# Programação Funcional Para leigos

Usar as técnicas de programação funcional pode torná-lo muito eficiente ao resolver certos problemas ou quando precisar usar por completo as técnicas de multiprocessamento para obter o benefício máximo de cada processador (ou núcleo). O paradigma de programação funcional suporta o conceito de linguagens puras (como o Haskell, que usa apenas técnicas funcionais) e impuras (como o Python, que suporta vários paradigmas da programação). E mais, a programação funcional tem uma boa base matemática: cada linha de código é, na verdade, uma expressão, não um procedimento, como em muitos outros paradigmas. Esta folha de cola o ajuda a entender as diferenças dos outros paradigmas para melhorar sua experiência com a programação funcional.

## OS QUATRO PARADIGMAS COMUNS DO PYTHON

O Python é uma linguagem de programação funcional impura, significando que ele suporta outros paradigmas de programação. Dependendo da perspectiva, o suporte dos outros paradigmas pode ser uma vantagem ou não. Usar uma linguagem impura significa que você não aproveitará totalmente o uso das técnicas da programação funcional e que algumas técnicas podem nem mesmo estar disponíveis. Porém uma linguagem impura também permite resolver alguns problemas de um modo compreensível, que poderia parecer um pouco complicado usando as técnicas da programação funcional. Com esses prós e contras em mente, veja os quatro paradigmas essenciais do Python, também chamados de estilos de codificação:

- **Funcional:** Toda instrução é uma equação matemática. Esse estilo é bem adequado para usar nas atividades de processamento paralelo. Estudiosos e cientistas de dados tendem a usar esse estilo de codificação com regularidade. Porém nada o impede de usá-lo, mesmo que não faça parte de um desses grupos.
- **Imperativo:** As computações ocorrem como alterações no estado do programa. Esse estilo é mais usado para manipular as estruturas de dados. Todos os cientistas contam com ele porque demonstra os processos com muita clareza.
- **Orientado a objetos:** É o estilo comumente usado com outras linguagens para simplificar o ambiente de codificação usando objetos para modelar o mundo real. O Python não o implementa totalmente por não suportar recursos como o encapsulamento convencional, mas você ainda pode usar essa abordagem em grande parte. É o estilo que a maioria dos desenvolvedores usa, mas outros grupos podem usá-lo ao criar aplicativos mais complicados.
- **Procedural:** A maioria das pessoas começa aprendendo uma linguagem com o código procedural, em que as tarefas avançam uma etapa por vez. Esse estilo é mais usado para iteração, sequenciamento, seleção e modularização. É a forma mais simples de codificação que se pode usar. Pessoas que não são da área adoram esse estilo porque é o modo menos complicado de realizar tarefas experimentais mais simples.

# Programação Funcional Para leigos

## RECURSOS ESSENCIAIS DO PARADIGMA DE PROGRAMAÇÃO FUNCIONAL

O paradigma de programação funcional não tem uma implementação, mas descreve os recursos que uma implementação da linguagem teria. A seguinte lista dá uma ideia de quais recursos procurar em uma linguagem que suporta esse paradigma. Quanto mais recursos uma linguagem suporta, mais pura é sua implementação.

- **Suporte do cálculo lambda:** A base da programação funcional é o cálculo lambda, que é uma abstração matemática. Sempre que você cria e usa uma função lambda, provavelmente está usando técnicas da programação funcional (de um modo impuro, pelo menos).
- **Funções de primeira classe e funções de alta ordem:** Essas funções permitem fornecer uma função como entrada, como seria ao usar uma função de alta ordem no cálculo.
- **Funções puras:** Uma função pura não tem efeitos colaterais. Ao trabalhar com uma função pura, é possível:
    - Remover a função, caso nenhuma outra conte com sua saída.
    - Obter os mesmos resultados sempre que chamar a função com certo conjunto de entradas.
    - Inverter a ordem das chamadas para diferentes funções sem nenhuma alteração na funcionalidade do aplicativo.
    - Processar as chamadas da função em paralelo sem nenhuma consequência.
    - Avaliar as chamadas da função em qualquer ordem, supondo que a linguagem inteira não permita efeitos colaterais.
- **Recursão:** As implementações da linguagem funcional contam com a recursão para implementar o loop. Em geral, a recursão trabalha de modo diferente nas linguagens funcionais porque não ocorre nenhuma mudança no estado do aplicativo.
- **Transparência referencial:** O valor de uma variável nunca muda em uma implementação da linguagem funcional porque essas linguagens não têm um operador de atribuição.

