

Lista de Computação I (Python)

DCC / UFRJ

Pedro Maciel Xavier (monitor)
pedromxavier@poli.ufrj.br

January 10, 2020

Introdução

Essa lista de exercícios ainda se encontra em desenvolvimento. A intenção é que ela tenha um gabarito bem aberto, deixando muito das respostas para a criatividade do aluno. As questões estão organizadas em relação aos assuntos. Mesmo assim cada exercício demanda conhecimentos da disciplina como um todo. Elas tem estrelinhas ★ indicando a dificuldade estimada. É importante você tire suas dúvidas e dê um retorno do que achou dos exercícios através do e-mail no cabeçalho.

Boa diversão!



Contents

1	Funções e Recursividade (<code>def</code>, <code>return</code>)	3
1.1	? ★	4
1.2	? ★	4
1.3	Fibonacci <i>X-treme</i> ★	5
1.4	Tempo ★★★	6
1.5	Diferencial I ★	6
1.6	Diferencial II ★★	7
2	Condições (<code>if</code>, <code>elif</code>, <code>else</code>)	8
2.1	Funções por partes ★	8
2.2	Árvores de Decisão !Incompleto! ★★★	9
3	Laços (<code>for</code>, <code>while</code>)	10
3.1	Sequência de Collatz ★★	10
3.2	Integral I ★★★	10
4	Números (<code>int</code>, <code>float</code>, <code>complex</code>)	11
4.1	Números Primos I ★★	11
4.2	Gerando números "aleatórios" ★	11
4.3	Música I ★	11
5	<i>Strings</i> e Texto (<code>str</code>)	13
5.1	Unidades de Medida ★★	13
5.2	DNA I ★★★	13
5.3	Separando sílabas ★★★	14
5.4	Bases numéricas alternativas !Incompleto! ★★★	15
6	Listas e Tuplas (<code>list</code>, <code>tuple</code>)	17
6.1	Crivo ★★	17
7	Conjuntos e Dicionários (<code>set</code>, <code>dict</code>)	18
7.1	Música II !Incompleto! ★★	18
7.2	<i>Blockchain</i> !Incompleto! ★★★★	18
8	Arquivos e Diretórios (<code>sys</code>, <code>os</code>, <code>with</code>, <code>open</code>)	19
8.1	Sem sair do diretório ★★	19
8.2	DNA II !Incompleto! ★★★	19

1 Funções e Recursividade (**def**, **return**)

Interlúdio: Argumentos de funções, ***args** e ****kwargs**

Oberseve esta forma de definir uma função:

```
1 def f(x, *y, **z):
2     print(x)
3     print(y)
4     print(z)
5 ...
6 >>> f(0, 1, 2, 3, penúltimo=4, último=5)
7 0
8 (1, 2, 3)
9 {'penúltimo' : 4, 'último' : 5}
```

Pode parecer um tanto confuso de início, mas o que estamos falando aqui é de parâmetros opcionais (***args**) e de parâmetros nomeados (*keyword args*, ou ****kwargs**).

1 - ***args**

Quando um dos parâmetros de uma função é precedido por um asterisco (*), ele passa a armazenar uma quantidade indeterminada de valores em uma tupla (**tuple**).

2 - ****kwargs**

Atua de forma bem semelhante ao anterior, mas em vez de receber valores comuns, este receberá pares na forma *chave=valor* que serão guardados em um dicionário (**dict**).

De um ponto de vista mais filosófico, pensamos * e ** como operadores de **desempacotamento**. De fato eles tem muitos usos. Podemos simplesmente desempacotar uma lista dentro da outra:

```
1 >>> x = [1, 2, 3]
2 >>> y = [0, *x, 4]
```

```
3 >>> print(y)
4 [0, 1, 2, 3, 4]
```

E o mesmo vale pra dicionários:

```
1 >>> x = {1: 'b', 2: 'c', 3: 'd'}
2 >>> y = {0: 'a', **x, 4: 'e'}
3 >>> print(y)
4 {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e'}
```

Mas também existem outros usos, como separar o primeiro e o último elemento do restante de uma lista (**list**):

```
1 >>> L = ['a', 'b', 'c', 'd']
2 >>> x, *y, z = L
3 >>> x
4 'a'
5 >>> y
6 ['b', 'c']
7 >>> z
8 'd'
```

