

Paradigma funcional



Prof. Dr. Rafael Giusti
rgiusti@icomp.ufam.edu.br

Leitura recomendada

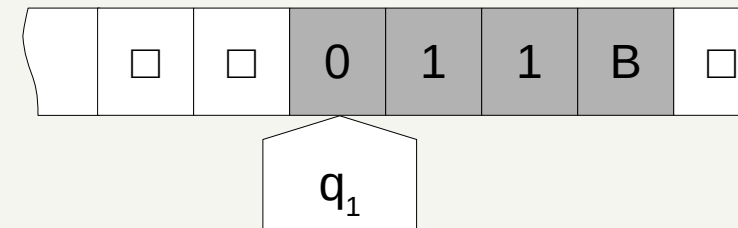
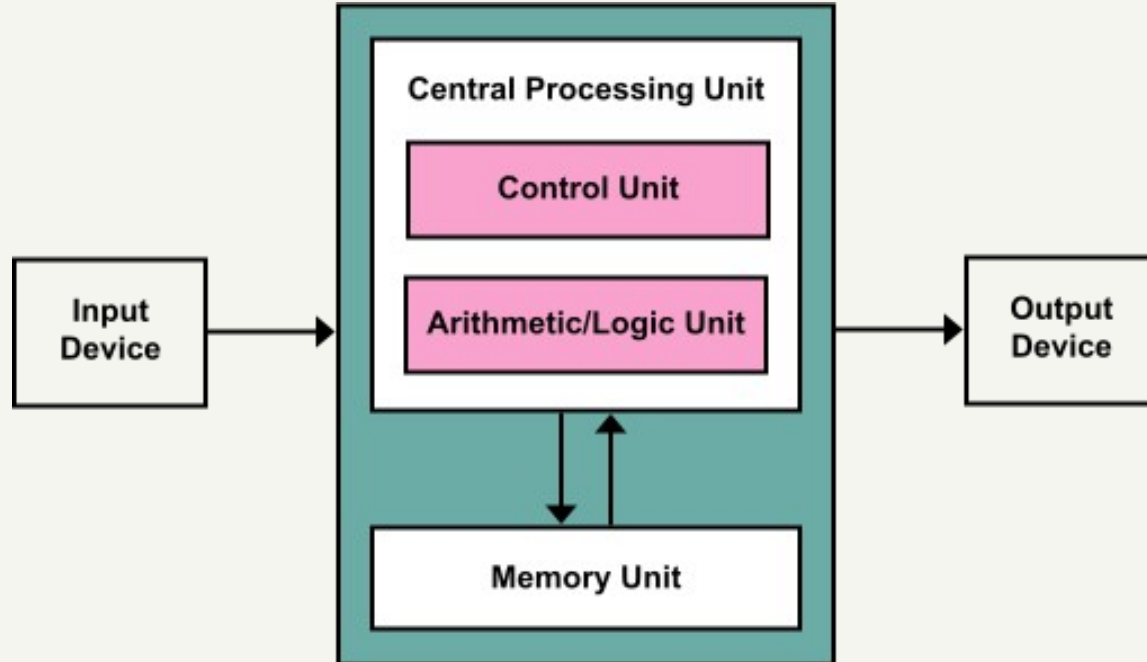
- » SEBESTA. "Conceitos de Linguagens de Programação"
- » Capítulo 7: "Expressões e Sentenças de Atribuição"
 - ~ Em particular, Seção 7.2.2 e subseções
 - ~ Ordem de avaliação dos operandos
 - ~ Efeitos colaterais
 - ~ Transparência referencial
- » Capítulo 15: "Linguagens de Programação Funcional"

Agenda

- » Funcional vs. imperativo
- » Python funcional
- » Funções de ordem mais alta
- » Compreensões de listas
- » Geradores e avaliação preguiçosa

Paradigmas imperativos

- » Linguagens imperativas têm como base
 - » A arquitetura de von Neumann
 - » O modelo da máquina de Turing



Paradigmas imperativos

- » Características de linguagens imperativas
 - » Programas são sequências de **comandos**
 - » Algoritmos descrevem **como** o programa deve ser executado
 - » **Mudança de estado** (no sentido de semântica operacional / denotacional)

