

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/343971915>

# Introdução a Computação em Python: Problemas e Aplicações

Technical Report · August 2020

DOI: 10.13140/RG.2.2.29068.95368

---

CITATIONS

0

---

READS

9,407

1 author:



Alexandre L M Levada

Universidade Federal de São Carlos

135 PUBLICATIONS 435 CITATIONS

SEE PROFILE



Departamento de Computação  
Centro de Ciências Exatas e Tecnologia  
Universidade Federal de São Carlos

# **Introdução a Computação em Python: Problemas e Aplicações**

Apostila com exercícios práticos e soluções

Prof. Alexandre Luis Magalhães Levada  
Email: [alexandre.levada@ufscar.br](mailto:alexandre.levada@ufscar.br)

# Sumário

A linguagem Python.....	3
O problema da precisão numérica em computação.....	5
Estruturas sequenciais.....	8
Estruturas de seleção.....	11
Estruturas de repetição.....	16
Modularização e funções.....	26
Variáveis compostas 1D: Listas, strings e vetores.....	33
Variáveis compostas 2D: Matrizes.....	45
Arquivos.....	51
Dicionários.....	60
Ordenação de dados.....	70
Recursão.....	76
Aplicação: A recorrência logística e os sistemas caóticos.....	83
Aplicação: o dilema do prisioneiro.....	90
Aplicação: autômatos celulares.....	97
Bibliografia.....	105
Sobre o autor.....	106

“Experiência não é o que acontece com um homem; é o que ele faz com o que lhe acontece”  
(Aldous Huxley)

# A linguagem Python

*“Python é uma linguagem de programação de alto nível, interpretada, de script, imperativa, orientada a objetos, funcional, de tipagem dinâmica e forte. Foi lançada por Guido van Rossum em 1991.[1] Atualmente possui um modelo de desenvolvimento comunitário, aberto e gerenciado pela organização sem fins lucrativos Python Software Foundation. Apesar de várias partes da linguagem possuírem padrões e especificações formais, a linguagem como um todo não é formalmente especificada. A linguagem foi projetada com a filosofia de enfatizar a importância do esforço do programador sobre o esforço computacional. Prioriza a legibilidade do código sobre a velocidade ou expressividade. Combina uma sintaxe concisa e clara com os recursos poderosos de sua biblioteca padrão e por módulos e frameworks desenvolvidos por terceiros. Python é uma linguagem de propósito geral de alto nível, multiparadigma, suporta o paradigma orientado a objetos, imperativo, funcional e procedural. Possui tipagem dinâmica e uma de suas principais características é permitir a fácil leitura do código e exigir poucas linhas de código se comparado ao mesmo programa em outras linguagens”. (Wikipedia)*

Nesse curso, optamos pela linguagem Python, principalmente pela questão didática, uma vez que a sua curva de aprendizado é bem mais suave do que linguagens de programação como C, C++ e Java. Existem basicamente duas versões de Python coexistindo atualmente. O Python 2 e o Python 3. Apesar de existirem poucas diferenças entre elas, é o suficiente para que um programa escrito em Python 2 possa não ser compreendido por interpretador do Python 3. Optamos aqui pelo Python 3, pois além de ser considerado uma evolução natural do Python 2, representa o futuro da linguagem.

Porque Python 3?

- Evolução do Python 2 (mais moderno)
- Sintaxe simples e de fácil aprendizagem
- Linguagem de propósito geral que mais cresce na atualidade
- Bibliotecas para programação científica e aprendizado de máquina

Scikit_learn Scikit_image NetworkX,...
Scipy Matplotlib Statsmodels Pandas, Ipython,...
Numpy
Python Standard Library

Dentre as principais vantagens de se aprender Python, podemos citar a enorme gama de bibliotecas existentes para a linguagem. Isso faz com que Python seja extremamente versátil. É possível desenvolver aplicações científicas como métodos matemáticos numéricos, processamento de sinais e imagens, aprendizado de máquina até aplicações mais comerciais, como sistemas web com acesso a bancos de dados.

## Plataformas Python para desenvolvimento

Para desenvolver aplicações científicas em Python, é conveniente instalar um ambiente de programação em que as principais bibliotecas para computação científica estejam presentes. Isso poupa-nos muito tempo e esforço, pois além de não precisarmos procurar cada biblioteca individualmente, não precisamos saber a relação de dependência entre elas (quais devem ser instaladas primeiro e quais tem que ser instaladas posteriormente). Dentre as plataformas Python para computação científica, podemos citar as seguintes:

**a) Anaconda** - <https://www.anaconda.com/products/individual>

Uma ferramenta multiplataforma com versões para Windows, Linux e MacOS. Inclui mais de uma centena de pacotes para programação científica, o que o torna um ambiente completo para o desenvolvimento de aplicações em Python. Inclui diversos IDE's, como o idle, ipython e spyder.

**b) WinPython** - <http://winpython.github.io/>

Uma plataforma exclusiva para Windows que contém inúmeros pacotes indispensáveis, bem como um ambiente integrado de desenvolvimento muito poderoso (Spyder).

**c) Pyzo (Python to people)** - <http://www.pyzo.org/>

Um projeto que visa simplificar o acesso à plataforma Python para computação científica. Instala os pacotes standard, uma base pequena de bibliotecas e ferramentas de atualização, permitindo que novas bibliotecas sejam incluídas sob demanda.

**d) Canopy** - <https://store.enthought.com/downloads>

Mais uma opção multiplataforma para usuários Windows, Linux e MacOS.

### Repl.it

Uma opção muito interessante é o interpretador Python na nuvem [repl.it](https://repl.it)

Você pode desenvolver e armazenar seus códigos de maneira totalmente online sem a necessidade de instalar em sua máquina um ambiente de desenvolvimento local.

Nossa recomendação é a plataforma Anaconda, por ser disponível em todos os sistemas operacionais. Ao realizar o download, opte pelo Python 3, que atualmente deve estar na versão 3.7. Para a execução das atividades presentes nessa apostila, o editor IDLE é recomendado. Além de ser um ambiente extremamente simples e compacto, ele é muito leve, o que torna sua execução possível mesmo em máquinas com limitações de hardware, como pouca memória RAM e processador lento.

Após concluir a instalação, basta digitar anaconda na barra de busca do Windows. A opção Anaconda Prompt deve ser selecionada. Ela nos leva a um terminal onde ao digitar o comando idle e pressionarmos enter, seremos diretamente redirecionados ao ambiente IDLE. Um vídeo tutorial mostrando esse processo pode ser assistido no link a seguir:

<https://www.youtube.com/watch?v=PWdrdWDmJIY&t=8s>

“Não trilhe apenas os caminhos já abertos. Por serem conhecidos eles nos levam somente até onde alguém já foi um dia.” (Alexander Graham Bell)

## O problema da precisão numérica em computação

A representação de números fracionários por computadores digitais pode levar a problemas de precisão numérica. Sabemos que um número na base 10 (decimal) é representado como:

$$23457 = 2 \times 10^4 + 3 \times 10^3 + 4 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$$

Analogamente, um número binário (base 2) pode ser representado como:

$$110101 = 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 53$$

O bit mais a direita é o menos significativo e portanto o seu valor é  $1 \times 2^0 = 1$

O segundo bit a partir da direita tem valor de  $0 \times 2^1 = 0$

O terceiro bit a partir da direita tem valor de  $1 \times 2^2 = 4$

O quarto bit a partir da direita tem valor de  $0 \times 2^3 = 0$

O quinto bit a partir da esquerda tem valor de  $1 \times 2^4 = 16$

Por fim, o bit mais a esquerda tem valor de  $1 \times 2^5 = 32$

Somando tudo temos:  $1 + 4 + 16 + 32 = 53$ .

Essa é a regra para convertermos um número binário para sua notação decimal.

Veremos agora o processo inverso: como converter um número decimal para binário. O processo é simples. Começamos dividindo o número decimal por 2:

$53 / 2 = 26$  e sobra resto **1** → esse 1 será nosso bit mais a direita (menos significativo no binário)

Continuamos o processo até que a divisão por 2 não seja mais possível:

$26 / 2 = 13$  e sobra resto **0** → esse 0 será nosso segundo bit mais a direita no binário

$13 / 2 = 6$  e sobra resto **1** → esse 1 será nosso terceiro bit mais a direita no binário

$6 / 2 = 3$  e sobra resto **0** → esse 0 será nosso quarto bit mais a direita no binário

$3 / 2 = 1$  e sobra resto **1** → esse 1 será nosso quinto bit mais a direita no binário

$1 / 2 = 0$  e sobra resto **1** → esse 1 será o nosso último bit (mais a esquerda)

Note que de agora em diante não precisamos continuar com o processo pois

$0 / 2 = 0$  e sobra 0

$0 / 2 = 0$  e sobra 0

ou seja a esquerda do sexto bit teremos apenas zeros, e como no sistema decimal, zeros a esquerda não possuem valor algum. Portanto, 53 em decimal equivale a 110101 em binário.

Com números fracionários, a ideia é similar:

Na base 10:

$$456,78 = 4 \times 10^2 + 5 \times 10^1 + 6 \times 10^0 + 7 \times 10^{-1} + 8 \times 10^{-2}$$

Na base 2:

$$101,101 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-2} = 4 + 1 + 0.5 + 0.125 = 5,625$$

Na base 10, dividir por 10 significa deslocar a vírgula uma casa para a esquerda e multiplicar por 10 significa deslocar a vírgula para direita, acrescentando ou removendo zeros quando for o caso.

$$2317 / 10 = 231,7$$

$$231,7 / 10 = 23,17$$

$$23,17 / 10 = 2,317$$

Com números binários a ideia é a mesma. Mover a vírgula para a esquerda significa divisão por 2 e mover a vírgula para direita significa multiplicação por 2

$$28 = 11100$$

$$14 = 1110$$

$$7 = 111$$

$$3,5 = 11,1$$

$$1,75 = 1,11$$

Para armazenar números reais em computadores, é preciso representá-los na base 2 (binário)

No caso de números inteiros, a conversão é direta

$$25 = 11001 \text{ pois}$$

$$25 / 2 \text{ resulta em } 12 \text{ com resto } 1$$

$$12 / 2 \text{ resulta em } 6 \text{ com resto } 0$$

$$6 / 2 \text{ resulta em } 3 \text{ com resto } 0$$

$$3 / 2 \text{ resulta em } 1 \text{ com resto } 1$$

$$1 / 2 \text{ resulta em } 0 \text{ com resto } 1$$

No caso de números reais, o processo é similar

$$5,625$$

Primeiramente, devemos dividir o número em 2 partes: parte inteira e parte fracionária

A conversão é feita independentemente para cada parte. Assim, primeiro devemos converter o número 5

$$5 / 2 = 2 \text{ com resto } 1$$

$$2 / 2 = 1 \text{ com resto } 0$$

$$1 / 2 = 1 \text{ com resto } 1$$

Então, temos que  $5 = 101$

Em seguida iremos trabalhar com a parte fracionária: 0,625. Nesse caso ao invés de dividir por 2, iremos multiplicar por 2 a parte fracionária e tomar a parte inteira do resultado (a esquerda da vírgula), repetindo o processo até que não se tenha mais casas decimais depois da vírgula.

$$0,625 \times 2 = 1,25 \rightarrow 1 \text{ (primeira casa fracionária)}$$

$$0,25 \times 2 = 0,5 \rightarrow 0 \text{ (segunda casa)}$$

$$0,5 \times 2 = 1,0 \rightarrow 1 \text{ (terceira casa)}$$

Assim, temos que  $0,625 = 0,101$  e portanto  $5,625 = 101,101$

Porém, em alguns casos, alguns problemas podem surgir. Por exemplo, suponha que desejamos armazenar num computador o número 0,8 na base 10. Para isso, devemos proceder da forma descrita anteriormente.

$$0,8 \times 2 = 1,6 \rightarrow 1$$

$$0,6 \times 2 = 1,2 \rightarrow 1$$

$$0,2 \times 2 = 0,4 \rightarrow 0$$

$$0,4 \times 2 = 0,8 \rightarrow 0$$

$$0,8 \times 2 = 1,6 \rightarrow 1$$

$$0,6 \times 2 = 1,2 \rightarrow 1$$

$$0,2 \times 2 = 0,4 \rightarrow 0$$

$$0,4 \times 2 = 0,8 \rightarrow 0$$

$$0,8 \times 2 = 1,6 \rightarrow 1$$

$$0,6 \times 2 = 1,2 \rightarrow 1$$

$$0,2 \times 2 = 0,4 \rightarrow 0$$

$$0,4 \times 2 = 0,8 \rightarrow 0$$

...

Infinitas casas decimais. Porém, como na prática temos um número finito de bits, deve-se truncar o número para uma quantidade finita. Isso implica numa aproximação. Por exemplo, qual é o erro cometido ao se representar 0,8 como 0,11001100?

$$\text{Portanto, } 0,8 = 0,110011001100110011001100....$$

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{32} + \frac{1}{64} = \frac{51}{64} = 0.796875, \text{ o que implica num erro de } 0,003125.$$

O problema pode ser amplificado ao se realizar operações matemáticas com esse valor (é como se o erro fosse sendo propagado nos cálculos). Existem outros valores para que isso ocorra: 0,2, 0,4, 0,6

Ex: Forneça as representações computacionais na base 2 (binário) para os seguintes números. Quais são os erros cometidos se considerarmos apenas 8 bits para a parte fracionária?

a) 11,6

b) 27,4

c) 53,6

d) 31,2

“The real voyage of discovery consists not in seeking new landscapes, but in having new eyes.”  
(Marcel Proust)



## Estruturas sequenciais

Antes de iniciar a programação em si, iremos definir alguns comandos básicos da linguagem Python. Uma das primeiras perguntas que surgem quando vamos construir um programa é: como declarar variáveis? Python é uma linguagem dinâmica e portanto não é necessário declarar variáveis. Basta usá-las conforme a necessidade. Para ilustrar o funcionamento básico da linguagem, iremos introduzir os comandos de entrada e saída:

O comando **print** imprime informações na tela e o comando **input** lê dados do teclado.