1.1 ? *

1.2 ? *

Interlúdio: Funções são objetos

No Python quase tudo é objeto. O conceito de objeto vai ficar mais claro no curso de Computação II. O importante agora é saber que uma função não é tão diferente dos outros tipos, como os inteiros (**int**), listas (**list**) e strings (**str**). Isso tem uma implicação imediata: podemos passar funções como parâmetros pra outras funções. Isso pode ser muito útil, como veremos mais

adiante, para definir funções a partir de outras, como no caso das derivadas, integrais e dos campos vetoriais, por exemplo.

1.3 Fibonacci *X-treme* ★

A sequência de Fibonacci é definida por

$$\begin{cases} F_n = F_{n-1} + F_{n-2} \\ F_0 = 0 \text{ e } F_1 = 1 \end{cases}$$

O n -ésimo termo pode ser obtido pela função

```
1 def f(n):
2     if n == 0:
3         return 0
4     elif n == 1:
5         return 1
6     else:
7         return f(n-1) + f(n-2)
```

Listing 1: "Fibonacci"

Desafio: Tente encontrar, usando a função acima, o número F_{50} . Reflita um pouco sobre o resultado.

Interlúdio: Decoradores

Decoradores são funções que proporcionam uma maneira simples de alterar o funcionamento de uma função no momento da sua definição. Um decorador recebe uma função como parâmetro e deve retornar também uma função.

```
1 def mod3(f):
2     def g(x):
3         return f(x) % 3
4     return g
5
6 @mod3
```

```

7 def f(x):
8     return x ** 2 + 1

```

1.4 Tempo ***

Agora que já sabemos como construir uma função e decorá-la, podemos fazer um decorador que nos ajude a medir o tempo que uma função leva pra rodar. Para isso, a melhor ideia é usar a função `clock`, do módulo `time`, que agora no Python 3 se chama `perf_counter`.

```

1 from time import perf_counter as clock

```

Desafio: O seu decorador deve alterar a função para que ela retorne uma tupla contendo o resultado e o tempo de execução em segundos.

1.5 Diferencial I *

Aprendemos em Cálculo I que a derivada de uma função $f(x)$ pode ser calculada como:

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

Se escolhermos um h pequeno, algo como 0.001, podemos deixar a ideia de limite de lado e partir para uma aproximação

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \text{ para } h \ll 1$$

Outras aproximações razoáveis são

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

$$f'(x) \approx \frac{\text{Im}\{f(x + \mathbf{i}h)\}}{h}$$

sendo essa última um pouco mais holística.

Desafio: Faça uma função `d(f, h)` que receba uma função `f` e um número bem pequeno `h` e retorne uma outra função, que aproxime a derivada de `f`.

Detalhe: Você pode definir o parâmetro `h` por um valor padrão (*default*):

```
1 def d(f, h=1e-6):
2     ...
```

1.6 Diferencial II $\star\star$

[illegible]

Desafio: Faça uma função $D(f, n, h)$ que calcule a n -ésima derivada da função f , de maneira recursiva.

2 Condições (**if**, **elif**, **else**)

2.1 Funções por partes ★

Funções podem ser definidas por partes, isto é, utilizamos uma função para cada trecho do domínio. Dentre os exemplos mais conhecidos temos

A função de *Heaviside*, ou função degrau:

$$u(t) \triangleq \begin{cases} 1, & \text{se } t \geq 0 \\ 0, & \text{c.c.} \end{cases}$$

A sua derivada, o delta de Dirac ¹²:

$$\delta(t) \triangleq \begin{cases} \infty, & \text{se } t = 0 \\ 0, & \text{c.c.} \end{cases}$$

O módulo de um número real:

$$|x| \triangleq \begin{cases} x, & \text{se } x \geq 0 \\ -x, & \text{c.c.} \end{cases}$$

Mas podemos pensar muitos outros exemplos

$$f(x) \triangleq \begin{cases} -2x, & \text{se } 1 \leq x < 2 \\ x, & \text{se } 3 \leq x < 5 \\ x^2, & \text{c.c.} \end{cases}$$

$$g(x, y) \triangleq \begin{cases} x^2 + y^2, & \text{se } x^2 + y^2 \leq 1 \\ x^2 + xy + y^2, & \text{se } 1 < x^2 + y^2 \leq 4 \\ 0, & \text{c.c.} \end{cases}$$

¹Uma definição mais precisa supõe que $\int_{-\infty}^{\infty} \delta(t) dt = 1$

²O infinito (∞) no Python é dado por `float("inf")`.

