main() {
  let int = [5, 6, 7, 8, 9, 10, 11];

  println!("{:?} Nombre d'éléments : {}", int, int.iter().fold(0, |n, _| n+1));
  println!("{:?} Somme = {}", int, int.iter().fold(0, |n, i| n+i));
  println!("{:?} Produit = {}", int, int.iter().fold(1, |n, i| n*i));
  println!("{:?} Valeur maximale : {}", int, int.iter().fold(i32::min_value(), |m, &i| max(m, i)));
}
```

Résultat

```
[5, 6, 7, 8, 9, 10, 11] Nombre d'éléments : 7
[5, 6, 7, 8, 9, 10, 11] Somme = 56
[5, 6, 7, 8, 9, 10, 11] Produit = 1663200
[5, 6, 7, 8, 9, 10, 11] Valeur maximale : 11
```

*Les valeurs de l'accumulateur sont transférées vers et ramenées depuis la clôture. Vous pouvez donc utiliser **fold()** avec des types d'accumulateurs non copiant.*

Concaténation de chaînes

```
fn main() {
  let mots = ["Bienvenue ", "tout ", "le monde"];
  let phrase = mots.iter().fold(String::new(), |mut s, &m| {s.push_str(m); s});

  println!("{:?} => '{}'", mots, phrase);
}
```

Résultat

```
["Bienvenue ", "tout ", "le monde"] => 'Bienvenue tout le monde'
```

Comparaison de séquences d'éléments – nth

Cette méthode **nth()** permet d'accéder au **n-ième** élément. Vous lui fournissez un indice **n**, ce qui la force à feuilleter l'itérateur pour ne renvoyer que l'élément correspondant à l'indice ou bien **None** si la séquence s'est arrêtée en cours de route. Autrement dit l'appel « **.nth(0)** » équivaut à « **.next()** ».

La méthode ne prend pas la possession de l'itérateur comme le fait un adaptateur, et vous l'appeler à répétition autant de fois que vous le désirez.

Récupérer un élément par indice

```
fn main() {
  let mut carres = (0..10).map(|n| n*n);
  println!("{:?}", carres.nth(4));
  println!("{:?}", carres.nth(0));
  for x in &mut carres { print!("{} ", x)}
}
```

Résultat

```
Some(16)
Some(25)
36 49 64 81
```

Comparaison de séquences d'éléments – last

Cette méthode s'apparente à la précédente. Elle balaye l'itérateur en délivrant **None**, pour ne renvoyer que le dernier élément. Si l'itérateur est vide, **last()** renvoie directement **None**.

Récupérer le dernier élément

```
fn main() {
  let carres = (0..10).map(|n| n*n);
  println!("{:?}", carres.last());
}
```

Résultat

```
Some(81)
```

*L'exemple précédent consomme la totalité de l'itérateur à partir du début, même s'il est réversible. Dans ce dernier cas, si vous ne désirez pas consommer les éléments initiaux, il est plus judicieux d'utiliser l'expression suivante **iter().rev().next()**.*

Récupérer le dernier élément

```
fn main() {
  let carres = (0..10).map(|n| n*n);
  println!("{:?}", carres.rev().next());
}
```

Résultat

```
Some(81)
```

Comparaison de séquences d'éléments – find

Cette méthode **find()** extrait des éléments en renvoyant le premier pour lequel la clôture renvoie la valeur **true** ou bien **None** si nous arrivons en fin de séquence avant d'avoir trouvé.

Trouver une ville de plus de 700 000 habitants

```
use std::collections::HashMap;

fn main() {
  let mut villes= HashMap::new();
  villes.insert("Lyon", 518_635);
  villes.insert("Marseille", 873_716);
  villes.insert("Paris", 2_175_601);
  println!("Ville de plus de 700 000 d'habitants {:?}",
            villes.iter().find(|&(_ville, &habitants)| habitants > 700_000).unwrap());
}
```

Résultat

```
Ville de plus de 700 000 d'habitants ("Marseille", 873716)
```

Construction d'une collection avec collect et FromIterator

Nous nous sommes souvent servi dans les différents exemples de cette étude de la méthode **collect()** pour construire automatiquement des vecteurs avec les éléments d'un itérateur. Pourtant **collect()** n'est pas spécialisée pour les vecteurs uniquement et permet de construire n'importe quel genre de collection à partir de la librairie standard de **Rust**, à la seule condition que l'itérateur produise un type d'élément compatible.

*Lorsque vous déclarer des variables qui vont être associées à la mise en collection au travers de cette méthode, vous pouvez alors utiliser l'inférence de type en spécifiant explicitement le type dans la phase déclarative. La deuxième approche consiste à indiquer au moment de l'appel de la méthode **collect()** le type de collection souhaité, comme pour les génériques.*

Générer différents types de collection