Para demonstrar a utilização desses comandos, iremos para o modo de edição. Para isso, basta criar um novo arquivo e salvá-lo com uma extensão .py, por exemplo, entrada\_saida.py

```
# Leitura dos valores de entrada (isso é um comentário)
# Por padrão tudo que é lido pelo input é uma cadeia de caracteres
a = int(input('Entre com o valor de a: '))
b = int(input('Entre com o valor de b: '))

print() # Pula linha

# Calculo do(s) valor(es) de saída
c = a + b

# Impressão do(s) resultado(s) na saída
print('O valor de %.2f + %.2f é igual a %.2f' %(a,b,c))
```

Note que ao ler um valor digitado pelo teclado com o comando input, utilizamos o comando int() para converter a string de leitura para um dado do tipo inteiro. Por padrão, toda informação lida através do teclado em Python é do tipo string, ou seja, não permite cálculos matemáticos. Para tratar os dados de entrada como números inteiros usamos int() e para tratar como números reais usamos float(). A seguir iremos resolver alguns exercícios básicos que envolvem estruturas sequenciais, isto é, uma sequência linear de comandos.

Ex1: Sabe-se que dado uma temperatura em graus Celsius, o valor na escala Farenheit é dado por:

$$F = \frac{9}{5}C + 32$$

Faça um programa que leia uma temperatura em Celsius e imprima na tela o valor em Farenheit

```
# celsius para fahrenheit
c = float(input('Entre com a temperatura em Celsius (C): '))
f = 9*c/5 + 32
print('%.1f graus Celsius equivalem a %.1f graus Farenheit' %(c, f))
```

Ex2: Faça um programa em Python que leia 3 números reais e imprima na tela a média dos três.

```
n1 = float(input('Entre com a primeira nota: '))
n2 = float(input('Entre com a segunda nota: '))
n3 = float(input('Entre com a terceira nota: '))

media = (n1+n2+n3)/3

print('A média final é %.2f' %media)
```

Ex3: A distância entre 2 pontos P = (x1, y1) e Q = (x2, y2) é definida por:

$$d(P, Q) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Faça um programa que leias as coordenadas x1, y1, x2 e y2 e imprima a distância entre P e Q.

```
x1 = float(input('Entre com a coordenada x do ponto P (x1): '))
y1 = float(input('Entre com a coordenada y do ponto P (y1): '))
x2 = float(input('Entre com a coordenada x do ponto Q (x2): '))
y2 = float(input('Entre com a coordenada y do ponto Q (y2): '))

distancia = ((x1 - x2)**2 + (y1 - y2)**2)**0.5

print('A distância entre os pontos é %.2f' %distancia)
```

Ex4: A expressão utilizada para computar juros compostos é dada por:

$$M = C \left( 1 + \frac{J}{100} \right)^P$$

onde C é o capital inicial, J é a taxa de juros mensal (%), e P é o período em meses. Faça um programa que leia os valores de C, J e P e imprima a tela o valor final após a aplicação dos juros.

```
C = float(input('Entre com o capital inicial (R$): '))
J = float(input('Entre com a taxa de juros (% ao mês): '))
P = float(input('Entre com o período em meses: '))

M = C*(1+J/100)**P

print('O montante final será de %.2f reais' %M)
```

Ex5: Num certo país da América do Sul, a moeda nacional é a Merreca (M\$). No sistema monetário desse país só existem cédulas de M\$ 100, M\$ 50, M\$ 10, M\$ 5 e M\$ 1. Dado um valor em Merreca, faça um script que retorne a quantidade mínima de cédulas que totalizam o valor especificado. Por exemplo, se o valor for M\$ 379, devemos ter:

3 cédulas de M\$ 100  
1 cédula de M\$ 50  
2 cédulas de M\$ 10  
0 cédulas de M\$ 5  
4 cédulas de M\$ 1

```

valor = int(input('Entre com o valor em M$: '))

cedulas100 = valor // 100
resto = valor % 100
cedulas50 = resto // 50
resto = resto % 50
cedulas10 = resto // 10
resto = resto % 10
cedulas5 = resto // 5
resto = resto % 5

print('O valor digitado corresponde a:')
print('%d cédulas de M$ 100' %cedulas100)
print('%d cédulas de M$ 50' %cedulas50)
print('%d cédulas de M$ 10' %cedulas10)
print('%d cédulas de M$ 5' %cedulas5)
print('%d cédulas de M$ 1' %resto)

```

Ex6: Faça um Programa que pergunte quanto você ganha por hora e o número de horas trabalhadas no mês. Calcule e mostre o total do seu salário no referido mês, sabendo-se que são descontados 11% para o Imposto de Renda, 8% para o INSS e 5% para o sindicato, faça um programa que dê:

- salário bruto.
- quanto pagou ao INSS.
- quanto pagou ao sindicato.
- o salário líquido.
- calcule os descontos e o salário líquido, conforme a formatação abaixo:

```

+ Salário Bruto : R$
- IR (11%) : R$
- INSS (8%) : R$
- Sindicato ( 5%) : R$
= Salário Líquido : R$

```

```

ganho_hora = float(input('Ganho por hora: '))
horas_mes = int(input('Horas de trabalho por mês: '))

salario_bruto = ganho_hora*horas_mes
IR = 0.11*salario_bruto
INSS = 0.08*salario_bruto
sindicato = 0.05*salario_bruto
salario_liquido = salario_bruto - IR - INSS - sindicato

print('+ Salário bruto: R$ %.2f' %salario_bruto)
print('- IR (11%): R$ %.2f' %INSS)
print('- INSS (8%): R$ %.2f' %IR)
print('- Sindicato (5%): R$ %.2f' %sindicato)
print('= Salário líquido: R$ %.2f' %salario_liquido)

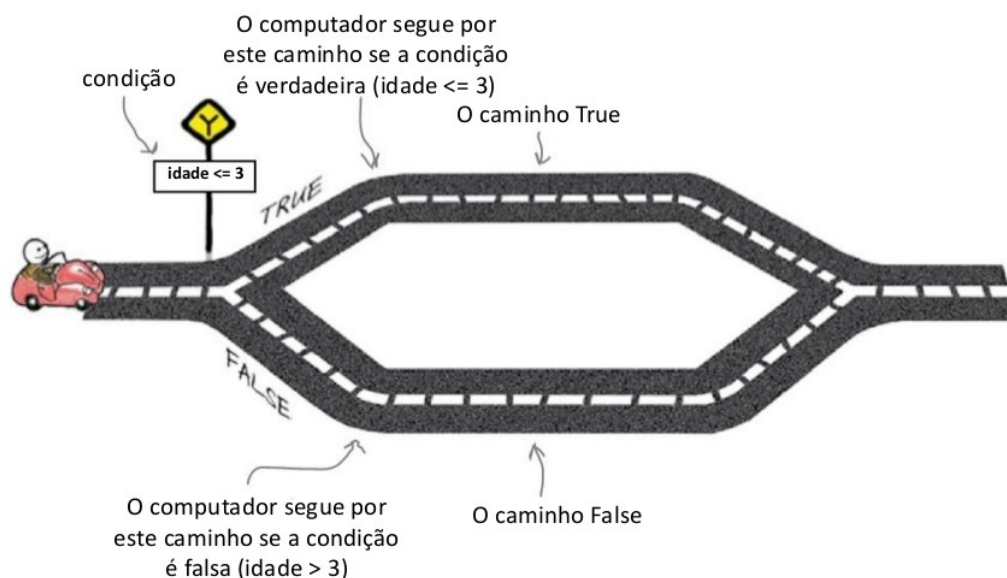
```

Note que para imprimir o caracter de porcentagem em Python é preciso digitar %%, uma vez que % é reconhecido como um caracter especial da linguagem.

“Aprender é a única coisa de que a mente nunca se cansa, nunca tem medo e nunca se arrepende.”  
(Leonardo da Vinci)

## Estruturas de seleção

Os programas desenvolvidos até o presente momento eram sempre compostos por sequências simples e lineares de comandos, no sentido de que em toda a execução todos os comandos serão executados. Em diversos problemas, é necessário tomar decisões antes de se executar um trecho de código. Por exemplo, para imprimir na tela uma mensagem que diz 'Carro novo', primeiramente devemos verificar se a idade do carro é menor ou igual a 3 anos. Caso contrário, gostaríamos de imprimir a mensagem 'Carro usado'. Isso significa que dependendo do valor da variável idade, um de dois caminhos possíveis será escolhido.



```
idade = int(input("Digite a idade de seu carro: "))
if idade <= 3:
    print("Seu carro é novo")
else:
    print("Seu carro é velho")
```

Ex6: Dados o sexo (M ou F) e a altura (em metros) de uma pessoa, informe o seu peso ideal, obtido através das seguintes fórmulas:

- para homens:  $72.7 \times \text{altura} - 58$
- para mulheres:  $62.1 \times \text{altura} - 44.7$

```
sexo = input('Entre com o sexo (M ou F): ')
altura = float(input('Entre com a altura (m): '))

if sexo == 'M':
    peso_ideal = 72.7*altura - 58
else:
    peso_ideal = 62.1*altura - 44.7

print('O peso ideal é %.2f kg' % peso_ideal)
```

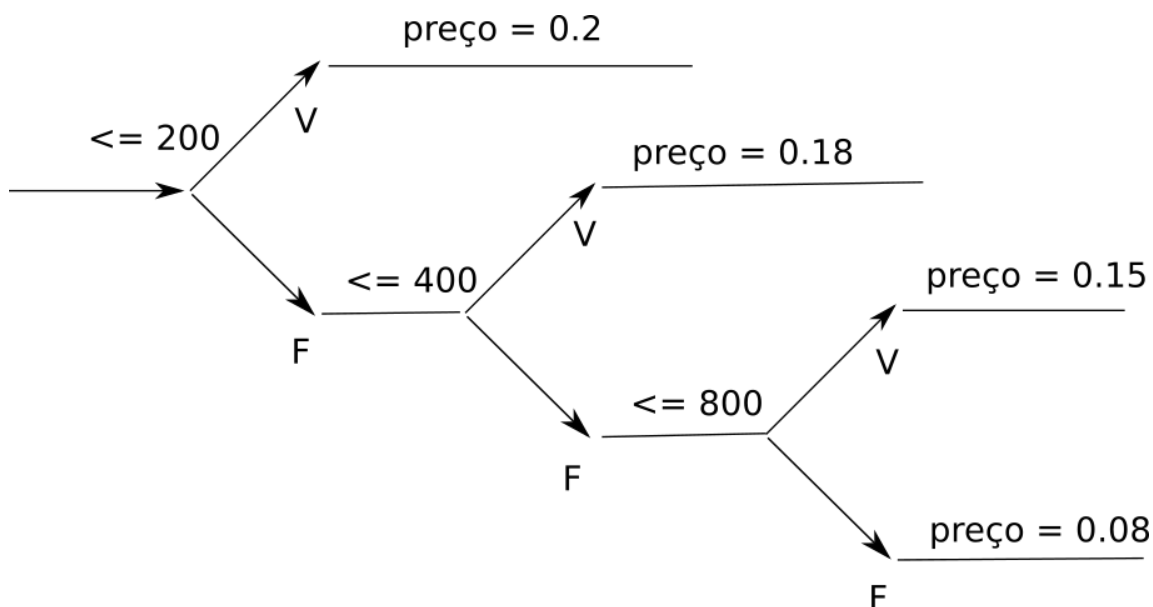
Em diversas ocasiões, mesmo após fazermos uma escolha, é preciso tomar outra decisão dentro da primeira e assim dividir o fluxo de execução de um programa em mais de duas opções. Construções assim recebem o nome de estruturas de seleção aninhadas, e são bastante comuns na prática.

Ex7: Considere a empresa de telefonia Tchau. Abaixo de 200 minutos a empresa cobra R\$ 0.20 por minuto. Entre 200 e 400 minutos, o preço é de R\$ 0.18, entre 400 e 800 minutos, o preço é de R\$ 0.15 e acima de 800 minutos, o preço é R\$ 0.08. Faça um programa para calcular o valor da sua conta de telefone, dado como entrada o total de minutos utilizados.

```
minutos = int(input("Minutos utilizados: "))
if minutos < 200:
    preço = 0.20
else:
    if minutos <= 400:
        preço = 0.18
    else:
        if minutos <= 800:
            preço = 0.15
        else:
            preço = 0.08

print("Conta telefônica: R$%6.2f" % (minutos * preço))
```

A representação do fluxo do programa é dada pelo diagrama a seguir. Note que há diversos caminhos possíveis para a execução de um mesmo programa dependendo dos dados de entrada.



Um detalhe importante que merece ser destacado: note que em Python diferentemente da maioria das linguagens de programação, não há caracteres especiais para abertura e fechamento de blocos. Quem define os blocos em Python, é o nível de indentação, o recuo em relação a coluna zero. Ao iniciar um novo bloco, os comandos são deslocados para a direita com um Tab. Para indicar que o bloco terminou, voltamos para o nível de indentação anterior. Um erro comum a iniciantes em Python é misturar Tab e espaços no alinhamento dos comandos. O interpretador Python não aceita isso. Ou se usa apenas Tabs ou se usa apenas espaços.

