2_Functions_fr

May 2, 2023

1 Fonctions

La définition des fonctions en Rust commence par le mot clé `fn`; puis des parenthèses après le nom de la fonction. Les accolades ({ }) indique au compilateur le corp de la fonction.

En Rust, la convention est d'utiliser le style *snake_case* pour nommer les fonctions et les variables. En *snake_case*, toutes les lettres sont en minuscules et des underscore (_) séparent les mots. Ci-dessous se trouve un exemple de programme avec un exemple de définition de fonction.

[ ]:
```
fn main() {
    println!("Hello, world!");

    another_function();
}

fn another_function() {
    println!("Another function.");
}
main();
```

Il est possible d'appeler n'importe quelle fonction définie par son nom et en ajoutant des parenthèses. Notez que nous avons défini la fonction `another_function` après la fonction `main` dans le code. Nous aurions pu la définir avant également, il n'y a pas de différence en Rust.

2 Paramètres des fonction

Les fonctions peuvent également recevoir des paramètres. Dans la signature des fonctions, vous devez déclarer le type de chacun d'eux. Ceci est un choix délibéré dans le design de Rust : le fait d'imposer une annotation de type dans la définition d'une fonction signifie que le compilateur n'a jamais besoin de le typage soit indiqué ailleurs dans le code.

[ ]:
```
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("La valeur de x est: {}", x);
}
```

```
main();
```

[ ]:
```
fn main() {
    another_function(5, 6);
}

fn another_function(x: i32, y: i32) {
    println!("La valeur de x est: {}", x);
    println!("La valeur de y est: {}", y);
}
main();
```

3 Instructions et expressions

Le corps des fonctions est composé d'une série d'instructions, se terminant éventuellement par une expression. Les **instructions** (statements) réalisent des opérations et ne *retourne pas de valeur*. Les **expression** s'évaluent en une valeur qui peut être retournée.

[ ]:
```
fn main() {    // La définition de la fonction est une instruction
    let y = 6; // une autre instruction
}
main();
```

3.1 Instructions

Les instructions ne retournent pas de valeur. En conséquence, il n'est pas possible n'assigner le retour d'une instruction `let` à une autre variable. Cette manière de structurer le langage est différente d'autres comme le C ou Ruby où l'assignation retourne la valeur assignée à la variable.

Example causant une erreur :

[ ]:
```
fn main() {
    let x = (let y = 6); // erreur
}
main();
```

3.2 Expressions

Les expressions s'évaluent en une valeur. Les opérations mathématiques, l'appel à une fonction ou à une macro sont des exemples d'expressions. Notez que le block que nous utilisons pour créer un nouveau scope, **{ }** est aussi une expression.

[ ]:
```
fn main() {
    let x = 5;

    let y = {  // ce block est une expression qui s'évalue à 4
        let x = 3;
```

```
        x + 1 // la dernière expression est la valeur pour le block
    };

    println!("The value of y is: {}", y);
}
main();
```

Notez la ligne `x + 1` sans point-virgule (;) à la fin, différente de toutes les autres vue jusque ici. Les expressions ne se terminent jamais par un point-virgule. Si vous ajouter un point-virgule à une expression, cela la transforme en une instruction qui ne retourne alors pas de valeur.

4 Fonctions retournant une valeur

Les fonctions peuvent retourner une valeur au code appelant. Il n'est pas nécessaire de nommer cette valeur, mais il faut déclarer son type en utilisant une flèche (`->`). En Rust, la valeur de retour de la fonction est en fait la valeur à laquelle s'évalue le block du corps de cette fonction, c'est à dire la dernière expression. Il est cependant possible de retourner plus tôt, avec le mot clé `return` et en spécifiant la valeur. Mais la plupart des fonctions retourne leur dernière expression implicitement.

[ ]:
```
fn five() -> i32 {  // la fonction retourne un i32
    5    // ceci est une expression (pas de point-virgule)
}

fn main() {
    let x = five();   // équivalent à: let x = 5;

    println!("La valeur de x est: {}", x);
}
main();
```

[ ]:
```
fn main() {
    let x = plus_one(5);

    println!("La valeur de x est: {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1;  // le point-virgule ici est une erreur
}
main();
```

5 Exercises

Exercise 1: Ecrire une fonction qui détermine si deux valeurs flottantes sont égales, considérant une tolérance `EPSILON`.

Valeurs de test:

- (0.00009384921 , 0.00009384922)
- (0.000093849221 , 0.000093849222)

[ ]:
```rust
/// La valeur de tolérance: 1e-11
const EPSILON: f64 = 1e-11;

/// TO DO (1) Implémenter une fonction `eq`

fn main() {
    let x = 0.000093849221;
    let y = 0.000093849222;
    println!("The numbers are equal: {}", eq(x,y));
}
main();
```

Solution:

[ ]:
```rust
/// The standard value for epsilon 1e-11
const EPSILON: f64 = 1e-11;

/// Determines whether x and y are equals with a tolerance of eps
fn eq(x: f64, y: f64) -> bool {
    (x - y).abs() < EPSILON
}

fn main() {
    let x = 0.000093849221;
    let y = 0.000093849222;
    println!("The numbers are equal: {}", eq(x,y));
}

main();
```