```
fn main() {
  let lettres = ['R', 'u', 's', 't'];
  let mot = lettres.iter().collect::<String>();
  let vecteur = lettres.iter().collect::<Vec<_>>();
   println!("{:?} => {} => {:?}", lettres, mot, vecteur);
}
```

Résultat

```
['R', 'u', 's', 't'] => Rust => ['R', 'u', 's', 't']
```

Construction d'une collection avec le trait Extend

Pour un type, dès que vous implémentez le trait **std::iter::Extend**, sa méthode **extend()** dote la collection cible d'une séquence d'éléments itérables. Toutes les collections standard implémentent ce trait, y compris **String**. En revanche, les tableaux et les tranches possèdent une longueur fixe et ne l'implémente donc pas.

*Elles ressemblent beaucoup à celle de **FromIterator** qui sert à créer une nouvelle collection alors que **Extend** étend une collection existante. D'ailleurs, plusieurs implémentations de **FromIterator** dans la librairie standard se limitent à créer une collection vide pour appeler ensuite **extend()** afin de la remplir.*

Générer différents types de collection

```
fn main() {
  let mut entiers : Vec<i32> = (0..5).map(|n| 1 << n).collect();
  entiers.extend(&[32, 64, 128, 256]);
  print!("{:?}", entiers);
}
```

Résultat

```
[1, 2, 4, 8, 16, 32, 64, 128, 256]
```

Construction d'une collection - partition

Cette méthode partitionne les éléments d'un itérateur pour obtenir deux collections en servant d'une clôture pour décider. Comme avec la méthode **collect()**, **partition()** permet de produire n'importe quelle collection, mais les deux entrées doivent être du même type. Vous devez également indiquer le type renvoyé.

*Dans l'exemple suivant, nous indiquons clairement le type des deux collections d'entrée, mais nous laissons l'inférence de type choisir les paramètres de types appropriés pour l'appel à **partition()**. L'astuce dans ce code, c'est que le nom des vivants commencent par une lettre impaire, et nous nous servons de ce critère pour séparer les deux collections.*

Partitionner une collection en deux sous-parties

```
fn main() {
  let choses = ["domino", "mangue", "nouille", "girafe", "myrtille"];
  let (vivants, non_vivants) : (Vec<&str>, _) = choses.iter().partition(|nom| nom.as_bytes()[0] & 1 != 0);
  print!("{:?} {:?}", vivants, non_vivants);
}
```

Résultat

```
["mangue", "girafe", "myrtille"] ["domino", "nouille"]
```

Implémentation de vos propres itérateurs

En implémentant les deux traits **IntoIterator** et **Iterator** pour vos propres types, vous disposez d'office de tous les adaptateurs et de tous les consommateurs présentés lors de cette étude. Voyons comment élaborer notre propre itérateur.

*Je vous propose de créer la structure **Parite** qui va nous permettre d'itérer des nombres pairs avec deux champs, **debut** et **fin** qui précisent les limites de cette itération. Nous devons effectivement maintenir deux informations d'état pour balayer la plage de valeurs. Nous devons au moins avoir un champ **debut** qui spécifie la prochaine valeur pour l'occurrence suivante et **fin** pour donner la limite et arrêter le processus.*

Occurrences de nombres pairs

```
struct Parite {
  debut: u32,
  fin: u32
}

impl Iterator for Parite {
  type Item = u32;
  fn next(&mut self) -> Option<u32> {
    if self.debut%2 != 0 { self.debut += 1 }
    if self.debut > self.fin { return None }
    let suivant = Some(self.debut);
    self.debut += 2;
    suivant
  }
}

fn main() {
  for p in (Parite {debut: 3, fin: 18}) {
    print!("{}  ", p);
  }
}
```

Résultat

```
4  6  8  10  12  14  16  18
```

*Pour résoudre le trait **Iterator**, nous devons redéfinir la méthode **next()**. Notre itérateur produit des éléments de type **u32** qui est donc le type de **Item**. En fin d'itération, **next()** renvoie **None**. Dans les autres cas, il produit la prochaine valeur puis met à jour son état pour être prêt pour l'occurrence suivante. Si le premier nombre est impair, nous ne le prenons pas en compte et nous incrémentons sa valeur pour être sûr de commencer par un nombre pair.*