Na linguagem Python, é possível utilizar a palavra-chave **elif** sempre que aparece um else seguido imediatamente por um if. Dessa forma, o programa anterior pode ser simplificado para o trecho de código a seguir.

```
minutos = int(input("Minutos utilizados: "))
if minutos < 200:
    preço = 0.20
elif minutos <= 400:
    preço = 0.18
elif minutos <= 800:
    preço = 0.15
else:
    preço = 0.08

print("Conta telefônica: R$%6.2f" % (minutos * preço))
```

A seguir são apresentados alguns exercícios sobre estruturas de seleção.

Ex8: Sabendo que a média final do curso é computada por

$$MF = 0.35 \times P1 + 0.35 \times P2 + 0.2 \times MT + 0.1 \times EX$$

onde P1 é a nota da primeira avaliação, P2 é a nota da segunda avaliação e NT é a nota dos trabalhos, faça um programa que leia P1, P2 e NT e imprima na tela a média final, bem como a situação do aluno: Reprovado, se a media final é inferior a 5; Recuperação, se a média final está entre 5 e 6 ou Aprovado, se a média final é maior ou igual a 6.

```
p1 = float(input('Entre com a nota da P1: '))
p2 = float(input('Entre com a nota da P2: '))
mt = float(input('Entre com a média dos trabalhos: '))
ex = float(input('Entre com a média dos exercícios: '))

media = 0.35*p1 + 0.35*p2 + 0.2*mt + 0.1*ex
print('A média final é %.2f' %media)

if media < 5:
    print('REPROVADO')
elif media >= 5 and media < 6:
    print('RECUPERAÇÃO')
else:
    print('APROVADO')
```

Ex9: O índice de massa corpórea (IMC) de uma pessoa é dado pelo seu peso (em quilogramas) dividido pelo quadrado de sua altura (em metros). Faça um programa que leia o peso e a altura de um indivíduo e imprima seu IMC, além da sua classificação, de acordo com a tabela a seguir.

IMC <= 18.5 → magro  
18.5 < IMC <= 25 → normal  
25 < IMC <= 30 → sobrepeso  
IMC >= 30 → obeso

```

peso = float(input('Entre com o peso (kg): '))
altura = float(input('Entre com a altura (m): '))

imc = peso/altura**2
print('IMC = %.2f' %imc)

if imc <= 18.5:
    print('MAGRO')
elif imc > 18.5 and imc <= 25:
    print('NORMAL')
elif imc > 25 and imc <= 30:
    print('SOBREPESO')
else:
    print('OBESO')

```

Ex10: Uma equação do 2º grau é definida na sua forma padrão por  $ax^2+bx+c=0$  . As soluções de uma equação desse tipo são dadas pela conhecida fórmula de Bhaskara:

$$x_1 = \frac{-b - \sqrt{\Delta}}{2a} \text{ e}$$

$$x_2 = \frac{-b + \sqrt{\Delta}}{2a} \text{ , onde } \Delta = b^2 - 4ac$$

Faça um programa em que, dados os valores de a, b e c, imprima na tela as soluções reais da equação do segundo grau. Note que a deve ser diferente de zero e  $\Delta$  não pode ser negativo. Note que caso  $\Delta$  seja igual a zero, a solução é única.

```

# resolve uma simples equação do segundo grau
print('Entre com os coeficientes A, B e C da equação de segundo grau')
a = float(input('Entre com o valor de A: '))
b = float(input('Entre com o valor de B: '))
c = float(input('Entre com o valor de C: '))

if a == 0:
    print('Não é equação do segundo grau')
else:
    delta = b**2 - 4*a*c
    if delta < 0:
        print('Não há soluções reais para equação')
    elif delta == 0:
        x = -b/2*a
        print('Solução única x = %.3f' %x)
    else:
        x1 = (-b - delta**(0.5))/2*a
        x2 = (-b + delta**(0.5))/2*a
        print('x1 = %.3f e x2 = %.3f' %(x1, x2))

```

Ex11: Uma fruteira está vendendo frutas com a seguinte tabela de preços:

	Até 5 Kg	Acima de 5 Kg
Morango	R\$ 2,50 por Kg	R\$ 2,20 por Kg
Maçã	R\$ 1,80 por Kg	R\$ 1,50 por Kg

Se o cliente comprar mais de 8 Kg em frutas ou o valor total da compra ultrapassar R\$ 25,00, receberá ainda um desconto de 10% sobre este total. Escreva um algoritmo para ler a quantidade (em Kg) de morangos e a quantidade (em Kg) de maçãs adquiridas e escreva o valor a ser pago pelo cliente.

```
morango = float(input('Entre com a quantidade de morangos em Kg: '))
maca = float(input('Entre com a quantidade de macas em Kg: '))
print()    # pula linha

# Define o preço por Kg
if morango <= 5.0:
    preco_morango = 2.5
else:
    preco_morango = 2.2

if maca <= 5.0:
    preco_maca = 1.8
else:
    preco_maca = 1.5

# Calcula o valor total
valor_total = morango*preco_morango + maca*preco_maca

# Aplica o desconto se for o caso
if morango + maca > 8 or valor_total > 25:
    valor_total = 0.9*valor_total

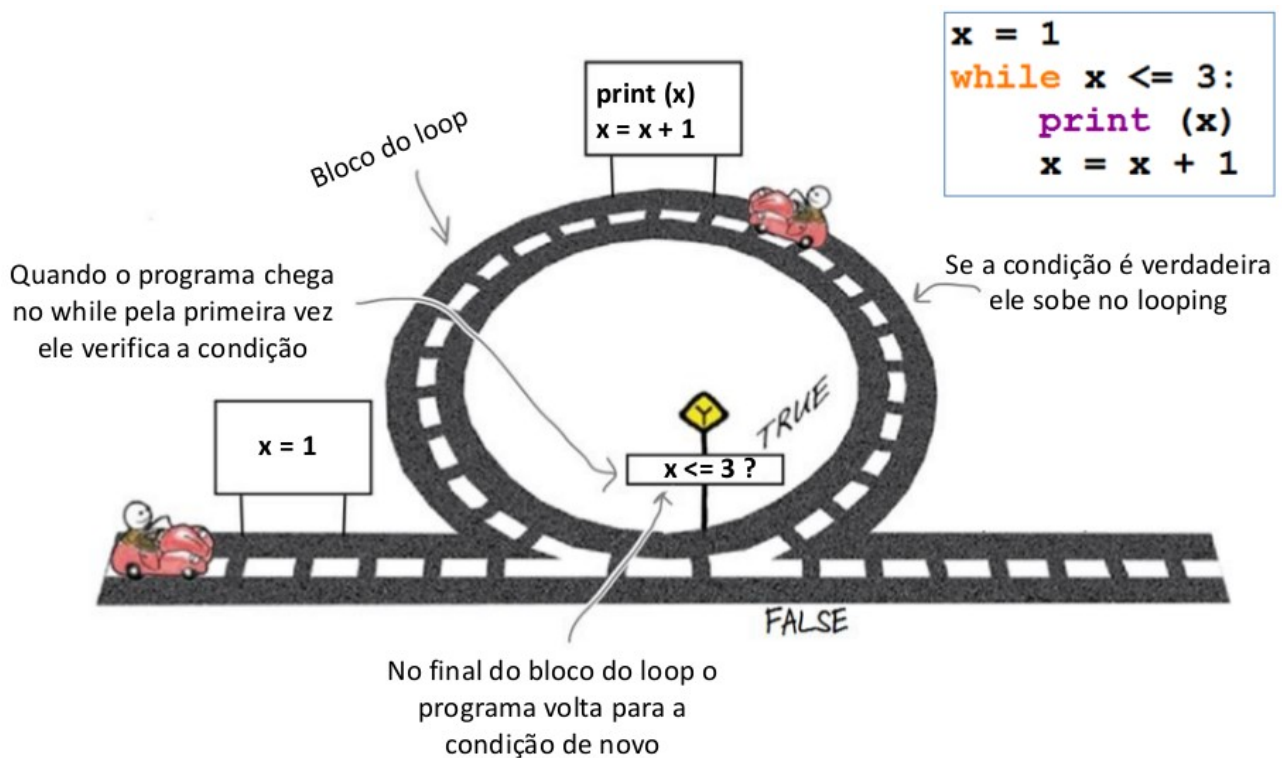
print('Total a pagar = R$ %.2f' %valor_total)
```

"Work for a cause, not for applause. Live life to express and not to impress. Don't strive to make your presence noticed, just make your absence felt."  
(Author Unknown)



## Estruturas de repetição

As estruturas de seleção são ferramentas importantes para controlar o fluxo de execução de programas, porém são limitadas no sentido que nos permitem apenas ao programa ignorar blocos de instruções indesejados. Um outro tipo de estrutura muito importante e que nos permite a execução de um trecho de código diversas vezes são as estruturas de repetição. Suponha por exemplo que desejamos imprimir os números de 1 até 3 na tela. Ao invés de repetir três vezes o mesmo comando na tela, podemos simplesmente usar um comando de repetição. Uma maneira de implementar estruturas de repetição em Python é através do comando **while**. Considere o exemplo a seguir.



A variável  $x$  inicia com valor 1. Na primeira passagem pelo loop, compara-se  $x$  com o valor 3 e verifica-se que  $1 \leq 3$ . Assim, como a condição é verdadeira, entra-se no laço. Dentro do laço imprime-se  $x$  na tela (será mostrado o valor 1) e em seguida  $x$  é incrementado em uma unidade, fazendo com que seu valor seja 2. O fluxo então retorna para a condição da repetição, onde a variável  $x$  é novamente compara com o valor 3, sendo verificado que  $2 \leq 3$ , e portanto mais uma vez iremos entrar no loop, imprimindo o valor 2 na tela e incrementando  $x$  novamente. Novamente, retornamos a condição do while onde compara-se  $x$  com o valor 3. Como  $3 \leq 3$ , entra-se de novo no loop, imprimindo o valor 3 na tela e incrementando o valor de  $x$  para 4. Por fim, ao voltarmos para a condição do loop, verifica-se que 4 não é menor ou igual a 3 (retorna Falso), e portanto não entra-se no loop. O comando while se encerra e o fluxo de execução é dirigido para a linha subsequente ao último comando do loop.

Ex11: Faça um programa que imprima na tela todos os números pares de zero até um limite  $N$  definido pelo usuário e lido como entrada.

```
fim = int(input("Digite o último número: "))
x = 0
while x <= fim:
    if x % 2 == 0:
        print (x)
    x = x + 1
```

Um contador é uma variável que é incrementada em uma unidade a cada passagem pelo laço (a variável `x` no exemplo anterior é um exemplo). Seu objetivo é contar quantas vezes a repetição está sendo realizada. Um outro tipo de variável muito importante em repetições são as variáveis chamadas de acumuladoras. A diferença entre um contador e um acumulador é que nos contadores o valor adicionado a cada passo é constante (em geral é uma unidade mas pode uma constante `c`), enquanto que nos acumuladores o valor adicionado é variável (muda de um passo para outro).

Ex12: Faça um programa que compute a soma dos `N` primeiros inteiros, onde `N` é fornecido como entrada.

```
# computa o somatório dos números de 1 até N
N = int(input('Entre com o valor de N: '))

i = 1
soma = 0

while i <= N:
    soma = soma + i
    i = i + 1

print('Somatório = %d' %soma)
```

Ex13: Faça um programa que compute o fatorial de um inteiro `N`, fornecido pelo usuários

```
i = 1
fat = 1
n = int(input("Digite n: "))
while i <= n:
    fat = fat * i
    i = i + 1
print ("Fat(%d) = %d" %(n, fat))
```

Existe um outro comando da linguagem Python para implementar repetição contada, isto é, repetir o loop um número pré-determinado de vezes. Trata-se do comando **for**, cuja sintaxe é dada por:

```
for i in range(n):
    print(i)
```

Por exemplo, o mesmo trecho usando o comando `while` ficaria:

```
i = 0
while i < n:
    print(i)
    i = i + 1
```

Um bom exercício consiste em reescrever os códigos feitos com o comando `while` utilizando o comando `for` e vice-versa. Isso ajuda na familiarização com as estruturas de repetição. A seguir veremos uma série de problemas e exercícios que envolvem estruturas de repetição.

