

Universidade de Araraquara

Programação Funcional

Adaptado de Simão Melo de Sousa

Programação Funcional

Sobre a elegância e simplicidade em programação

*Beauty is more important in computing than anywhere else in technology because software is so complicated. **Beauty is the ultimate defence against complexity.***

— David Gelernter

*Beleza é mais importante em computação do que em qualquer outro lugar por que software é muito complicado. **Beleza é a ultima defesa contra complexidade***

*Complexity has nothing to do with intelligence, **simplicity does.***

— Larry Bossidy

*Complexidade não tem nada a ver com inteligencia, **simplicidade tem.***

Sobre a elegância e simplicidade em programação

*When I am working on a problem, I never think about beauty. I think only how to solve the problem. But when I have finished, **if the solution is not beautiful, I know it is wrong.***

— R. Buckminster Fuller

*Quando estou trabalhando em um problema eu nunca penso em beleza. Eu penso em como resolver o problema. **Mas quando eu termino, se a solução não é bela, eu sei que está errada.***

Simplicity is prerequisite for ***reliability***.

— Edsger W. Dijkstra

Simplicidade é um pré-requisito para ***confiabilidade***.

Sobre a elegância e simplicidade em programação

Sometimes, ***the elegant implementation is a function.***
Not a method, not a class, not a framework. Just a function.
— John Carmack

As vezes, ***a implementação elegante e uma função.*** Não
é um método nem uma classe. Apenas uma função.

estas aulas visam introduzir a programação funcional que em
muito está alinhada com estes princípios aqui expostos.

mas....

nunca se esqueça:

não há linguagens ótimas, só há bons programadores!

dado um desafio, um bom programador **sabe escolher** a melhor ferramenta (linguagem de programação, API, etc.) e a melhor resolução (algoritmos, estrutura de dados, etc.)

as linguagens de programação têm os seus domínios de excelência

História e motivação

calcular e programar

Computacao: o estudo de processos algoritmicos que descrevem e transformam informacao. A questao fundamental é "o que pode ser (eficientemente) automatizado?" — 1989 ACM report on Computing as a Discipline

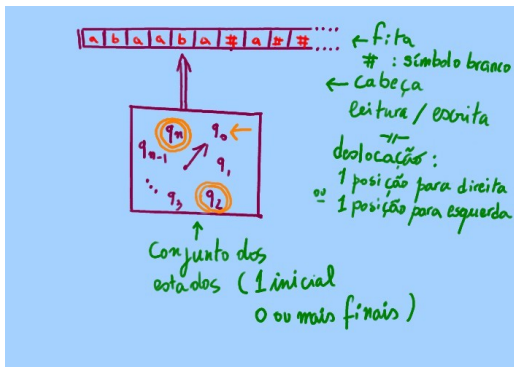
as componentes básicas da *computação*

- um **algoritmo** que descreve o processo de resolução do problema por resolver
- um **programa** que descreve o algoritmo na forma das transformações que desejamos aplicar à informação por processar tendo em conta a máquina subjacente
- uma **máquina** que seja capaz de executar tais transformações

há **muitas** máquinas e **muitas formas** possíveis de escrever programas algumas até foram inventadas antes do primeiro computador!

Turing

inventou o conceito de Máquina de Turing



estas máquinas dão um fundamento teórico às arquiteturas de computadores e à programação imperativa

a fita tem o papel da **memória** (onde se encontram dados e programas) o automato do **processador**

máquinas de Turing e o paradigma imperativo

um programa imperativo **lê, escreve, executa operações** e **toma decisões** com base no conteúdo de células de memória que contêm informação sobre as variáveis do programa, como `x,n,res` no seguinte programa java (que calcula o factorial)

```
public class Fatorial
{
    public static int calcular(int n)
    {
        int res = 1;
        for (int c = 1; c <= n; c++) res = res * c;
        return res;
    }
}
```

máquinas de Turing e o paradigma imperativo

um programa imperativo **lê, escreve, executa operações** e **toma decisões** com base no conteúdo de células de memória que contêm informação sobre as variáveis do programa, como `x,n,res` no seguinte programa java (que calcula o factorial)

```
public class Fatorial
{
    public static int calcular(int n)
    {
        if (n > 0)
            return n * calcular(n - 1);
        return 1;
    }
}
```

tal como Alan Turing, e no mesmo ano de 1936, Alonzo Church (orientador do A. Turing) dá uma definição alternativa de algoritmo e da execução de programas responde pela negativa à questão de D. Hilbert

a sua definição é de natureza bem diferente da das máquinas de Turing, embora se descobre cedo que ambas são equivalentes