# Programação Funcional Para leigos

## PIONEIROS DA AUTOMAÇÃO MATEMÁTICA

A matemática nem sempre foi automática, o que pode ser uma surpresa no mundo moderno. Pessoas como Alonzo Church, Haskell Curry, Kurt Gödel, Emil Post e Alan Turing tiveram que criar uma definição para os algoritmos. Antes de alguém conseguir construir um computador, uma pessoa teve que propor uma definição do que significa calcular, o que parece óbvio agora, mas não naquela época. A seguinte lista informa o motivo de cada uma das pessoas listadas propor a própria definição do que significa calcular, e cada uma das definições tem seu lugar no mundo moderno.

- **Alonzo Church: Cálculo λ (o assunto do livro *Programação Funcional Para Leigos*).**
- **Haskell Curry: Lógica combinatória.**
- **Kurt Gödel: Funções recursivas μ (veja também "Funções Recursivas μ" para obter detalhes).**
- **Emil Post: Sistema canônico de Post, também chamado de sistema reescrito.**
- **Alan Turing: Máquinas de Turing.**

## FUNÇÕES DE LISTA MAIS USADAS

Os desenvolvedores tendem a considerar listas e arrays como tendo o mesmo tipo de estrutura, mas eles são diferentes. A principal diferença está em como armazenam os dados. Um array sempre os armazena em locais da memória sequencial, o que dá ao array tempos de acesso mais rápidos em algumas situações, mas também deixa lenta a criação deles. E mais, como um array deve aparecer na memória sequencial, ele geralmente é difícil de atualizar, e algumas linguagens não permitem que eles sejam modificados do mesmo modo que as listas.

Uma lista armazena dados usando uma estrutura de dados vinculada, e seus elementos consistem no valor do dado e um ou dois ponteiros. As listas têm mais memória porque você agora deve alocar memória aos ponteiros para o próximo local do dado (e o local anterior também nas listas com dois vínculos, o tipo usado pela maioria das linguagens atualmente). Normalmente, é mais rápido criar listas e adicionar dados a elas, por causa do mecanismo de vinculação, mas elas têm um acesso de leitura mais lento que os arrays.

Porém interagir com listas e arrays é parecido. Com isso em mente, a seguinte relação mostra as funções de lista mais usadas do Haskell:

- **`head a`: Mostra o valor no índice 0, que é `1` neste caso.**
- **`tail a`: Mostra o resto da lista após o índice 0, que é `[2,3,4,5,6]`.**
- **`init a`: Mostra tudo, exceto o último elemento da lista, que é `[1,2,3,4,5]`.**

- `last a`: Mostra apenas o último elemento na lista, que é `6`.
- `take 3 a`: Requer o número de elementos que você deseja ver como entrada e mostra esse número no início da lista, que é `[1,2,3]`.
- `drop 3 a`: Requer o número de elementos que você não quer ver como entrada e mostra o resto da lista após retirar os elementos requeridos, que é `[4,5,6]`.
- `length a`: Retorna o número de elementos na lista, que é `6`.
- `null a`: Determina se a lista está vazia e retorna um resultado booleano, que é `False`.
- `minimum a`: Determina o menor elemento de uma lista e retorna-o, que é `1`.
- `maximum a`: Determina o maior elemento de uma lista e retorna-o, que é `6`.
- `sum a`: Soma os números da lista, que é `21`.
- `product a`: Multiplica os números da lista, que é `720`.

O Python não fornece listas, mas usa arrays. Porém, como no Haskell, você pode fazer certas operações com arrays que parecem muito com os equivalentes da lista. Veja uma lista parecida das funções de array mais usadas do Python:

- `a[0]`: Obtém o início da lista, que é `1` neste caso.
- `a[1:]`: Obtém o final da lista, que é `[2,3,4,5,6]` no caso.
- `a[:-1]`: Obtém tudo, exceto o último elemento, que é `[1,2,3,4,5]`.
- `a[-1:]`: Obtém apenas o último elemento, que é `6`.
- `a[:-3]`: Faz o mesmo que `take 3 a` no Haskell.
- `a[-3:]`: Faz o mesmo que `drop 3 a` no Haskell.
- `len(a)`: Retorna o número de elementos em uma lista.
- `not a`: Verifica uma lista vazia. Essa verificação é um resultado diferente de `a is None`, que procura um valor real nulo — a não sendo definido.
- `min(a)`: Retorna o menor elemento da lista.
- `max(a)`: Retorna o maior elemento da lista.
- `sum(a)`: Soma o número da lista.