Ex14: A sequência de Fibonacci é um padrão muito conhecido por ser recorrente em diversos fenômenos da natureza ([https://pt.wikipedia.org/wiki/Sequ%C3%Aancia\\_de\\_Fibonacci](https://pt.wikipedia.org/wiki/Sequ%C3%Aancia_de_Fibonacci)). Essa sequência é definida como segue:

$$F(1) = 1$$

$$F(2) = 1$$

$$F(n) = F(n-1) + F(n-2), \text{ para } n > 2$$

a) Faça um programa que compute o n-ésimo termo da sequência de Fibonacci

b) Sabe-se que no limite (n grande) a razão  $F(n)/F(n-1)$  converge para a proporção áurea. Use o programa desenvolvido com  $n = 1000$  para aproximar essa proporção, dada por  $\frac{1+\sqrt{5}}{2}$ .

```
n = int(input('Entre com um inteiro (n > 0): '))

a = 1
b = 1

if n == 1 or n == 2:
    print('F(%d) = %d' %(n, b))
else:
    for i in range(3, n+1):
        a, b = b, a+b

    print('F(%d) = %d' %(n, b))

print('A proporção áurea é aproximadamente %f' %b/a)
```

Ex15: A série harmônica é definida como o somatório de 1 até n de 1 sobre n, ou seja:

$$H = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$$

Faça um script que compute o valor de H dado um valor inteiro n de entrada.

```
n = int(input('Entre com o número de termos: '))

# Inicialização de variáveis
i = 1
soma = 0
while i <= n:
    soma = soma + 1/i
    i += 1

print('A soma dos %d termos da série vale %f' %(n, soma))
```

Ex16: Séries definidas por somatórios matemáticos são utilizadas para computar e aproximar diversas quantidades importantes. Para cada uma das séries abaixo, faça um programa que compute seu valor.

a)

$$S = \frac{1}{1} + \frac{3}{2} + \frac{5}{3} + \frac{7}{4} + \dots + \frac{99}{50}$$

```

n = int(input('Entre com o número de termos: '))
soma = 0
num = 1
den = 1

while den <= n:
    soma = soma + num/den
    num = num + 2
    den = den + 1

print(soma)

```

b)

$$W = \frac{1}{1} - \frac{2}{4} + \frac{3}{9} - \frac{4}{16} + \frac{5}{25} - \frac{6}{36} + \dots + \frac{10}{100}$$

```

n = int(input('Entre com o número de termos: '))
soma = 0
num = 1
den = 1
sinal = 1
while num <= n:
    soma = soma + sinal*num/den
    num = num + 1
    den = num**2
    sinal = -1*sinal

print(soma)

```

c)

$$S = \frac{1000}{1} - \frac{997}{2} + \frac{994}{3} - \frac{991}{4} + \dots$$

```

n = int(input('Entre com o número de termos: '))
soma = 0
num = 1000
den = 1
sinal = 1
while den <= n:
    soma = soma + sinal*num/den
    num = num - 3
    den = den + 1
    sinal = -1*sinal

print(soma)

```

Ex17: A constante matemática pi desempenha um papel importante nas mais variadas áreas da ciência. Existem diversas séries que aproximam o valor real dessa constante e podem ser usadas para gerar aproximações computacionais para esse número irracional. Três dessas formas são apresentadas a seguir. Faça 3 programas que computam aproximações para a constante pi, sendo um para cada série em questão.

$$a) \pi = 4 \sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1} = 4 \left( 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \right)$$

$$b) \pi = \sqrt{\sum_{i=1}^{\infty} \frac{6}{i^2}} = \sqrt{6 + \frac{6}{2^2} + \frac{6}{3^2} + \frac{6}{4^2} + \frac{6}{5^2} + \dots}$$

$$c) \pi = 4 \times \frac{2}{3} \times \frac{4}{3} \times \frac{4}{5} \times \frac{6}{5} \times \frac{6}{7} \times \frac{8}{7} \dots$$

Letra a)

```
import math

n = int(input('Entre com o número de termos da série: '))
print()    # pula linha

soma = 0

for i in range(n+1):
    termo = (-1)**i/(2*i+1)
    soma = soma + termo

pi = 4*soma

print('Aproximação de pi com %d iterações: %.10f' %(n, pi))
print('Valor exato de pi: %.10f' %math.pi)

# Calcula o erro de aproximação
erro = abs(math.pi - pi)

print('Erro de aproximação: %.10f' %erro)
```

Letra b)

```
import math

n = int(input('Entre com o número de termos da série: '))
print()

soma = 0

for i in range(1, n+1):
    termo = 6/i**2
    soma = soma + termo
```

```

pi = math.sqrt(soma)

print('Aproximação de pi com %d iterações: %.10f' %(n, pi))
print('Valor exato de pi: %.10f' %math.pi)

# Calcula o erro de aproximação
erro = abs(math.pi - pi)

print('Erro de aproximação: %.10f' %erro)

```

Letra c)

```

import math
import time

n = int(input('Entre com o número de termos da série: '))
print()

numerador = 2
denominador = 3

produto = numerador/denominador

inicio = time.time()

for i in range(1, n+1):
    if i % 2 == 1: # verifica se é ímpar
        numerador = numerador + 2
    else:
        denominador = denominador + 2

    produto = produto*(numerador/denominador)

pi = 4*produto

fim = time.time()

print('Tempo de execução: %.3f s' %(fim - inicio))
print('Aproximação de pi com %d iterações: %.10f' %(n, pi))
print('Valor exato de pi: %.10f' %math.pi)

# Calcula o erro de aproximação
erro = abs(math.pi - pi)

print('Erro de aproximação: %.10f' %erro)

```

Ex18: Uma das maneiras de se conseguir calcular a raiz quadrada de um número (que seja quadrado perfeito) é subtrair dele os números ímpares consecutivos a partir de 1, até que o resultado seja menor ou igual a zero. O número de vezes que se conseguir fazer as subtrações é a raiz quadrada.

Considere como entrada o número 16

Devemos subtrair de 16 os números ímpares maiores que 1 até chegar em zero ou num número negativo.

- 1)  $16 - 3 = 13$
- 2)  $13 - 5 = 8$
- 3)  $8 - 7 = 1$
- 4)  $1 - 9 = -8$  (ficou negativo)

Logo, a raiz quadrada de 16 é 4

Faça um algoritmo que calcule a raiz quadrada de dado numero inteiro (que seja quadrado perfeito) conforme essa regra.

```
n = int(input('Entre com um inteiro quadrado perfeito: '))

j = n    # para não perder o valor de n
i = 3    # contador para os números ímpares
k = 0    # o número de vezes que a subtração ocorre

while True:
    j = j - i    # subtrai i
    k = k + 1    # e incrementa o valor de k
    if j <= 0:   # se for menor ou igual a zero, pare
        break
    i = i + 2    # se não entrar no if, passa para próximo número ímpar

print('Raíz inteira de %d é %d' %(n, k))
```

Ex19: Os números primos possuem várias aplicações dentro da Computação, por exemplo na Criptografia. Um número primo é aquele que é divisível apenas por um e por ele mesmo. Faça um programa que peça um número inteiro e determine se ele é ou não um número primo. Lembre-se que um número  $n$  é primo se for divisível apenas por 1 e ele mesmo.

```
n = int(input('Entre com um inteiro: '))
print()

divisor = 2
primo = True

# // representa a divisão inteira
while (divisor < n//2 ) and (primo):
    if (n % divisor == 0):
        primo = False
    else:
        divisor = divisor + 1

if primo:
    print('O número %d é primo' %n)
else:
    print('O número %d não é primo' %n)
```

Ex20: Faça um programa que leia um nome de usuário e a sua senha e não aceite os dados de login diferentes aos valores pré-definidos, mostrando uma mensagem de erro e voltando a pedir as informações. Considere como username seu primeiro nome e a senha como sendo kawabunga.

```
user = 'alex'
password = 'kawabunga'

while True:
    login = input('Login: ')
    senha = input('Senha: ')

    if login == user and senha == password:
        break
    else:
        print('Username ou senha incorretos... Tente novamente.')

print('Senha OK! Login realizado com sucesso!')
```

Ex21: Faça um programa que peça para n pessoas a sua idade, ao final o programa devera verificar se a média de idade da turma varia entre 0 e 25,26 e 60 e maior que 60; e então, dizer se a turma é jovem, adulta ou idosa, conforme a média calculada.

```
n = int(input('Entre com o número de pessoas na turma: '))

soma = 0

for i in range(n):
    idade = int(input('Entre com a idade da pessoa %d: ' %(i+1)))
    soma = soma + idade

media = soma/n

print('A média de idade da turma é de %.2f anos' %media)

if media <= 25:
    print('A turma é jovem!')
elif media > 25 and media <= 60:
    print('A turma é adulta!')
else:
    print('A turma é idosa!')
```

Ex22: O Departamento Estadual de Meteorologia lhe contratou para desenvolver um programa que leia as um conjunto indeterminado de temperaturas, e informe ao final a menor e a maior temperaturas informadas, bem como a média das temperaturas. Digite -999 para interromper o processo de leitura das temperaturas.

```
n = 0
soma = 0
minimo = 100    # o mínimo deve iniciar com valor bem alto
maximo = -100   # o máximo deve iniciar com valor bem pequeno

while True:
    temperatura = float(input('Entre com a temperatura em Celsius: '))
```



```

n = n + 1    # quantas temperaturas já foram lidas

if temperatura == -999:
    break

soma = soma + temperatura

if temperatura < minimo:
    minimo = temperatura
elif temperatura > maximo:
    maximo = temperatura

media = soma/n

print('Temperatura mínima: %.2f' %minimo)
print('Temperatura máxima: %.2f' %maximo)
print('Temperatura média: %.2f' %media)

```

Ex23: Pode-se mostrar que a seguinte relação de recorrência pode ser utilizada para computar numericamente uma aproximação para a raiz quadrada de  $a$ .

$$x_{k+1} = \frac{1}{2} \left( x_k + \frac{a}{x_k} \right)$$

onde  $x_0 = 1$  e  $a$  é o número cuja raiz é desejada. Faça uma função que receba como entrada um número  $a$  e retorne como saída sua raiz quadrada.

```

# calcula a raiz quadrada de um número usando o método de Newton
a = float(input('Entre com o valor de a: '))

x = 1

while True:
    x_novo = 0.5*(x + a/x)
    erro = abs(x - x_novo)
    print('x : %.10f ***** Erro: %.10f' %(x_novo, erro))
    x = x_novo

    if erro < 10**(-8):
        break

print('A raiz quadrada de %.3f é %f' %(a, x))

```

Ex24: O método de Newton é um algoritmo numérico para calcular a raiz (zero) de uma função qualquer. Ele é baseado na seguinte iteração:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

onde  $x_0$  pode ser escolhido arbitrariamente e  $f'(x)$  denota a derivada da função  $f$ . Após um certo número de passos, a diferença entre  $x_{k+1}$  e  $x_k$  em módulo tende a zero, o que significa que o algoritmo convergiu para a solução desejada. Implemente o método de Newton para encontrar a raiz da função a seguir:

$$f(x) = e^x - 2\cos(x)$$

Como a derivada da exponencial é a própria exponencial e a derivada do seno é menos cosseno, temos:

$$f'(x) = e^x + 2\sin(x)$$

O código a seguir ilustra uma implementação em Python.

```
from math import exp # importa a função exponencial do pacote math
from math import sin # importa a função seno do pacote math
from math import cos # importa a função cosseno do pacote math

# Define as funções matemáticas com lambda functions
f = lambda x : exp(x) - 2*cos(x)
df = lambda x : exp(x) + 2*sin(x)

x = float(input('Entre com o chute inicial (x0): '))

while True:
    novo_x = x - f(x)/df(x)
    erro = abs(x - novo_x)
    print('x : %.10f ***** Erro: %.10f' %(novo_x, erro))
    x = novo_x

    if erro <= 10**(-8):
        break

print('Raiz da função: %f' %x)
```

"Winning doesn't always mean being first. Winning means you're doing better than you've done before."  
(Bonnie Blair)

## Modularização e funções

Assim como aprendemos a utilizar diversas funções da linguagem Python como `print()`, `input()` e `range()`, iremos aprender a criar nossas próprias funções. A ideia consiste em empacotar um trecho de código para reutilizá-lo sempre que necessário. Em Python criamos uma função com a palavra reservada **def**. Para indicar o valor de retorno da função utilizamos a palavra-chave **return**. Por exemplo, suponha que desejamos empacotar numa função o código que computa o fatorial de N.

```
def fat(n):  
    f = 1  
    while n > 0:  
        f = f * n  
        n = n - 1  
    return f
```

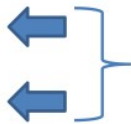
Basta chamar a função com o argumento desejado:

```
fat(5)  
fat(10)
```

### Variáveis globais e locais


Variáveis declaradas dentro de uma função possuem escopo local, ou seja, não podem ser acessadas fora da função. Ao se usar o mesmo nome para designar variáveis dentro e fora da função, elas são objetos distintos.

```
a = 5  
def muda_e_imprime():  
    a = 7  
    print('a dentro da função: %d' %a)  
print('a antes de mudar: %d' %a)  
muda_e_imprime()  
print('a depois de mudar: %d' %a)  
  
>>>  
a antes de mudar: 5  
a dentro da função: 7  
a depois de mudar: 5
```



Para indicar que as duas variáveis são de fato a mesma, é necessário utilizar a palavra-chave **global**.

```
a = 5  
def muda_e_imprime():  
    global a  
    a = 7  
    print('a dentro da função: %d' %a)  
print('a antes de mudar: %d' %a)  
muda_e_imprime()  
print('a depois de mudar: %d' %a)  
  
>>>  
a antes de mudar: 5  
a dentro da função: 7  
a depois de mudar: 7
```



A seguir serão apresentados alguns exemplos de problemas envolvendo estruturas de repetição e funções.

Ex25: Faça um função que retorne o n-ésimo termo da sequência de Fibonacci

```
# gera o n-ésimo termo da série de Fibonacci
def fibonacci(n):
    a = 1
    b = 1

    if n == 1 or n == 2:
        nt('F(%d) = %d' %(n, b))
    else:
        for i in range(3, n+1):
            a, b = b, a+b

    return b

# Início do script
n = int(input('Entre com um inteiro (n > 0): '))
numero = fibonacci(n)
print('F(%d) = %d' %(n, numero))
```

Ex26: A função seno pode ser aproximada numericamente pela seguinte série

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \quad \text{for all } x$$

Faça um programa que aceite como entrada o valor do ângulo x em radianos e o número de termos n da série e imprima na tela o valor computado de sen(x).

```
import math

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

def seno(x, n):
    soma = 0
    for i in range(0, n+1):
        soma = soma + (-1)**i*x**(2*i+1)/fatorial(2*i+1)
    return soma

x = float(input('Entre com o valor do angulo (radianos) : '))
n = int(input('Entre com o número de termos da série (n): '))

print('sen(%f) = %.9f' %(x, seno(x, n)))
print('Valor exato: %.9f' %(math.sin(x)))
```

Ex27: A função cosseno pode ser aproximada numericamente pela seguinte série

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots \quad \text{for all } x$$

Faça um programa que aceite como entrada o valor do ângulo x em radianos e o número de termos n da série e imprima na tela o valor computado de cos(x).