A. Church introduz o **cálculo lambda** (λ -calculus)

- Elemento de base: x , variável.
- abstração: $\lambda x.M$, função anônima de um parâmetro formal, x e de corpo M
- Aplicação: $(M N)$, a função M aplicada ao parâmetro N

Computação: a regra da redução β : $(\lambda x.M)N \rightarrow_{\beta} M[x := N]$

programa: um termo do cálculo lambda

execução: **basta a aplicação** da regra β (nada de hardware complexo)



o cálculo lambda e a programação funcional

num **programa funcional**, **definimos funções** (possivelmente recursivas),
elas são compostas e aplicadas para **calcular** os **resultados** esperados

```
let rec fact =  
  function n → if n=0 then 1 else n * (fact (n-1))
```

numa linguagem de programação funcional pura, as funções são cidadãs de primeira classe, como qualquer outro valor (como os inteiros, por exemplo) podem

- lhes ser atribuído um nome (ou não)
 - ser avaliadas
 - ser passadas como argumento (de outras funções)
 - ser devolvidas como resultado de funções
 - ser usadas em qualquer local onde se espera uma expressão
- veremos a importância destes fatos

o cálculo lambda e a programação funcional

(ab)usando da notação original de A. Church podemos reescrever o código seguinte

```
function n → if n=0 then 1 else n * (fact (n-1))
```

na forma

$$\lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * (\text{fact } (n - 1))$$

são as agora famosas **lambda expressions** que se introduziram no Java, C#, C++, Python.

são elementos programação funcional que as linguagens agora integram

a tese de Church-Turing

em 1937, A. Turing demonstrou que o calculo lambda e as máquinas de Turing são formalismos com a mesma expressividade: uma função é **computável** por uma máquina de Turing **se e só se** é **computável** no cálculo- λ

a **Tese de Church-Turing** foi então formulada e afirma que uma função **computável** (um algoritmo) em **qualquer formalismo computacional** é também **computável por uma máquina de Turing**

em termos mais leigos:

- todas as linguagens de programação são **computacionalmente equivalentes**
- **qualquer algoritmo** pode ser expresso usando um destes formalismos (que são equivalentes)

mas as linguagens de programação **não nascem todas iguais...**

são computacionalmente equivalentes, mas tem comodidades e expressividade diferentes

a procura das construções programáticas mais expressivas ou mais cômodas é uma luta sem fim que leva a

- diferentes formas de representação dos dados
- diferentes modelos de execução
- diferentes mecanismos de abstração

e muitas outras características desejáveis entram em conta no momento do desenho de novas linguagens de programação

- segurança da execução
- eficiência
- modularidade e capacidade de manutenção
- etc.

dependendo do problema em causa, algumas linguagens de programação serão mais adequadas do que outras

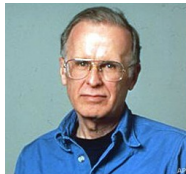
uma opinião do criador da linguagem FORTRAN

... e precursor da programação funcional

Functional programs deal with structured data, ... do not name their arguments, and do not require the complex machinery of procedure declarations ...

Can programming be liberated from the von Neumann style?

— John Backus, Turing lecture, 1978



porque a programação funcional está para ficar?

uma reflexão sobre o ensino introdutório em informática na CMU (<http://www.cs.cmu.edu/~bryant/pubdir/cmu-cs-10-140.pdf>) destaca tendências emergentes das quais:

Precisamos de software mais confiável.

Programas (puramente) funcionais são mais fáceis de serem provados corretos do que os imperativos

Utilizar o poder da computação paralela.

Uma escolha cuidadosa de funções de alta-ordem permitem escrever problemas altamente paralelizáveis.

um exemplo bem conhecido que advém diretamente da programação funcional é o paradigma cloud **MapReduce criado pelo Google para indexar a internet.**

imutabilidade

Um objeto imutável não pode ter nenhuma informação sua alterada após a criação

Exemplos: Strings em Java, Tuplas em Python.