Paradigma funcional

- » O paradigma funcional tem como base
 - » Cálculo lambda
 - » Funções matemáticas
- » Características
 - » Programas são coleções de funções e de **aplicações** dessas funções (chamadas)
 - » Algoritmos descrevem **o quê** o programa deve realizar, não como
 - » **Não existe estado**

Paradigma funcional

» Fatorial imperativo vs. funcional

```
def fatorial(n):  
    f = 1  
    for i in range(1, n + 1):  
        f *= i  
    return f
```

Mudança de estado

Iteração e sequência

```
def fatorial(n):  
    if n <= 1:  
        return 1  
    else:  
        return n * fatorial(n - 1)
```

Não há variáveis (sem estado)

Recursão

Por quê?

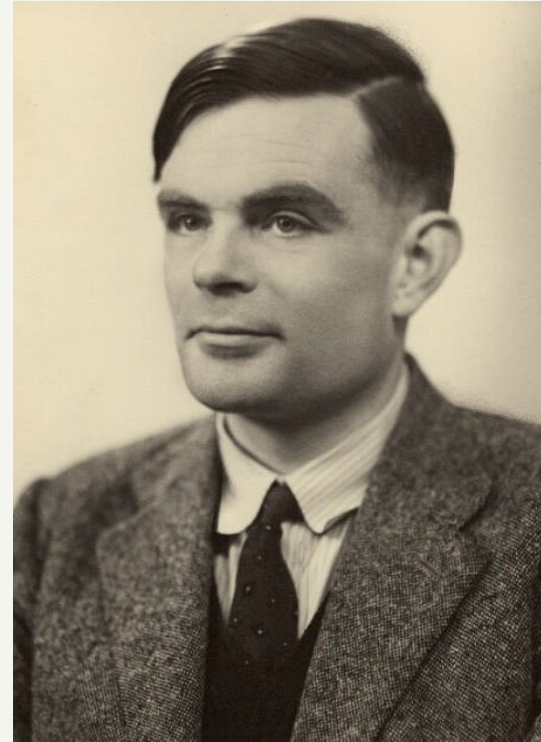
- » Pontos positivos da programação funcional
 - » Mais abstrata (alto nível)
 - » Soluções mais modulares
 - » Fácil de testar e depurar
 - » Não propensa a efeitos colaterais
 - » Simplifica paralelização

Tese de Church-Turing

- » Embora fundamentalmente diferentes, o cálculo lambda e a máquina de Turing são equivalentes



Cálculo lambda, proposto por Alonzo Church ^[1]



Máquina de Turing, proposto por Alan Turing ^[2]

[1] https://en.wikipedia.org/wiki/File:Alonzo_Church.jpg
[2] <https://en.wikipedia.org/wiki/File:Alan-Turing.jpg>

Tese de Church-Turing

- » Consequências da tese de Church-Turing
 - » Existe uma expressão lambda que simula uma máquina de Turing universal
 - » Existe uma máquina de Turing que calcula expressões lambda arbitrárias
 - » Qualquer modelo de computação universal será equivalente à máquina de Turing
 - » Qualquer paradigma pode ser utilizado para simular um modelo de computação Turing-completo

Agenda

- » Funcional vs. imperativo
- » Python funcional
- » Funções de ordem mais alta
- » Compreensões de listas
- » Geradores e avaliação preguiçosa

Linguagens funcionais

» LISP

- » Primeira linguagem funcional
- » Uma **linguagem funcional "pura"**
- » Não possui variáveis
- » Todos os dados são **imutáveis**
- » O principal tipo de dados é a lista
- » Dados e programas possuem a mesma representação

Linguagens funcionais

» LISP

```
; LISP Example function
; The following code defines a LISP predicate function
; that takes two lists as arguments and returns True
; if the two lists are equal, and NIL (false) otherwise
(DEFUN equal_lists (lis1 lis2)
  (COND
    ((ATOM lis1) (EQ lis1 lis2))
    ((ATOM lis2) NIL)
    ((equal_lists (CAR lis1) (CAR lis2))
     (equal_lists (CDR lis1) (CDR lis2)))
    (T NIL)
  )
)
```