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

def cosseno(x, n):
    soma = 0
    for i in range(0, n+1):
        soma = soma + (-1)**i*x**(2*i)/fatorial(2*i)
    return soma

x = float(input('Entre com o valor do angulo (radianos) : '))
n = int(input('Entre com o número de termos da série (n): '))

print('sen(%.1f) = %f' %(x, cosseno(x, n)))
```

Ex28: A função exponencial pode ser aproximada numericamente pela seguinte série

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

Faça um programa que aceite como entrada o valor do ângulo x em radianos e o número de termos n da série e imprima na tela o valor computado de exp(x).

```
import math

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

def exponencial(x, n):
    soma = 0
    for i in range(0, n+1):
        soma = soma + (x**i)/fatorial(i)
    return soma

x = float(input('Entre com o valor do expoente (x) : '))
n = int(input('Entre com o número de termos da série (n): '))

print('exp(%.1f) = %.9f' %(x, exponencial(x, n)))
print('Valor exato: %.9f' %(math.exp(x)))
```

Ex29: Um número de Keith é um inteiro maior que 9, tal que os seus dígitos, ao serem usados para iniciar uma sequência de Fibonacci, alcançam posteriormente o referido número. Um exemplo é o número 14, pois  $1 + 4 = 5$ ,  $4 + 5 = 9$  e  $5 + 9 = 14$ . Pergunta-se: quantos e quais são os números de Keith menores que 100? Faça um programa para determinar a resposta.

```
# Função que retorna True se a sequência iniciada com a e b atinge num
def fib(a, b, num):
    converge = False
    while b <= num:
        a, b = b, a+b
        if b == num:
            converge = True
    return converge

# Computa quais são os números de Keith de 10 até 99
qtde = 0
for num in range(10, 100):
    a = num // 10    # pega a dezena
    b = num % 10     # pega a unidade
    if fib(a, b, num):
        qtde += 1
    print('Número de Keith %d: %d' %(qtde, num))
```

Ex30: Pelo método de Newton, pode-se mostrar que a seguinte relação de recorrência pode ser utilizada para computar numericamente uma aproximação para a raiz quadrada de a.

$$x_{k+1} = \frac{1}{2} \left( x_k + \frac{a}{x_k} \right)$$

onde  $x_0=1$  e a é o número cuja raiz é desejada. Faça uma função que receba como entrada um número a e retorne como saída sua raiz quadrada.

```
def calcula_raiz(a):
    x = 1
    while True:
        x_novo = 0.5*(x + a/x)
        erro = abs(x - x_novo)
        print('x : %.10f ***** Erro: %.10f' %(x_novo, erro))
        x = x_novo

        if erro < 10**(-8):
            break

    return x

# Início do script
a = float(input('Enter com o valor de a: '))
raiz = calcula_raiz(a)
print('A raiz quadrada de %.3f é %f' %(a, raiz))
```

Ex31: Números primos são muito importantes em diversas aplicações que vão desde fatoração de números inteiros até criptografia de dados. Faça um programa que compute a soma de todos números primos menores que N, onde N é fornecido como entrada.

- a) Compute o valor da soma e o tempo gasto para computá-la se N=1000
- b) Compute o valor da soma e o tempo gasto para computá-la se N=10000
- c) Compute o valor da soma e o tempo gasto para computá-la se N=100000

```
import time

# verifica se inteiro n é primo
def verifica_primo(n):
    divisor = 2
    primo = True
    # o número 1 não é primo!
    if n == 1:
        return False
    # // representa a divisão inteira
    while (divisor <= n//2 ) and (primo):
        if (n % divisor == 0):
            primo = False
        else:
            divisor = divisor + 1
    return primo

# Início do programa
N = int(input('Entre com o valor de N: '))

inicio = time.time()
soma = 0
for i in range(2, N):
    # Só precisa testar se é primo se i for ímpar
    if verifica_primo(i):
        soma = soma + i

print('A soma dos primos menores que %d é %d' %(N,soma))
fim = time.time()
tempo = fim - inicio
print('Tempo: %f segundos' %tempo)
```

Ex32: Implemente uma função para calcular a raiz de uma função f(x) utilizando o método de Newton. Lembre-se que a iteração é dada por:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

onde  $x_0$  pode ser escolhido arbitrariamente e  $f'(x)$  denota a derivada da função f. Após um certo número de passos, a diferença entre  $x_{k+1}$  e  $x_k$  em módulo tende a zero, o que significa que o algoritmo convergiu para a solução desejada. Implemente o método de Newton para encontrar a raiz da função a seguir:

$$f(x) = e^x - 2 \cos(x)$$

A função deve receber como parâmetros a solução inicial, a função f e sua derivada df.

```

from math import exp      # importa a função exponencial do pacote math
from math import sin      # importa a função seno do pacote math
from math import cos      # importa a função cosseno do pacote math

# encontra a raiz da função f usando chute inicial x e derivada df
def newton(x, f, df):
    while True:
        novo_x = x - f(x)/df(x)
        erro = abs(x - novo_x)
        print('x : %.10f ***** Erro: %.10f' %(novo_x, erro))
        x = novo_x

        if erro <= 10**(-8):
            break

    return x

# Define as funções matemáticas com lambda functions
f = lambda x : exp(x) - 2*cos(x)
df = lambda x : exp(x) + 2*sin(x)

x0 = float(input('Entre com o chute inicial (x0): '))
raiz = newton(x0, f, df)
print('Raiz da função: %f' %raiz)

```

## Aplicação interessante

O número de um CPF tem 9 algarismos e mais dois dígitos verificadores, que são indicados após uma barra. Logo, um CPF tem 11 algarismos. O número do CPF é escrito na forma ABCDEFGHI-JK ou diretamente como ABCDEFGHIJK, onde os algarismos não podem ser todos iguais entre si.

O J é chamado 1º dígito verificador do número do CPF.

O K é chamado 2º dígito verificador do número do CPF.

### Primeiro Dígito

Para obter J multiplicamos A, B, C, D, E, F, G, H e I pelas constantes correspondentes:

A x 10, B x 9, C x 8, D x 7, E x 6, F x 5, G x 4, H x 3, I x 2

O resultado da soma,  $S = (10A + 9B + 8C + 7D + 6E + 5F + 4G + 3H + 2I)$ , é dividido por 11. Analisamos então o RESTO dessa divisão:

Se for 0 ou 1, o dígito J é 0 (zero). Se for 2, 3, 4, 5, 6, 7, 8, 9 ou 10, o dígito J é (11 - RESTO)

### Segundo Dígito

Já temos J. Para obter K multiplicamos A, B, C, D, E, F, G, H, I e J pelas constantes:

A x 11, B x 10, C x 9, D x 8, E x 7, F x 6, G x 5, H x 4, I x 3, J x 2



O resultado da soma,  $S = 11A + 10B + 9C + 8D + 7E + 6F + 5G + 4H + 3I + 2J$ , é dividido por 11. Verificamos então o RESTO dessa divisão: se for 0 ou 1, o dígito K é 0 (zero). Se for 2, 3, 4, 5, 6, 7, 8, 9 ou 10, o dígito K é (11 - RESTO).

Faça uma função em Python que recebe os 9 primeiros dígitos de um CPF e gere os 2 dígitos verificadores. Teste com o seu CPF.

```
# Os dígitos do CPF formam uma string (sequência de caracteres).
# É preciso converter cada dígito para inteiro para fazer os cálculos
def digito_verificador(cpf):
    ##### Cálculo do primeiro dígito
    soma = 0
    # Nesse laço, o valor de i é o multiplicador do dígito
    for i in range(10, 1, -1):
        k = 10 - i      # índice para os dígitos do CPF
        soma = soma + i*int(cpf[k])
    resto = soma % 11

    if resto == 0 or resto == 1:
        primeiro_digito = 0
    else:
        primeiro_digito = 11 - resto

    # Anexa o primeiro dígito no CPF
    # Soma de strings = concatenação: 'ab' + 'cd' = 'abcd'
    cpf = cpf + str(primeiro_digito)

    ##### Cálculo do segundo dígito
    soma = 0
    # Nesse laço, o valor de i é o multiplicador do dígito
    for i in range(11, 1, -1):
        k = 11 - i      # índice para os dígitos do CPF
        soma = soma + i*int(cpf[k])
    resto = soma % 11

    if resto == 0 or resto == 1:
        segundo_digito = 0
    else:
        segundo_digito = 11 - resto

    # Anexa o segundo dígito no CPF
    cpf = cpf + str(segundo_digito)

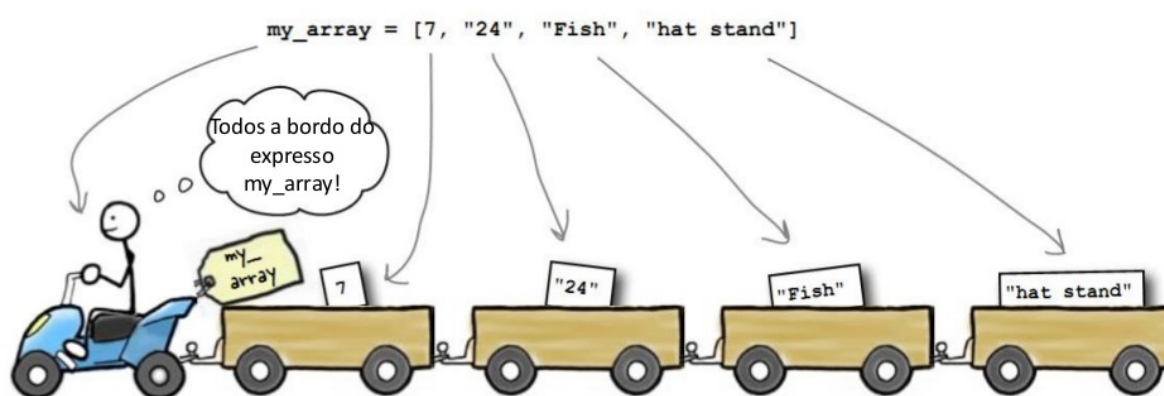
    return cpf

# Início do script
numeros = input('Entre com os 9 primeiros dígitos do CPF: ')
cpf = digito_verificador(numeros)
print('CPF: ', cpf)
```

"Start by doing what's necessary; then do what's possible; and suddenly you're doing the impossible."  
(Francis of Assisi)

## Variáveis compostas 1D: Listas, strings e vetores

Tuplas e listas são variáveis compostas, isto é, estruturas de dados em Python muito úteis para armazenar uma coletânea de objetos de forma indexada, ou seja, em cada posição da lista ou da tupla temos uma variável. A figura a seguir ilustra essa ideia como um trem de dados.



O trem de dados `my_array` é uma única variável

A única diferença entre uma tupla e uma lista, é que tuplas são objetos imutáveis, ou seja, uma vez criados não podem ser modificados. Em outras palavras, é impossível usar o operador de atribuição com uma tupla. As listas por outro lado podem ser modificadas sempre que necessário e portanto são muito mais utilizadas na prática do que as tuplas.

```
# cria uma lista vazia
lista = []
# cria uma lista com 3 notas
notas = [7.5, 9, 8.3]
# imprime a primeira nota
print(notas[0])
# mudando a primeira nota
notas[0] = 8.7
```

Veremos a seguir uma série de comandos da linguagem Python para manipulação de listas.

```
import random

# cria uma lista com 5 elementos
la = [1,2,3,4,5]
print('Lista la possui %d elementos' %len(la))
print(la)

input('Pressione qualquer tecla para continuar...')

lb = []
# cria uma lista com 5 números aleatórios entre 0 e 99
for i in range(5):
    lb.append(random.randint(0, 100))

print('Lista lb possui %d elementos aleatórios' %len(lb))
print(lb)
```

```

input('Pressione qualquer tecla para continuar...')

# concatena 2 listas e gera uma nova
lc = la + lb
print('Lista lc = la + lb possui %d elementos' %len(lc))
print(lc)

input('Pressione qualquer tecla para continuar...')

print('Percorre lista incrementando cada elemento')
for i in range(len(lc)):
    lc[i] += 1
print(lc)

input('Pressione qualquer tecla para continuar...')

# insere elementos no final da lista
lc.append(99)
lc.append(72)
print('Após inserção de 99 e 72 lc tem %d elementos' %len(lc))
print(lc)

input('Pressione qualquer tecla para continuar...')

# encontra o maior e o menor elementos da lista
print('Menor elemento de lc é %d' %min(lc))
print('Maior elemento de lc é %d' %max(lc))

# ordena a lista
print('Lista lc ordenada')
lc.sort()
print(lc)

```

Ex33: Faça um programa que leia uma lista de 5 números inteiros e mostre-a na tela.

```

vetor = []
i = 1
while i <= 5:
    n = int(input("Digite um número: "))
    vetor.append(n)
    i = i + 1
print ("Vetor lido:", vetor)

```

Ex34: Faça um programa que leia 4 notas, armazene-as em uma lista e mostre-as na tela juntamente com a média.