Pense em um classe em Java que possui vários atributos e métodos get e um construtor com argumentos mas nenhum método set.

as suas vantagens:

- facilita depuração de código devido a maior estabilidade do código
- evita problemas de concorrência, ex: várias tarefas alterando o mesmo objeto
- menor preocupação com o estado da execução
- viabiliza caching ou memoization

Objetos imutáveis são predominantes na programação funcional

a onipresença funcional

a programação funcional é longe de ser um exclusivo das linguagens de programação ditas funcionais, **pratica-se em todas as linguagens**

as suas vantagens estão largamente reconhecidas clareza, elegancia, simplicidade e expressividade:

- a **recursividade** é uma componente habitual de qualquer linguagem de programação
- Java 1.5 e C#/NET introduziram os **genéricos** (polimorfismo de tipos, nas linguagens funcionais tipadas)
- Java 1.8 e C++ (v. 11) introduziram as **lambda expressions** (as funções puras, como já as descrevemos) e as construções relacionadas (fluxos)
- etc.

a onipresença funcional

a programação funcional é longe de ser um exclusivo das linguagens de programação ditas funcionais, **pratica-se em todas as linguagens**

recentes linguagens de programação **assimilam** nas suas construções as **boas práticas** que as linguagens de programação funcionais estabeleceram desde sempre
citamos linguagens como **rust**, **swift**, **scala**, **python**, **typescript**, **go**, **closure** (que é funcional por definição) que muito devem à programação funcional

a onipresença funcional

mensagem: quaisquer que sejam as suas linguagens de programação favoritas, perceber os princípios da programação funcional

- **é uma competência transversal, importante e de base**
- **nos torna melhores programadores**


exemplos de linguagens funcionais: **Lisp, Haskell, Racket, Scheme, SML**, etc.

Aspectos da programação funcional em Python

Principais diferenças

-
- **Ausência de estado**: variáveis e argumentos são constantes
 - Imutabilidade
 - **Laços de repetição não existem**: usa-se a recursividade
 - **Funções podem ser passadas como argumentos**

Exemplo 1 - Fatorial

 principal.py ×

2 usages

```
1  def fatorial(n):  
2      if n == 1:  
3          return 1  
4      if n > 1:  
5          return fatorial(n - 1) * n  
6  
7  
8  print(fatorial(5))  
9  
10  
11 |
```

Exe



principal.py

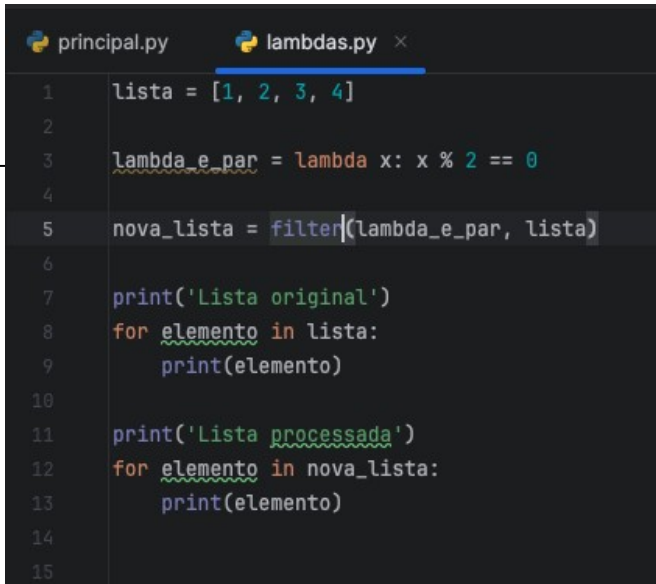


lambdas.py



```
1 lista = [1, 2, 3, 4]
2
3 lambda_3x_mais_1 = lambda x: 3 * x + 1
4
5 nova_lista = map(lambda_3x_mais_1, lista)
6
7 print('Lista original')
8 for elemento in lista:
9     print(elemento)
10
11 print('Lista processada')
12 for elemento in nova_lista:
13     print(elemento)
```

Exemplo 2 – Lambda – Passando função como arg



The image shows a code editor with two tabs: 'principal.py' and 'lambdas.py'. The 'lambdas.py' tab is active and highlighted with a blue underline. The code in 'lambdas.py' is as follows:

```
1 lista = [1, 2, 3, 4]
2
3 lambda_e_par = lambda x: x % 2 == 0
4
5 nova_lista = filter(lambda_e_par, lista)
6
7 print('Lista original')
8 for elemento in lista:
9     print(elemento)
10
11 print('Lista processada')
12 for elemento in nova_lista:
13     print(elemento)
14
15
```

Exercicio 1

-
- Faça um programa em Python que processe uma lista de números inteiros, calculando e exibindo o valor do número elevado ao cubo utilizando uma função lambda.
 - Faça um programa em Python que tenha uma função recursiva que receba uma lista e um número e retorne verdadeiro se o número estiver na lista ou falso caso não esteja
 - Faça o mesmo programa acima usando uma função lambda e filter