Linguagens funcionais

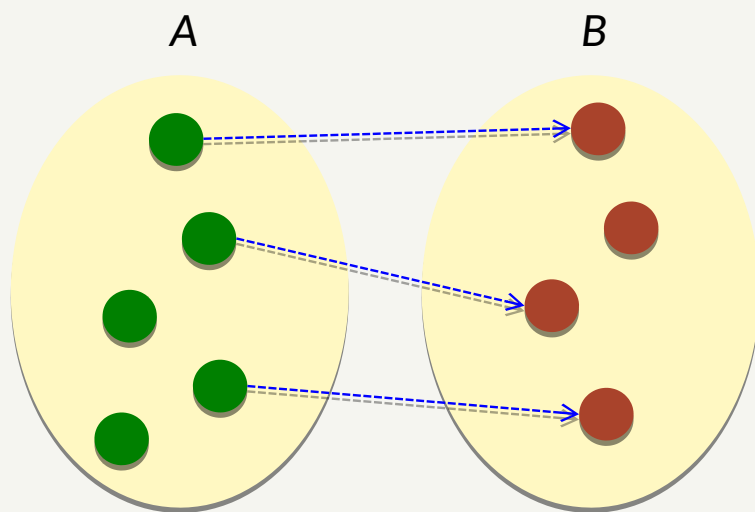
» Python funcional

- » Programação "puramente" funcional é **difícil**
- » Algumas tarefas são naturalmente associadas a estados, como processamento de E/S
 - ~ Leitura/escrita do teclado/de arquivos
- » Na prática, linguagens "puramente" funcionais são menos utilizadas
 - ~ Multiparadigma permite empregar conceitos funcionais em linguagens menos abstratas

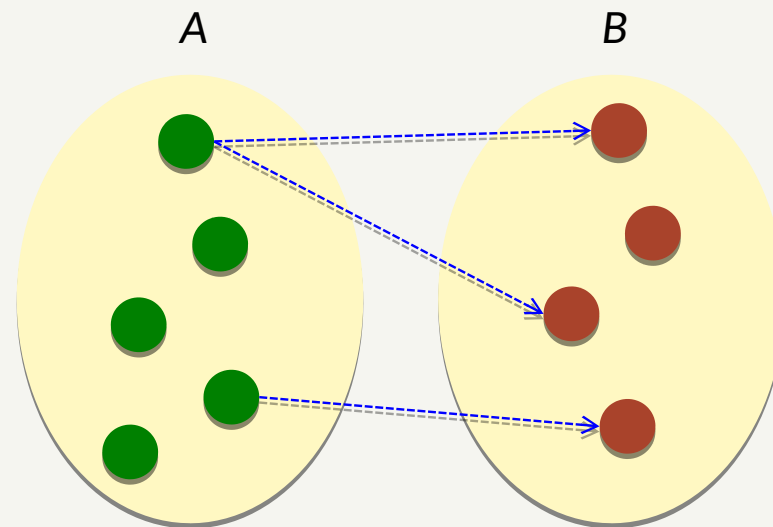
Python funcional

» Princípios norteantes

- » Uma função é um mapeamento da entrada para uma saída única



Função



Relação binária

Python funcional

» Princípios norteantes

- » Uma função é um mapeamento da entrada para uma saída única
 - ~ O resultado de uma função depende exclusivamente dos argumentos
 - ~ Efeitos colaterais inexistentes ou restritos
 - ~ Uma função chamada duas vezes com os mesmos argumentos deve retornar o mesmo valor

Python funcional

- » Princípios norteantes
 - » A recursão substitui a iteração
 - ~ Característica marcante do paradigma
- » Manipulação de coleções
 - ~ Listas
 - ~ Tuplas (listas imutáveis)
 - ~ Dicionários
 - ~ Conjuntos

Python funcional

» Princípios norteantes

» Sem efeitos colaterais

- ~ Estritamente, sem atribuições
- ~ Exceção: em alguns casos, variáveis locais
- ~ Mas as funções não podem compartilhar estados!
 - ~ Uma função não pode alterar um objeto
 - ~ Não utilizamos variáveis globais