```

notas = []
soma = 0
i = 1
while i <= 4:
    n = float(input("Nota: "))
    notas.append(n)
    soma += n
    i += 1
print ("Notas:", notas)
print ("Média: %4.2f" %(soma/4))

```

## Slicing (Fatiamento)

Ao se trabalhar com listas em Python, uma ferramenta bastante poderosa para obter subconjuntos de uma lista é a técnica conhecida como *slicing*. Perguntas como quais são os elementos das posições pares, ou qual é a sublista obtida entre o terceiro e o sétimo elemento podem ser facilmente respondidas através do fatiamento. Suponha a lista  $L = [9, 8, 7, 6, 5, 4, 3, 2, 1]$ .

$L[-1]$  retorna o último elemento: 1

$L[-2]$  retorna o penúltimo elemento: 2

$L[-3]$  retorna o antepenúltimo elemento: 3

$L[:3]$  retorna: [9, 8, 7] (note que não inclui o limite superior)

$L[3:]$  retorna: [6, 5, 4, 3, 2, 1]

$L[2:5]$  retorna: [7, 6, 5]

$L[::2]$  retorna: [9, 7, 5, 3, 1] (apenas as posições pares)

$L[::-3]$  retorna: [9, 6, 3]

$L[::-1]$  retorna: [1, 2, 3, 4, 5, 6, 7, 8, 9] (lista ao contrário)

$L[::2]$  retorna: [1, 3, 5, 7, 9]

## List comprehensions

Suponha agora que desejamos criar uma nova lista  $L2$  que contenha somente os elementos de  $L$  que sejam divisíveis por 3. Uma forma rápida de construir  $L2$  é através da técnica list comprehensions.

```
L = [9, 8, 7, 6, 5, 4, 3, 2, 1]
L2 = [x for x in L if x % 3 == 0]
```

Por exemplo, se desejamos construir  $L3$  com os números pares, teremos:

```
L3 = [x for x in L if x % 2 == 0]
```

Suponha agora que desejamos criar  $L4$  com o quadrado dos números ímpares:

```
L4 = [x**2 for x in L if x% 2 == 1]
```

Veremos a seguir problemas e exercícios que utilizam estruturas do tipo lista.

Ex35: Supondo que uma lista de  $n$  elementos reais represente um vetor no  $R^n$ , faça uma função para computar o produto escalar entre 2 vetores.

```
# Calcula o produto escalar entre 2 vetores
def produto_escalar(u, v):
    soma = 0
    for i in range(len(u)):
        soma = soma + u[i]*v[i]
    return soma
```

Faça uma função que compute a norma de um vetor:  $\|\vec{v}\| = \sqrt{v_1^2 + v_2^2 + \dots + v_n^2}$

Ex36: Faça uma função que encontre o maior elemento de uma lista.

```
# Retorna o maior elemento de uma lista de inteiros
def maximo(lista):
    maior = lista[0]
    for i in range(len(lista)):
        if lista[i] > maior:
            maior = lista[i]
    return maior
```

Ex37: Uma tarefa fundamental na computação consiste em dado uma lista e um valor qualquer, verificar se aquele valor pertence a lista ou não. Essa funcionalidade é usada por exemplo em qualquer sistema que exige o login de um usuário (para verificar se o CPF da pessoa está cadastrada). Faça uma função que, dada uma lista de inteiros L e um número inteiro x, verifique se x está ou não em L. A função deve retornar o índice do elemento (posição) caso ele pertença a ele ou o valor lógico False se ele não pertence a L. (isso equivale ao operador in de Python)

```
def busca_sequencial(lista, x):
    achou = False
    i = 0
    while i < len(lista) and not achou:
        if (lista[i] == x):
            achou = True
            pos = i
        else:
            i = i + 1
    if achou:
        return pos
    else:
        return achou
```

Ex38: Deseja-se contar quantos números entre 1000 e 9999 (inclusive) são ímpares, divisíveis por 7 e cujo primeiro algarismo é um número par. Como resolver o problema em Python com listas?

```
def impar(n):
    if n % 2 == 1:
        return True

def div_por_7(n):
    if n % 7 == 0:
        return True

def inicia_par(s):
    num = int(s[0])
    if num % 2 == 0:
        return True

def armazena_numeros(low, high):
    L = []
    for i in range(low, high+1):
        s = str(i) # converte para string
        if impar(i) and div_por_7(i) and inicia_par(s):
            L.append(i)
    return L
```

```
# Início do script
a = int(input('Entre com o limite inferior do intervalo: '))
b = int(input('Entre com o limite superior do intervalo: '))

lista = armazena_numeros(a, b)

print(lista)
print('Há %d números satisfazendo as condições.' %(len(lista)))
```

Ex39: Faça uma função que receba como entrada um número inteiro na base 10 e retorne uma lista contendo sua representação em binário (cada elemento da lista é um bit 0 ou 1).

```
def decimal_binario(n):
    binario = []
    while n != 0:
        quociente = n // 2
        resto = n % 2
        binario.append(resto)
        n = quociente
    return binario[::-1]

# Início do script
n = int(input('Entre com um número inteiro: '))
b = decimal_binario(n)
print('O número %d em binário é: ' %n)
print(b)
```

Ex40: Dada a seguinte lista de 10 elementos  $L = [-8, -29, 100, 2, -2, 40, 23, -8, -7, 77]$  faça funções que recebam L como entrada e retorne:

- a) o menor número da lista
- b) o maior número da lista
- c) a média de todos os elementos da lista
- d) a lista inversa
- e) o desvio padrão dos elementos da lista

$$Desvio = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2}, \text{ onde } \bar{x} \text{ é a média e } n \text{ é o tamanho da lista}$$

O script a seguir mostra a solução em Python.

```
import math

def minimo(L):
    menor = L[0]
    for x in L:
        if x < menor:
            menor = x
    return menor
```

```

def maximo(L):
    maior = L[0]
    for x in L:
        if x > maior:
            maior = x
    return maior

def media(L):
    soma=0
    for x in L:
        soma = soma + x
    media = soma/len(L)
    return media

def desvio_padrao(L):
    m = media(L)
    soma = 0
    for x in L:
        soma = soma + (x - m)**2
    desvio = math.sqrt(soma/len(L))
    return desvio

def inverte(L):
    return L[::-1]

# Início do script
L = [-8, -29, 100, 2, -2, 40, 23, -8, -7, 77]
print('Mínimo: %d' %minimo(L))
print('Máximo: %d' %maximo(L))
print('Média: %f' %media(L))
print('Desvio padrão: %f' %desvio_padrao(L))
print('Inverso:', inverte(L))

```

## Strings

Strings são tipos de dados definidos como uma sequência de caracteres. Assemelham-se muito com listas, porém não é possível adicionar novos elementos ao conjunto (não posso fazer append, nem ordenar, nem inserir ou remover elemento). Por outro lado, há uma série de funções que strings possuem mas listas não. Como exemplo, podemos citar upper e lower para mudar os caracteres para maiúsculo ou minúsculo. Mas de longe a função mais conhecida e importante que apenas strings possuem é split (quebra em pedaços baseado num separador e monta uma lista). Como o próprio nome diz, a função split() recebe uma string como entrada e a quebra em diversos pedaços, retornando uma lista em que cada elemento é um pedaço da string original. Isso tudo de acordo com um caracter separador passado como parâmetro (caracteres típicos de separação são : ; , . - \_). Se nenhum caracter especial for passado como parâmetro, assume-se que devemos quebrar a string nos espaços em branco. Por exemplo:

```

digitos = '0 1 2 3 4 5 6 7 8'
lista_digitos = digitos.split()

digitos = '0,1,2,3,4,5,6,7,8'
lista_digitos = digitos.split(',')

```

O processo inverso, ou seja, pegar um lista de pedaços e juntar numa única string é feito com a função `join()`. O que essa função `join` faz nada mais é que pegar uma lista de strings e gerar uma única string

```
numeros = ''.join(lista_digitos)
numeros = ','.join(lista_digitos)
numeros = '-'.join(lista_digitos)
numeros = ':'.join(lista_digitos)
```

Na prática o que essa função faz é equivalente ao seguinte código:

```
lista = ['a','b','c','d','e','f']
g = ''
for x in lista:
    g = g + x    # concatena caracteres
```

Ex41: Saber quantas palavras distintas existe em uma frase é uma tarefa que pode ser bastante complicada em determinadas linguagens de programação. Em Python, isso pode ser feito facilmente. Veja o exemplo a seguir.

```
import string

def lista_palavras(frase):
    frase = frase.lower()    # tudo em minúscula
    # remove pontuação
    for c in string.punctuation:
        frase = frase.replace(c, '')
    # quebra string em lista de palavras
    lista = frase.split()
    palavras = []
    for x in lista:
        if x not in palavras:
            palavras.append(x)

    return palavras

# Início do script
sentenca = 'If you have an apple and I have an apple and we exchange
these apples then you and I will still each have one apple. But if you
have an idea and I have an idea and we exchange these ideas, then each
of us will have two ideas.'
print(sentenca)
print()

L = lista_palavras(sentenca)
print('Há %d palavras distintas na sentença!' %len(L))
print(L)
```

Ex42: Implemente uma função que recebe uma frase como entrada (string) e retorne quantas vogais ela possui, independente se maiúscula ou minúscula.



```

from unicodedata import normalize

def conta_vogais(palavra):
    # letras minúsculas
    p = palavra.lower()
    # remove acentos, til, cedilhas
    p = normalize('NFKD', p).encode('ascii', 'ignore').decode('ascii')
    vogais = 0
    for letra in p:
        if letra in 'aeiou':
            vogais = vogais + 1
    return vogais

palavra = input('Entre com uma palavra: ')
n = conta_vogais(palavra)
print('Há %d vogais na palavra %s' %(n, palavra))

```

Ex43: Implemente uma função que receba 2 palavras como entrada (strings) e retorne se elas são palíndromas, ou seja, se uma é o inverso da outra.

```

def palindroma(p1):
    palavra = p1.lower()
    if palavra == palavra[::-1]:
        return True
    else:
        return False

```

Ex44: Um histograma é a representação gráfica da distribuição de frequências de uma variável. Trata-se de uma ferramenta visual muito importante para visualizar a distribuição dos valores observados. Faça um programa que simule n jogadas de um dado não viciado. Armazene o resultado numa lista e gere um histograma dos valores observados. Encontre a moda, isto é o valor que mais vezes aconteceu nas jogadas.

```

import numpy as np

# joga um dado n vezes
def joga_dados(n):
    # Gera números aleatórios inteiros de 1 a 6
    jogadas = np.random.randint(1, 7, n)
    return jogadas

# constrói um histograma (distribuição de frequências)
def histograma(dados):
    h = 6*[0] # cria uma lista com 6 zeros
    for x in dados:
        h[x-1] = h[x-1] + 1 # incrementa posição do histograma
    return h

# Calcula a moda (número que mais aparece)
def moda(hist):
    maximo = hist[0]
    modas = []
    # Acha o máximo
    for x in hist:
        if x > maximo:
            maximo = x

```

```

# Marca as posições em que o máximo aparece
for i in range(len(hist)):
    if hist[i] == maximo:
        modas.append(i+1) # pois começa em zero

return modas

# Início do script
n = int(input('Quantas vezes você quer jogar o dado? '))
resultados = joga_dados(n)
hist = histograma(resultados)
print('Histograma: ', hist)
print('Modas: ', moda(hist))

```

Ex45: Implemente uma função que gere uma aposta válida da Mega Sena. A função deve gerar uma lista contendo 6 números aleatórios entre 1 e 60. Note que não deve haver repetição.

```

import numpy as np

def mega_sena():
    numeros = []
    while len(numeros) < 6:
        r = np.random.randint(1, 61)
        if r not in numeros:
            numeros.append(r)
    numeros.sort()
    return numeros

```

Ex46: A série  $1^1 + 2^2 + 3^3 + 4^4 + \dots + 10^{10} = 10405071317$ . Faça um programa que retorne os últimos 10 dígitos de  $1^1 + 2^2 + 3^3 + 4^4 + \dots + 1000^{1000}$

```

lista = list(range(1, 1001))      # cria lista de 1 a 1000
potencias = [x**x for x in lista] # cria lista com x elevado a x
soma = sum(potencias)             # soma todos elementos da lista
digitos = str(soma)               # converte resultado para string
print(digitos[len(digitos)-10:])  # imprime os 10 últimos caracteres

```

O site Project Euler (<https://projecteuler.net/archives>) contém uma série de problemas matemáticos desafiadores que podem ser resolvidos através da programação de computadores. O exercício anterior é um deles. O exercício a seguir é o problema 14 da lista do Project Euler.

Ex47: A maior sequência de Collatz

Esse é um dos problemas matemáticos mais misteriosos do mundo, principalmente porque não existem provas formais da sua solução. Suponha que seja escolhido um número inteiro qualquer  $n$ . A sequência iterativa a seguir é definida para qualquer  $n$  positivo:

$n \rightarrow n/2$ ,     se  $n$  é par  
 $n \rightarrow 3n + 1$ ,   se  $n$  é ímpar

Por exemplo, usando o número 13 como entrada, iremos produzir a seguinte sequência:

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

O fato intrigante é que para todo  $n$ , cedo ou tarde a sequência atinge o número 1 (essa é a conjectura de Collatz). A pergunta é: qual é o número  $n$  até 1 milhão, que produz a maior sequência, isto é, que demora mais para atingir o número 1. Sugestão: armazene a sequência de cada número numa lista e retorne como saída a maior delas.