Desafio: Utilize condicionais para criar funções definidas por partes, como as dos exemplos.

2.2 Árvores de Decisão **!Incompleto!** ★★★

Uma técnica muito comum em aprendizado de máquina atualmente, uma **Árvore de Decisão** é uma estrutura de condicionais que se ramifica a cada etapa.

Idade	Altura (m)	Peso (kg)		
12	1.33	40		
65	1.70	72		
15	1.63	51		
59	1.65	55		
34	1.68	84		
42	1.72	78		
9	1.10	33		
22	1.80	79		

Desafio: Usando condicionais, construa uma função que consiga classificar os dados da tabela, tentando usar o mínimo possível de etapas.

3 Laços (**for**, **while**)

3.1 Sequência de Collatz ★★

A sequência de Collatz é descrita recursivamente por:

$$f(n) \triangleq \begin{cases} 3n + 1, & \text{se } n \text{ for ímpar} \\ n \div 2, & \text{se } n \text{ for par} \end{cases}$$

Por exemplo, começamos com $n = 26$. Após sucessivas aplicações temos:

$$26 \rightarrow 13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

Isso nos dá uma sequência com 11 números. 40 é maior que 26, mas sua sequência só teria 9 números.

Ainda não se sabe se todos os números induzem uma sequência que termina em 1. No entanto, até agora não foi encontrado um número sequer em que isso não tenha acontecido!

Desafio: Faça uma função que calcule o comprimento da sequência gerada a partir de um número natural n qualquer.

3.2 Integral I ★★★

A **Integral de Riemann** é dada por:

$$\int_a^b f(x) \, dx = \lim_{N \rightarrow \infty} \sum_{k=0}^N f(x_k) \cdot \delta x \text{ onde } \delta x = \frac{(b-a)}{N} \text{ e } x_k = a + k \cdot \delta x$$

No entanto, vamos nos ater somente a um N suficientemente grande e deixar o limite para lá.

Desafio: Faça uma função que calcule o valor da integral definida de uma função f , dados os limites a e b , assim como o valor de N .

4 Números (**int**, **float**, **complex**)

4.1 Números Primos I ★★

Desafio: Faça uma função que verifique se um número é primo ou não.

4.2 Gerando números "aleatórios" ★

Gerar um número aleatório é um pouco complicado em geral. Muitas formas de se fazer isso hoje em dia se baseiam em processos da natureza como fluidos em regime turbulento ou fenômenos quânticos. No entanto, é possível chegar perto disso com muito menos. Para começar, escolhemos uma potência de 2, como $N = 2^{16} = 65536$. Depois disso, escolhemos um número primo no meio do caminho, digamos, $p = 25773$. Por fim, um número ímpar menor que o número primo p , por exemplo, $b = 13849$.

Partimos de um número n_0 qualquer no intervalo $[0, 25536)$ e calculamos o sucessor da seguinte forma:

$$n_j = p \times n_{j-1} + b \mod N$$

Para obter um número "aleatório" distribuído de maneira uniforme no intervalo $[0, 1)$, basta calcular

$$x_j = \frac{n_j}{N}$$

Desafio: Faça o seu próprio gerador de números aleatórios!

4.3 Música I ★

Uma **nota musical** é o nome que se dá ao som relativo a uma frequência específica. Uma **oitava** é o intervalo entre uma nota musical e aquela que possui o dobro da sua frequência. Na música ocidental, uma **oitava** é dividida em 12 tons.

A nota *lá*, cujo símbolo é A, é normalmente dada pela frequência de 440Hz. Isso significa que também chamaremos de *lá* os sons de 220Hz, 880Hz e assim por diante.