Python funcional

» Princípios norteantes

» Ordem de avaliação é pouco importante

- ~ Os argumentos podem ser avaliados em qualquer ordem

- ~ Inclusive, simultaneamente

- ~ Isso facilita bastante a paralelização

Python funcional

» Princípios norteantes

» Ordem de avaliação é pouco importante

- ~ Os argumentos podem ser avaliados em qualquer ordem

- ~ Inclusive, simultaneamente

- ~ Isso facilita bastante a paralelização

» Funções de primeira classe ou de ordem mais alta

» Transparência referencial

Funções em Python

» Funções de primeira classe

- » As funções de Python são **objetos de primeira classe** porque suportam todas as operações disponíveis para objetos de primeira classe
 - ~ Uma função pode ser passada como **argumento** para outra função
 - ~ Uma função pode ser **retornada** por outra função

Funções em Python

- » Funções de ordem mais alta
 - » Uma **função de ordem mais alta** é aquela que
 - ~ Recebe outra função como argumento; ou
 - ~ Retorna uma função como argumento

```
>>> def aplicar(fun, arg):  
...     return fun(arg)  
...  
  
>>> aplicar(print, "Hello, world!")  
Hello, world!
```

Funções em Python

» Funções lambda

```
>>> def quadrado(x):  
...     return x * x  
...  
  
>>> quadrado(6)  
36  
  
>>> quad = lambda x : x * x  
  
>>> quad(6)  
36
```

Python só permite funções lambda com uma linha

Funções em Python

» Funções lambda

- » A notação lambda permite criar uma função anônima, sem vínculo a nenhum nome

~ **lambda** x : x * x

- » Funções lambda podem ser vinculadas a variáveis ou podem ser utilizadas em expressões

~ >>> aplicar(**lambda** x : x * x, 6)
36

Funções em Python

» Funções lambda

```
<lambda> ::= "lambda" [ <parâmetros> ] : <expressão>  
<parâmetros> ::= <id> { , <id> }
```

Tarefas de Python funcional

- » Retornar o menor elemento de uma lista
- » Ordenar uma lista
- » Encontrar o máximo divisor comum
 - » Algoritmo de Euclides: subtraia o menor do maior até os valores serem iguais
- » Inverter uma lista
- » Verificar se um número é primo

Agenda

- » Funcional vs. imperativo
- » Python funcional
- » Funções de ordem mais alta
- » Compreensões de listas
- » Geradores e avaliação preguiçosa

Map, filter e reduce

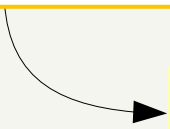
- » Três importantes operações em linguagens funcionais são
 - » *Map*: mapeia um conjunto de valores para a imagem de uma função
 - » *Filter*: filtra uma coleção
 - » *Reduce*: aplica uma função em uma coleção, reduzindo a um único valor
- » As funções map e filter retornam geradores, que podem ser convertidos em listas

Filter

» `filter(função, lista)`

- » Faz com que a função **atue como um filtro** sobre os elementos da lista
- » Os elementos que são mapeados para um valor falso são filtrados (removidos)

```
def par(x):  
    return x % 2 == 0  
  
list(filter(par, [1, 2, 3, 4, 5, 6]))
```



Resultado: [2, 4, 6]

Map

» `map(função, lista)`

- » Utiliza uma função para **transformar cada elemento** da lista
- » Retorna uma nova lista com cada elemento mapeado por meio da função

```
def quadrado(x):  
    return x ** 2  
  
list(map(quadrado, [1, 2, 3, 4, 5, 6]))
```

Resultado: [1, 4, 9, 16, 25, 36]

Reduce

- » `reduce(função, lista)`
 - » A função deve ter dois parâmetros: um acumulador e um elemento da lista
 - ~ Inicialmente, o acumulador é o primeiro elemento da lista
 - ~ Então o valor mapeado pela função se torna o acumulador para o próximo elemento

```
from functools import reduce
```

```
def somador(acc, elem):  
    return acc + elem
```

```
reduce(somador, [1, 2, 3, 4, 5, 6])
```



Resultado: 21

Funções de ordem mais alta e lambda

- » Não é necessário declarar cada função separadamente com **def**
 - » Podemos empregar funções lambda
 - » As listas podem ser implícitas
 - » Por exemplo, podemos empregar um iterador
 - ~ `list(range(6))`: produz [0, 1, 2, 3, 4, 5]
 - ~ `list(range(1, 6))`: produz [1, 2, 3, 4, 5]
 - ~ `list(range(1, 6, 2))`: produz [1, 3, 5]