```
def collatz(n):
    sequencia = [n]
    while n > 1:
        if n % 2 == 0:
            n = n//2
        else:
            n = 3*n+1
        sequencia.append(n)
    return sequencia

# Testa as sequencias com todos valores de 1 a 1000000
maior = 1
lista = []
for i in range(1, 1000000):
    L = collatz(i)
    if len(L) > maior:
        maior = len(L)
        lista = L

print(lista)
print()
print('Tamanho: %d' %maior)
```

Ex48: Leia uma lista de  $n$  posições com elementos maiores ou iguais a zero e o compacte, ou seja, leve os elementos nulos para o final do vetor. Dessa forma todos os “zeros” devem ficar para as posições finais do vetor.

Ex.   Entrada:       1 0 2 3 6 0 9 4 0 13 29  
         Saída:       1 2 3 6 9 4 13 29 0 0 0

Dica: Crie uma lista vazia e percorra a lista original adicionando um número  $x$  apenas se ele for maior que zero. Ao final complete com zeros (o número de zeros a ser adicionado deve ser a diferença entre o tamanho da lista maior e o tamanho da lista menor).

```
# função para compactar a lista L
def compacta(L):
    lista = []
    for x in L:
        if x > 0:
            lista.append(x)
    n = len(L)
    m = len(lista)
    zeros = (n - m)*[0]
    lista = lista + zeros
    return lista
```

```
# Início do script
L = []
while True:
    num = int(input('Digite um número (negativo para sair): '))
    if num < 0:
        break
    L.append(num)

print('Lista original: ', L)
LC = compacta(L)
print('Lista compactada: ', LC)
```

Ex49: Dada uma lista L de inteiros positivos, encontre o comprimento da maior sublista tal que a soma de todos os seus elementos seja igual a k.

Exemplo: Para L = [1, 3, 3, 5, 1, 4, 7, 2, 2, 2, 1, 8] e k = 6, existem várias subsequências que somam 6, por exemplo, as subsequências [3, 3], [5, 1] e [2, 2, 2]. O programa deve retornar o comprimento da maior delas, ou seja, 3.

```
def max_sequence(L, k):
    sequencia = []
    max_len = -1      # retorna -1 se não há subsequencia que soma k

    for i in L:
        sequencia.append(i)

        while sum(sequencia) > k:
            sequencia = sequencia[1:]

        if sum(sequencia) == k:
            max_len = max(max_len, len(sequencia))

    return max_len
```

## Vetores

Em Python, listas são objetos extremamente genéricos, podendo ser utilizados em uma gama de aplicações. Entretanto, o pacote Numpy (Numerical Python) oferece a definição de um tipo de dados específico para representar vetores: são os chamados **arrays**. Em resumo, um array é praticamente uma lista de números reais (floats), mas com algumas adicionais que facilitam cálculos matemáticos. Um exemplo é o seguinte: suponha que você tenha uma lista em que cada elemento representa o gasto em reais de uma compra efetuada pelo seu cartão.

```
L = [13.5, 8.0, 5.99, 27.30, 199.99, 57.21]
```

Você deseja dividir cada um dos valores pelo gasto total, para descobrir a porcentagem referente a cada compra. Como você tem uma lista, você deve fazer o seguinte:

```
lista = L
for i in range(len(L)):
    lista[i] = L[i]/sum(L)
```

Ou seja, não é permitido fazer

```
lista = L/sum(L) → ERRO
```

Mas se os dados tiverem sido armazenados num vetor, é possível fazer diretamente o comando acima:

```
import numpy as np

# criando um vetor (as duas maneiras são equivalentes)
v = np.array(L)
v = np.array([13.5, 8.0, 5.99, 27.30, 199.99, 57.21])

v = v/sum(v) → OK
```

Algumas funções importantes que todo objeto vetor possui, supondo que nosso vetor chama-se v:

```
v.max() - retorna o maior elemento do vetor
v.min() - retorna o menor elemento do vetor
v.argmax() - retorna o índice do maior elemento do vetor
v.argmin() - retorna o índice do menor elemento do vetor
v.sum() - retorna a soma dos elementos do vetor
v.mean() - retorna a média dos elementos do vetor
v.prod() - retorna o produto dos elementos do vetor
v.T - retorna o vetor transposto
v.clip(a, b) - o que é menor que a vira a e o que é maior que b vira b
v.shape() - retorna as dimensões do vetor/matriz
```

Para a referência completa das funções veja os links:

<https://docs.scipy.org/doc/numpy-dev/user/quickstart.html>

<https://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html#array-methods>

"A mind is like a parachute. It doesn't work if it is not open."  
-- Frank Zappa

## Variáveis compostas 2D: Matrizes

Matrizes em Python também são objetos criados com o comando `np.array`. Elas são vetores que possuem mais de uma dimensão, sendo tipicamente duas. Suponha que desejamos criar uma matriz 3 x 3 contendo os elementos de 1 até 9.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Isso é feito com o seguinte comando (no que as chaves):

```
M = np.array([[1,2,3], [4,5,6], [7,8,9]])
```

A seguir veremos um exemplo onde algumas operações básicas com matrizes são realizadas.

```
import numpy as np

M = np.array([[1,2],[3,4]])
N = np.array([[5,6],[7,8]])

# soma de 2 matrizes
C = M + N
# subtração de 2 matrizes
D = M - N
# multiplicação elemento por elemento
E = M*N
# multiplicação de matrizes (também serve para produto escalar entre 2
vetores e produto matriz vetor)
F = np.dot(M, N)

# gera uma matriz com números aleatórios entre 1 e 99
G = np.random.randint(1, 100, (3, 3))
```

Ex50: Faça uma função que receba uma matriz quadrada n x n como entrada e retorne a sua matriz triangular superior.

```
import numpy as np

def triangular_superior(M):
    if M.shape[0] != M.shape[1]:
        print('Matriz não é quadrada.')
        return -1
    else:
        for i in range(M.shape[0]):
            for j in range(M.shape[1]):
                # zerar tudo que está abaixo da diagonal
                if i > j:
                    M[i,j] = 0
        return M

M = np.random.randint(1, 100, (5,5))
```

```
print(M)
print()
print(triangular_superior(M))
```

Modifique a função anterior para que ela retorne a matriz triangular inferior.

Ex51: Dada uma matriz  $n \times n$  faça funções para:

- a) calcular a soma dos elementos da diagonal principal.
- b) calcular a soma dos elementos da diagonal secundária.
- c) calcular a soma dos elementos de cada uma das linhas.
- d) calcular a soma dos elementos de cada uma das colunas.

```
import numpy as np

def soma_diagonal(M):
    if M.shape[0] != M.shape[1]:
        print('Matriz não é quadrada, não tem diagonal.')
        return -1
    else:
        soma_diagonal = 0
        for i in range(M.shape[0]):
            for j in range(M.shape[1]):
                if i == j:
                    soma_diagonal += M[i,j]
        return soma_diagonal

def soma_diagonal_secundária(M):
    if M.shape[0] != M.shape[1]:
        print('Matriz não é quadrada, não tem diagonal secundária')
        return -1
    else:
        valor = M.shape[0] - 1
        soma_diagonal = 0
        for i in range(M.shape[0]):
            for j in range(M.shape[1]):
                if i + j == valor:
                    soma_diagonal += M[i,j]
        return soma_diagonal

def soma_linhas(M):
    # cria um vetor de dimensão igual ao número de colunas
    somas = np.zeros(M.shape[1])

    for j in range(M.shape[1]):
        for i in range(M.shape[0]):
            somas[j] = somas[j] + M[i,j]

    return somas

def soma_colunas(M):
    # cria um vetor de dimensão igual ao número de linhas
    somas = np.zeros(M.shape[0])
```

```

    for i in range(M.shape[0]):
        for j in range(M.shape[1]):
            somas[i] = somas[i] + M[i,j]

    return somas

```

```
M = np.random.randint(1, 100, (5,5))
```

```

print(M)
print()
print('Soma diagonal: %d ' %soma_diagonal(M))
print()
print('Soma diagonal secundária: %d' %soma_diagonal_secundária(M))
print()
print(soma_linhas(M))
print()
print(soma_colunas(M))

```

Ex52: Uma imagem em tons de cinza é representada computacionalmente por uma matriz de dimensões  $n \times n$  em que cada pixel é um número inteiro entre 0 (preto) e 255 (branco). O programa a seguir, lê uma imagem da internet, salva-a no computador e faz a sua leitura, criando uma matriz de inteiros. O histograma de uma imagem é um gráfico que mostra a distribuição dos níveis de cinza da imagem, indicando por exemplo se a imagem é muito escura ou clara. A moda é o valor mais frequente, ou seja, é aquele que mais ocorre, neste caso a cor mais usada na imagem. Podemos notar que a moda nada mais é que o índice do máximo valor do histograma.

```

import numpy as np
import matplotlib.pyplot as plt
import urllib.request

# Calcula a moda - a cor que mais ocorre na imagem
def moda(h):
    maior = h[0]
    pos = 0
    for i in range(len(h)):
        if h[i] > maior:
            maior = h[i]
            pos = i
    return pos

urllib.request.urlretrieve('https://www.ece.rice.edu/~wakin/images/lena512.bmp',
'lena.bmp')

matriz = plt.imread('lena.bmp')

# Mostra imagem
plt.figure(1)
plt.imshow(matriz, cmap='gray')
plt.show()

# Cria o histograma
h = np.zeros(256)
for i in range(matriz.shape[0]):
    for j in range(matriz.shape[1]):
        h[matriz[i,j]] += 1

```



```

# Plota o gráfico
eixo_x = list(range(256))
plt.figure(2)
plt.bar(eixo_x, h)
plt.show()

# Calcula a moda
m = moda(h)
print('Moda = %d' %m)

```

Ex53: Dada uma matriz quadrada  $n \times n$ , faça uma função que diga se ela é simétrica ou não.

```

import numpy as np

def simetrica(M):
    N = M.T
    simetria = True
    for i in range(M.shape[0]):
        for j in range(M.shape[1]):
            if M[i,j] == N[i,j] and simetria:
                continue
            else:
                simetria = False
    if simetria:
        return True
    else:
        return False

A = np.random.randint(1, 99, (5, 5))
B = np.array([[1, 0, 0],[0, 1, 0],[0, 0, 1]])

print(A)
print(B)

print(simetrica(A))
print(simetrica(B))

```

Ex54: Faça um programa para calcular  $y = Ax$ , onde  $A$  é uma matriz  $n \times n$  e  $x$  é um vetor  $n \times 1$ . Compare o resultado obtido pela sua implementação com o resultado da função `dot` do pacote `numpy`.

```

import numpy as np

def produto_matriz_vetor(A, x):
    # Número de linhas
    n = A.shape[0]
    # Número de colunas
    m = A.shape[1]
    # Vetor de saída
    y = np.zeros(m)
    # Realiza o produto
    for i in range(n):
        for j in range(m):
            y[i] = y[i] + A[i, j]*x[j]
    return y

```

```

# Início do script

# Cria matriz 5 x 5
A = np.random.randint(1, 6, (5, 5))
print('Matriz A: ')
print(A)

# Cria vetor 5 x 1
x = np.random.randint(1, 6, 5)
print('Vetor x: ', x)

#Calcula o resultado
y = produto_matriz_vetor(A, x)
print('Resultado da implementação')
print('Vetor y: ', y)

# Comparando com a função dot do pacote numpy
z = np.dot(A, x)
print('Resultado da função dot')
print('Vetor z:', z)

```

Ex55: Faça um programa para calcular  $C = AB$ , onde A, B e C são matrizes  $n \times n$ . Compare o resultado obtido pela sua implementação com o resultado da função dot do pacote numpy.

```

import numpy as np

# A é matriz n x m
# B é matriz m x p
def produto_matriz(A, B):
    # Número de linhas
    n = A.shape[0]
    # Número de colunas de A e linhas de B
    m = A.shape[1]
    # Número de colunas de B
    p = B.shape[1]
    # Vetor de saída
    C = np.zeros((n, p))
    # Realiza o produto
    for i in range(n):
        for j in range(p):
            for k in range(m):
                C[i, j] = C[i, j] + A[i, k]*B[k, j]
    return C

# Início do script
# Cria matriz A 5 x 5
A = np.random.randint(1, 6, (5, 5))
print('Matriz A: ')
print(A)

# Cria matriz B 5 x 5
B = np.random.randint(1, 6, (5, 5))
print('Matriz B: ')
print(B)

```

```
#Calcula o resultado
C = produto_matriz(A, B)
print('Resultado da implementação')
print('Matriz C: ')
print(C)

# Comparando com a função dot do pacote numpy
D = np.dot(A, B)
print('Resultado da função dot')
print('Matriz D:')
print(D)
```

"Lembre-se: a maioria das pessoas quer te ver bem, mas não melhor do que elas próprias."  
(Anônimo)

# Arquivos

Até o presente momento, todas as informações processadas pelos nossos scripts Python são voláteis, uma vez que permanecem na memória RAM até o encerramento do programa. Quando o programa encerra sua execução, todas as variáveis alocadas na memória são permanentemente deletadas. A pergunta que surge é: como fazemos para conseguir manter os dados, ou seja, armazená-los em um dispositivo de memória persistente. É aí que entra o conceito de arquivo (file). Ao escrever dados em arquivos, podemos acessar essas informações mesmo depois que a execução do programa tiver encerrado. Há basicamente dois tipos de arquivos para armazenamento de dados:

1. Arquivos binários: nesse caso, toda informação gravada no arquivo está codificada em binário, ou seja, em 0's e 1's. Dessa forma, para ler o conteúdo do arquivo é preciso decodificar os dados.
2. Arquivos texto: nesse caso, toda informação gravada no arquivo será codificada como caracteres, ou seja, letras e números. Assim, a visualização do seu conteúdo pode ser feita utilizando qualquer editor de texto comum.

## Abrindo um arquivo

Antes de ler ou escrever dados em um arquivo, é preciso primeiramente abri-lo. Em Python, o comando para isso é o `open()`

```
file = open(nome_do_arquivo, modo_de_acesso)
```

em que `nome_do_arquivo` é uma string que armazenará o nome do arquivo no sistema operacional e `modo_de_acesso` é um string que especifica qual o tipo do arquivo e se desejamos ler ou escrever nele. A tabela a seguir resume dos modos de acessos.

Modos	Descrição
'r'	Abre um arquivo para leitura
'w'	Abre um arquivo para escrita
'x'	Abre um arquivo para criação do zero
'a'	Abre um arquivo para adicionar conteúdo no final (append)
't'	Abre um arquivo em modo texto
'b'	Abre um arquivo no modo binário
'+'	Abre um arquivo para leitura e escrita

## Arquivos binários

O script a seguir ilustra como escrever variáveis em um arquivo binário.

