UNIVERSIDADE ESTADUAL DE MARINGÁ
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA

CIÊNCIA DA COMPUTAÇÃO

PARADIGMA DE PROGRAMAÇÃO IMPERATIVA E ORIENTADA A OBJETOS

Avaliação comparativa de duas linguagens de programação

Professor: Marco A L Barbosa

Discentes

R.A.	Nome
110752	Felipe Diniz Tomás
105408	Igor Pícolo Carreira

Facilidade de leitura e escrita

A linguagem Python facilitou muito na escrita do programa, já que é uma linguagem alto nível, simples e flexível, onde não há a necessidade de declaração de variáveis e tipos. No entanto, essa falta de tipagem pode dificultar a leitura do código, já que o programador não tem a visão de que tipo de dado uma variável é.

Em Rust, que é uma linguagem de baixo nível com conceitos modernos ao uso de memória, existem diversos tipos, como por exemplo dois tipos de *Strings*, três tipos de *Array*, além dos *Enum* que permitem a criação de tipos definidos pelo seus valores (*variants*). Todas as variáveis exigem uma declaração de tipo, além de que, por padrão todas são imutáveis, sendo necessário especificar caso queira torná-las mutáveis. Apesar do código ficar mais claro para ler, escrever um programa em Rust é mais demorado, além de suas peculiaridades (como conceito de *ownership*, gerenciamento de memória), sua sintaxe é mais complexa que em uma linguagem alto nível.

A delimitação dos blocos de código em Python é feita apenas com indentação, o que facilita a leitura, pois o programador deve realizar a indentação corretamente. Porém, se não o fizer, o código poderá conter diversos bugs, especialmente em iterações. Além disso o código inteiro não precisa seguir a mesma indentação, o que pode torná-lo confuso, visto que em uma parte dele pode-se usar, por exemplo, espaço para indentação e em outra *tabs*.

A delimitação dos blocos de códigos em Rust é feita por chaves, semelhante a linguagem C. Os blocos de códigos ficam bem definidos, e é de fácil visualização onde cada bloco começa e termina. No entanto, o código ficará maior, devido a esses delimitadores.

O código Python pode ser escrito em um simples editor de texto, ou seja, é bem acessível e ideal para algoritmos de baixa complexidade, já em uso em grande escala, devido a dependência de bibliotecas externas, além da falta de tipagem e identificador de blocos por indentação, não é a melhor opção, já que sua leitura é confusa.

Rust em contrapartida, é ideal para sistemas complexos e seguros quanto ao gerenciamento de memória, além de sua manutenibilidade ser muito boa. No entanto a produtividade em Rust pode ser considerada menor, já que para escrevê-lo e compilá-lo é um processo demorado, devido às suas peculiaridades.

Confiabilidade

Como todas as variáveis em Python são ponteiros, isso significa que até mesmo variáveis que contém somente um inteiro, serão tratadas como referências. A segurança deste tipo de tratamento é fragilizada, já que se o programador necessita de uma variável de um tipo específico durante todo o programa ele não terá esta certeza, porque seu tipo pode mudar. Além disso as não tipagens das variáveis podem gerar erros em comparações, forçando a utilização de parses.

Um dos fatores mais atrativos da linguagem Rust é a sua confiabilidade, já que ela preza pela segurança no gerenciamento de memória. Um conceito novo de gerenciamento utilizado pelo Rust, é o *Ownership rules,* onde a memória é gerenciada através da "posse" e um conjunto de regras, verificado pelo compilador em tempo de compilação. *Ownership*, é a maneira como o Rust administra um heap data.

Cada valor em Rust tem uma variável chamada de "*Owner*" daquele valor. O "Owner" apenas é válido dentro do escopo. A memória que o valor precisa para estar alocado no *heap*, é automaticamente retornada ao sistema operacional quando a variável que possui esse valor sai do escopo. Assim o Rust chama uma função especial chamada "*drop*" automaticamente, e limpa o *heap* de memória daquela variável.

Quando você atribui uma variável antiga x, com dados alocados no *heap*, para uma nova variável y, o Rust retornará "*x moved to y i.e invalidates x for memory safety*". Dessa forma o Rust só pode fazer uma cópia "superficial" de x, sem copiar os dados reais do heap. Para fazer uma cópia "profunda" é necessário utilizar o método "*clone*", assim haverá uma cópia completa dos dados de x para y.

Para tipos de datas que têm tamanhos conhecidos em tempo de compilação, como inteiros, você não precisa cloná-los, basta copiá-los (“Copy” trait) do heap. Mesmo depois da chamada da função, essas variáveis ainda podem ser usadas, assim como os valores que retornam na transferência de “Ownership”.

O gerenciamento de memória em Python é feito através de *garbege collection,* que é um mecanismo responsável pela desalocação da memória uma vez que ela não é mais utilizada. Seu funcionamento ocorre através de contadores, que possuem a informação de quantas vezes um objeto é utilizado. Assim que esses contadores forem zerados, significa que o objeto já não é mais necessário, então o GC remove a referência do objeto da memória liberando espaço para outros recursos.

A linguagem python é interpretada, ou seja, não há uma verificação sintática que acontece no processo de compilação, deixando o código mais vulnerável a erros em tempo de execução.

Em contraste, caso seu código Rust compile, a chance de erros em tempo de execução é muito baixa, afinal muitos erros de memória são tratados em tempo de compilação, basicamente os erros possíveis que sobrariam seriam na lógica. Em consequência disso, a manutenibilidade de Rust é muito boa, podendo receber novas implementações sem grandes erros.

Custo de execução

O Python tem um maior nível de linguagem em relação ao Rust, isso significa que ele abstrai do desenvolvedor detalhes como gerenciamento de memória , ponteiros, etc, fazendo com que escrever programas fique mais próximo à maneira do ser humano pensar do que do próprio computador. Por este motivo, o Rust, que é uma linguagem de baixo nível, tem uma vantagem sobre o Python no tempo de execução, outro motivo é que Rust é uma linguagem que é compilada enquanto Python é interpretada.

O código fonte do Rust é executado diretamente pelo sistema operacional ou pelo processador, após ser traduzido diretamente em *linguagem de máquina* por meio de um processo chamado compilação,

diminuindo seu tempo de execução. Já o código fonte do Python é executado por um interpretador, ou seja, uma maquina virtual da linguagem que a interpreta e em seguida a traduz para *linguagem de máquina* e só assim é executado pelo sistema operacional ou processador, aumentando seu tempo de execução.

O gerenciamento de memória *Ownership*, explicado no tópico anterior, é uma característica utilizada em Rust que também influencia na questão de custo, pois o Rust consegue segurança e velocidade através de muitas “zero-cost abstractions” (ou, abstrações de custo zero), isto significa que o Rust implementa as suas abstrações com o menor custo possível. No sistema de *ownership* todas as análises são executadas durante a compilação. Assim, não existem custos de performance durante a execução.

Em linguagens como Pyhton, com *garbege collection (GC)*, o gerenciamento de memória geralmente é mais lento, devido a necessidade de *escanear* o heap, além de que os objetos precisam conter informações adicionais. Ainda sim, existem diversas implementações de GC que funcionam mais rapidamente, no entanto a maioria possuiu um custo de performance que raramente é melhorado.

Em geral a velocidade de Rust é semelhante a de linguagens como C, trazendo técnicas mais modernas e amigáveis para otimização e rapidez. Python é uma linguagem voltada a simplicidade na escrita, dinâmica, então seu custo de tempo e memória é naturalmente maior.