A	A#	B	C	C#	D	D#	E	F	F#	G	G#
0	1	2	3	4	5	6	7	8	9	10	11

Desafio: Faça uma função que calcule a frequência f de cada nota, a partir do seu número, supondo que $f(0) = 440\text{Hz}$.

5 *Strings* e Texto (**str**)

5.1 Unidades de Medida ★★

T	G	M	K	–	m	u	n	p
10^{12}	10^9	10^6	10^3	1	10^{-3}	10^{-6}	10^{-9}	10^{-12}

Desafio: Defina uma função que receba:

1. O valor
2. A unidade de medida

e que retorne uma string com o valor, a escala e a unidade, como no exemplo:

```
1 >>> f(2_000, 'g')
2 2.0 Kg
3 >>> f(1_200_000, 'Hz')
4 1.2 MHz
```

Bônus: A medida de *bits* (b) e *bytes* (B) ocorre de uma forma um pouco diferente: A mudança de escala acontece a cada 1024 unidades, e não a cada 1000 como em outras grandezas. Além disso, nesse caso não existem escalas fracionárias. Inclua condições para representar *bits* e *bytes*.

5.2 DNA I ★★★

Uma sequência de DNA (*ácido desoxirribonucleico*) é composta por 4 nucleotídeos: **A**denina, **C**itosina, **G**uanina **T**imina. Já no RNA mensageiro (mRNA), a **T**imina está ausente, e dá lugar para a **U**racila. Quando um ribossomo realiza a transcrição de DNA em mRNA ele segue uma regra muito simples:

A	→	U
T	→	A
C	→	G
G	→	C

Além disso, o DNA possui alguns códons (sequências de 3 nucleotídeos) que indicam o início e fim de uma transcrição. ATG marca o início, enquanto TAG, TAA, e TGA indicam o final.

Podemos representar uma fita de DNA usando uma *string* (**str**), como por exemplo:

"ACAATGTGTACGACTATACTACGTGCCTATTTCGCATACGATCGATGACTAG"

Desafio: Faça uma função que traduza uma fita de DNA como faria um ribossomo. Ela deve receber uma *string* de DNA e retornar uma *string* de mRNA. Por exemplo, a fita de DNA acima seria transcrita em

"ACAUGCUGAUAUGAUGCACGGAUAAGCGUAUGCUGAUCUAGCUACUG"

5.3 Separando sílabas ★★★

Separar sílabas é uma tarefa interessante. O Português é uma das poucas línguas que possui um algoritmo que permite saber como separar as sílabas de uma palavra qualquer, isto é, se uma palavra nova for criada hoje em português, já é possível dividi-la mesmo sem conhecer a sua origem.

Flamengo → Fla-men-go

Um algoritmo de separação silábica, para uma língua qualquer, normalmente conta com duas coisas: Um conjunto de regras e um conjunto de exceções.

As regras nada mais são do que padrões que, uma vez observados, implicam em uma cisão na palavra. Por exemplo, consoantes dobradas como **rr** e **ss** sempre vão indicar uma separação:

Pássaro → Pás-sa-ro
Carrossel → Car-ros-sel

As exceções nada mais são do que um conjunto de palavras que já sabemos como separar, armazenadas em pares da forma (palavra, separação).

Desafio: Crie um conjunto de regras para que, dada uma palavra (**str**), a função retorne uma lista com a respectiva separação silábica. Exemplo:

```

1 >>> f('pássaro')
2 ['pás', 'sa', 'ro']
3 >>> f('carrossel')
4 ['car', 'ros', 'sel']
5 >>> f('paralelepípedo')
6 ['pa', 'ra', 'le', 'le', 'pí', 'pe', 'do']

```

Listing 2: "Separando sílabas"

5.4 Bases numéricas alternativas **!Incompleto!** ***

Binária

0 0 1 1

Figure 1: Dígitos Binários, 2020 d.C.

Hexadecimal

0	0	1	1	2	2	3	3
4	4	5	5	6	6	7	7
8	8	9	9	A	10	B	11
C	12	D	13	E	14	F	15

Figure 2: Dígitos Hexadecimais, 2020 d.C.