Funções de ordem mais alta e lambda

- » Qual o quadrado dos números de 1 a 5?
 - » Mapeie a lista com $f(x) = x^2$
- » Quais os números pares entre 0 e 10?
 - » Filtre os elementos ímpares
- » Qual o cubo dos números pares entre 0 e 10?
 - » Filtre e mapeie a lista com $f(x) = x^3$
 - » Ou gere apenas os ímpares
- » Qual a soma dos 10 primeiros inteiros?
 - » Reduza cumulativamente para um valor único

Por quê?

- » Vantagens de *map, filter, reduce*
 - » Podem simplificar laços complexos (em programas imperativos)
 - » Podem ser encadeadas
 - » Muitas computações podem ser reduzidas a essas três operações
 - » Facilmente distribuídas
 - ~ Permitem paralelização do código

Agenda

- » Funcional vs. imperativo
- » Python funcional
- » Funções de ordem mais alta
- » **Compreensões de listas**
- » Geradores e avaliação preguiçosa

Notação implícita de conjuntos

- » Em Matemática Discreta, podemos representar um conjunto implicitamente por meio de uma expressão
 - » O quadrado dos números pares de 1 a 9
 - ~ $Q = \{n^2 \mid n \in [1, 10[\wedge n \equiv 0 \pmod{2}\}$
 - ~ $Q = \{4, 16, 36, 64\}$
- » Uma lista pode ser expressa de forma semelhante

```
[i ** 2 for i in range(1, 10) if i % 2 == 0]
```

Compreensão de listas

- » **Compreensão de lista** ou *list comprehension* (**LC**) é uma notação que substitui *map* e *filter*
- » Mapeamento através de uma expressão

```
[expressão for iterador in lista]
```

- » Equivalente a

```
res = []  
for iterador in lista:  
    res.append(expressão)
```

Compreensão de listas

- » LC pode ser associado a um condicional
 - » Apenas os elementos para os quais a condição é verdadeira são mapeados

```
[expressão for iterador in lista if condição]
```

- » Equivalente a

```
res = []  
for iterador in lista:  
    if condição:  
        res.append(expressão)
```

Compreensão de listas

- » Algumas pessoas (incluindo o criador de Python) defendem que LC é mais simples e legível do que funções lambda
- » Qual parece mais legível?

```
lis = [-1, 10, 8, -8, 5, -6, 3, 11, -13, 27, 30]  
rs = list(map(lambda x: x*x, filter(lambda x: x > 0, lis)))
```

```
lis = [-1, 10, 8, -8, 5, -6, 3, 11, -13, 27, 30]  
rs = [x*x for x in lis if x > 0]
```

Compreensão de listas

- » Algumas pessoas (incluindo o criador de Python) defendem que LC é mais simples e legível do que funções lambda
- » E com funções nomeadas?

```
lis = [-1, 10, 8, 7, 5, -3, 3, 11, -13, 27, 30]
pares = lambda x: x % 2 == 0
div5 = lambda x: x/5
res = list(map(div5, filter(pares, lis)))
```

```
lis = [-1, 10, 8, 7, 5, -3, 3, 11, -13, 27, 30]
res = [x/5 for x in lis if x % 2 == 0]
```


Quantificadores

- » Python possui dois quantificadores
 - » Quantificador existencial \exists (**any**)
 - ~ Retorna verdadeiro se **pelo menos um** elemento da lista possui um valor verdade
 - ~ True, valores numéricos diferentes de 0 e 0.0 e *strings* e coleções não vazias
 - » Quantificador universal \forall (**all**)
 - ~ Retorna verdadeiro se **todos** os elementos da lista possuem valor verdade

Quantificadores

» Número primo

» n é primo se $\neg \exists k \in [2, n[\mid n \equiv 0 \pmod{k}$

```
>>> def primo(n):  
...     if n < 2: return False  
...     else: return not any(map(lambda k: n % k == 0, range(2, n)))  
  
>>> primo(2)  
True  
  
>>> primo(5)  
True  
  
>>> primo(8)  
False
```