Babilônia

Estamos habituados com a base decimal, muito provavelmente porquê temos 10 dedos. O pessoal da antiga Babilônia tinha um sistema sexagesimal: Isso mesmo, eram 60 dígitos diferentes!




























































	1		11		21		31		41		51
	2		12		22		32		42		52
	3		13		23		33		43		53
	4		14		24		34		44		54
	5		15		25		35		45		55
	6		16		26		36		46		56
	7		17		27		37		47		57
	8		18		28		38		48		58
	9		19		29		39		49		59
	10		20		30		40		50		

Figure 3: Dígitos Babilônicos 1500 a.C.

Na verdade, eram só 59 porquê eles ainda não tinham o zero e deixavam um espaço em branco no lugar dele o que na hora das contas devia ser uma Torre de Babel! Dá pra perceber que apesar disso, a composição dos símbolos ainda tem sua própria carga da notação decimal, mesclando-a com a base 6.

Desafio: Crie uma função que transforme um número na base decimal em uma representação (**str**) na base babilônica, assim como uma outra que faça o caminho contrário.

6 Listas e Tuplas (**list**, **tuple**)

6.1 Crivo ★★

O Eratóstenes foi um cara legal que viveu em 200 a.C., bolou um crivo e calculou a curvatura da terra.



Figure 4: Eratóstenes de Cirene

Um **crivo** é uma forma de saber quem são os números primos até um determinado limite \mathbf{N} . O Crivo de Eratóstenes funciona de forma bem simples:

1. Escrevemos em uma tabela todos os números de 0 até \mathbf{N} .
2. Riscamos o 0 e o 1.
3. Se um número não estiver riscado, riscamos todos os múltiplos deste, menos o próprio.
4. Andamos para o próximo número e repetimos a etapa anterior.

Uma maneira interessante de fazer isso é criando uma lista de tamanho $\mathbf{N}+1$, ou seja, cujos índices vão de 0 até \mathbf{N} .

Desafio: Implemente o crivo de Eratóstenes e, em seguida, elabore versões melhoradas do algoritmo, até onde conseguir.

7 Conjuntos e Dicionários (**set**, **dict**)

7.1 Música II **!Incompleto!** ★★

Usando a função do exercício **Música I**[4], podemos usar um dicionário para obter a frequência a partir do símbolo que representa a nota.

Interlúdio: **hash !Incompleto!**

Para entender um pouco melhor como funcionam os conjuntos (**set**) e dicionários (**dict**) é importante compreender o que é e como funciona o *hash*.

7.2 *Blockchain* **!Incompleto!** ★★★

Você já deve ter ouvido falar no *Bitcoin*. Para prosseguir nessa questão é importante entender bem como funciona o *hash*[7].

Como o nome diz, um *Blockchain* se trata de um conjunto de blocos encadeados que representam transações, não somente financeiras, mas de um modo geral. Podemos pensar um *Blockchain* como um grande livro de caixa, onde cada bloco é o registro de uma atividade. Um bloco, por sua vez, contém as informações que caracterizam uma transação, vejamos:

De:	Pedro
Para:	Anna
Valor:	\$42.0

Figure 5: Bloco

8 Arquivos e Diretórios (**sys**, **os**, **with**, **open**)

Interlúdio: **csv**

Arquivos no formato `.csv` (*comma separated values*) são muito utilizados para representar tabelas em um arquivo de texto. São facilmente lidos por programas como o Excel™.

Por exemplo, o arquivo

```
1 nome,CPF,idade
2 João,82252631704,11
3 Pedro,12432236702,21
4 Anna,13241232392,15
```

é interpretado como

nome	CPF	idade
João	82252631704	11
Pedro	12432236702	21
Anna	13241232392	15

8.1 Sem sair do diretório ★★

Frequentemente, um programa lê e escreve em determinados arquivos. No entanto, queremos trabalhar com arquivos que estão numa pasta específica. Como fazer pra ir até essa pasta, fazer o que é necessário e retornar ao diretório original?

Desafio: Feito isso, crie um decorador[1] que faça com que uma função sempre retorne ao diretório que estava antes da sua execução.

8.2 DNA II **!Incompleto!** ★★★

Ler o DNA de um arquivo enorme.