```
# Abre arquivo para escrita em binário
f = open('binario.bin', 'wb')
# Cria um vetor de 5 elementos
L = [1, 2, 3, 4, 5]
M = [6, 7, 8]
# Converte o vetor para bytes (codificação para arquivo binário)
dados = bytearray(L)
# Escreve no arquivo
f.write(dados)
dados = bytearray(M)
f.write(dados)
# Fecha arquivo
f.close()
```

Agora, podemos ler o conteúdo do arquivo binario.bin para recuperar a variável L.

```
# Abre arquivo para leitura
f = open('binario.bin', 'rb')

# Lê primeira variável (primeiros 5 bytes)
# Cada inteiro é 1 byte
dados = f.read(5)
# Converte para lista
L = list(dados)
# Imprime conteúdo
print(L)

# Lê segunda variável (próximos 3 bytes)
dados = f.read(3)
# Converte para lista
M = list(dados)
# Imprime conteúdo
print(M)

# Fecha arquivo
f.close()
```

Para salvar vetores e matrizes, recomenda-se utilizar a função savez do pacote numpy.

```
import numpy as np

A = np.random.random((5, 5))
B = np.random.randint(1, 10, (7, 7))

np.savez('numpy.npz', A=A, B=B)
```

Para recuperar vetores e matrizes de um arquivo, há a função load do pacote numpy.

```
import numpy as np

dados = np.load('numpy.npz')

A = dados['A']
B = dados['B']

print(A)
print(B)
```

Note que ao utilizar as funções do pacote numpy não é necessário abrir e fechar o arquivo em questão de maneira explícita. Tudo isso já é gerenciado internamente pelos comandos savez e load.

## Arquivos texto

Diferentemente dos arquivos binários, os arquivos textos são compostos por caracteres e strings. Sendo assim, para nós humanos é bem mais natural do que o formato binário.

### Escrita em arquivo texto

Basicamente, para escrever em um arquivo texto, utilizamos o comando `write`, passando como parâmetro uma string.

```
# Abre o arquivo
f = open("texto.txt", 'w', encoding='utf-8')
# Escreve várias linhas no arquivo
f.write("Primeira linha de texto\n")
f.write("Segunda linha de texto\n\n")
f.write("Arquivos textos são legais!")
# Fechando o arquivo
f.close()
```

Também é possível escrever uma lista de strings no arquivo texto com o comando `writelines`.

```
# Escreve várias linhas de uma só vez
lista = ['Olá a todos, sejam bem-vindos!\n',\
        'Esse é um exemplo de como escrever em arquivos texto\n',\
        'utilizando o comando writelines()']
g = open('textfile.txt', 'w')
g.writelines(lista)
g.close()
```

Para salvar vetores e matrizes em arquivos texto, recomenda-se utilizar a função `savetxt` do pacote `numpy`.

```
import numpy as np

# Gera matriz com números aleatórios
A = np.random.random((5, 5))
# Salva no arquivo texto usando 3 casas decimais
np.savetxt('testando.txt', A, fmt='%.3f')
```

### Leitura de arquivo texto

Basicamente, a leitura de arquivos texto é realizada pela função `read()`. Se passarmos um argumento inteiro para a função, ela indica a quantidade de caracteres que serão lidos na sequência a partir do arquivo. Se não for passado parâmetro algum, a função lê todo conteúdo do arquivo texto em uma única string. Posteriormente, podemos utilizar o comando `split()` da string para separar em palavras. Alternativamente, o comando `readline()` realiza a leitura de exatamente uma linha do arquivo texto.

```
# Abre arquivo
f = open('texto.txt', 'r')
# Lê uma linha do arquivo
linha = f.readline()
# Imprime linha na tela
print(linha)
# Lê 5 caracteres do arquivo
caracteres = f.read(5)
# Imprime na tela
print(caracteres)
# Fecha arquivo
f.close()
```

É possível utilizar um loop para ler todas as linhas do arquivo texto sequencialmente.

```
# Abre arquivo
f = open('texto.txt', 'r')
# Loop para ler todas as linhas do arquivo
for linha in f:
    print(linha)
```

A seguir veremos alguns exercícios sobre manipulação de arquivos.

Ex56: Na pacata vila campestre de Numbertville, todos os telefones tem 6 dígitos. A companhia telefônica estabelece as seguintes regras sobre os números:

1. Não pode haver dois dígitos consecutivos idênticos, pois isso é chato;
2. A soma dos dígitos tem que ser par, porque isso é legal;
3. O último dígito não pode ser igual ao primeiro, porque isso dá azar;

Dadas essas regras perfeitamente razoáveis, bem projetadas e maduras, quantos números de telefone na lista abaixo são válidos?

```
213752 216732 221063 221545 225583 229133 230648 233222
236043 237330 239636 240138 242123 246224 249183 252936
254711 257200 257607 261424 263814 266794 268649 273050
275001 277606 278997 283331 287104 287953 289137 291591
292559 292946 295180 295566 297529 300400 304707 306931
310638 313595 318449 319021 322082 323796 326266 326880
327249 329914 334392 334575 336723 336734 338808 343269
346040 350113 353631 357154 361633 361891 364889 365746
365749 366426 369156 369444 369689 372896 374983 375223
379163 380712 385640 386777 388599 389450 390178 392943
394742 395921 398644 398832 401149 402219 405364 408088
412901 417683 422267 424767 426613 430474 433910 435054
440052 444630 447852 449116 453865 457631 461750 462985
463328 466458 469601 473108 476773 477956 481991 482422
486195 488359 489209 489388 491928 496569 496964 497901
500877 502386 502715 507617 512526 512827 513796 518232
521455 524277 528496 529345 531231 531766 535067 535183
```

536593 537360 539055 540582 543708 547492 550779 551595  
556493 558807 559102 562050 564962 569677 570945 575447  
579937 580112 580680 582458 583012 585395 586244 587393  
590483 593112 593894 594293 597525 598184 600455 600953  
601523 605761 608618 609198 610141 610536 612636 615233  
618314 622752 626345 626632 628889 629457 629643 633673  
637656 641136 644176 644973 647617 652218 657143 659902  
662224 666265 668010 672480 672695 676868 677125 678315

Obs: Copie e cole o conteúdo acima (números) num arquivo chamado numeros.txt

```
# Condição 1: não pode haver 2 dígitos iguais em sequencia
def condicao1(digitos):
    cond1 = True
    for i in range(len(digitos)-1):
        if (digitos[i] == digitos[i+1]):
            cond1 = False
            break
    return cond1

# Condição 2: soma dos dígitos deve ser par
def condicao2(digitos):
    lista = list(digitos) # converte para lista
    soma = 0

    for x in lista:
        soma = soma + int(x) # converte para inteiro!
        if soma % 2 == 0:
            cond2 = True # soma é par
        else:
            cond2 = False # soma é ímpar

    return cond2

# Condição 3: último dígito não pode ser igual ao primeiro
def condicao3(digitos):
    if digitos[0] != digitos[-1]:
        cond3 = True
    else:
        cond3 = False

    return cond3

# Início do script
# Abre arquivo para leitura
f = open('numeros.txt', 'r')

# Lê dados do arquivo numa única string
conteudo = f.read()

# Quebra string em lista (cada palavra vira um elemento da lista)
numeros = conteudo.split()
```



```
total = 0
# Percorre todos os elementos da lista
for num in numeros:

    if (condicao1(num) and condicao2(num) and condicao3(num)):
        total = total + 1

print('O total de números telefônicos válidos é %d' %total)
```

Ex57: Contar o número de ocorrências das palavras em um texto é uma tarefa muito importante na internet, uma vez que plataformas como o Google, o Twitter e o Facebook analisam milhares delas por segundo. Um exemplo são os *Trending Topics* do Twitter que registra os tópicos mais citados na rede. Esse problema visa ilustrar de forma básica como podemos construir um histograma de palavras que compõem um texto. Para isso, iremos utilizar uma cópia do famoso livro “Alice no país das maravilhas”, disponível gratuitamente em formato .txt da seguinte URL

<https://www.gutenberg.org/files/11/11-0.txt>

a) Quantas vezes o nome Alice aparece no texto?

b) Crie uma lista com todas as palavras iniciadas com a letra a e que tenha pelo menos 7 caracteres.

```
import urllib.request
import string

# Faz o download do livro em formato txt
urllib.request.urlretrieve('https://www.gutenberg.org/files/11/11-0.txt', 'alice.txt')

arquivo = open('alice.txt', 'r')    # abre arquivo para leitura

texto = arquivo.read()             # copia todo conteúdo do arquivo numa string
texto = texto.lower()              # converte string para letras minúsculas

# Substitui caracteres de pontuação por espaços em branco
for c in string.punctuation:
    texto = texto.replace(c, ' ')

# Converte string para uma lista de palavras (separa por espaços)
texto = texto.split()

alice = 0
L = []
for palavra in texto:
    if palavra == 'alice':
        alice = alice + 1
    if palavra[0] == 'a' and len(palavra) > 6 and palavra not in L:
        L.append(palavra)

# Fecha arquivo
arquivo.close()

print('O nome Alice aparece %d vezes no texto' %alice)
print('Palavras iniciadas com a letra A e com ao menos 7 letras:')
print(L)
print(len(L))
```

Ex58: Faça um programa que peça um número de 1 a 9 ao usuário e imprima a tabuada desse número em um arquivo texto.

```
# Abre arquivo
f = open('tabuada.txt', 'w')

n = int(input('Você quer gerar a tabuada de que número? '))

# Cabeçalho
s = 'Tabuada do ' + str(n) + '\n'
f.write(s)
f.write('-----\n')

for i in range(1, 11):
    x = i*n
    s = str(i) + ' x ' + str(n) + ' = ' + str(x) + '\n'
    f.write(s)

f.close()
```

Ex59: Dado o arquivo das notas de uma turma de Programação em Python conforme a seguir:

Alberto	8.5
Alan	6.5
Alex	7.0
Augusto	5.0
Beatriz	9.5
Carla	9.0
Débora	8.0
Diego	4.0
Eduardo	6.0
Gabriela	8.5
Gustavo	5.5
João	3.5
Juliana	7.0
Lucas	9.0
Marcela	7.0
Marcos	5.0
Osvaldo	4.5
Olívia	7.5
Paula	6.5
Roberto	10.0
Sílvio	5.5
Tatiana	8.0
Tiago	8.5
Valter	7.5
Vanderlei	3.0

OBS: Copie e cole o conteúdo acima para um arquivo chamado notas.txt

Leia o arquivo, calcule a média e o desvio padrão das notas. Em seguida produza 3 arquivos de saída: aprovados.txt, recuperacao.txt e reprovados.txt com os nomes e as notas dos alunos nessas condições. Imprima na tela a média da turma, o desvio padrão, a nota mais baixa e a nota mais alta.

**Critério de avaliação:**

nota  $\geq 6.0$  – Aprovado

$5.0 \leq \text{nota} < 6.0$  – Recuperação

nota  $< 5.0$  - Reprovado

```
import math

def media(L):
    soma = 0
    for x in L:
        soma = soma + x
    media = soma/len(L)
    return media

def desvio(L):
    soma = 0
    for x in L:
        soma = soma + (x - media(L))**2
    desvio = math.sqrt(soma/len(L))
    return desvio

def maximo(L):
    maior = L[0]
    for x in L:
        if x > maior:
            maior = x
    return maior

def minimo(L):
    menor = L[0]
    for x in L:
        if x < menor:
            menor = x
    return menor

# Abre arquivos
f = open('notas.txt', 'r')
aprovados = open('aprovados.txt', 'w')
recuperacao = open('recuperacao.txt', 'w')
reprovados = open('reprovados.txt', 'w')
# Lê as linhas
nomes = []
notas = []
for linha in f:
    L = linha.split()
    s = L[0]
    n = float(L[1])
    nomes.append(s)
    notas.append(n)
    if n >= 6.0:
        aprovados.write(linha)
    elif n >= 5.0 and n < 6.0:
        recuperacao.write(linha)
    elif n < 5.0:
        reprovados.write(linha)
```

```
# Imprime informações na tela
print('A média das notas é %.2f' %media(notas))
print('O desvio padrão das notas é %.2f' %desvio(notas))
print('A nota mais baixa da turma é %.2f' %minimo(notas))
print('A nota mais alta da turma é %.2f' %maximo(notas))

# Fecha os arquivos
f.close()
aprovados.close()
recuperacao.close()
reprovados.close()
```

"Oh, my friend, it's not what they take away from you that counts. It's what you do with what you have left."  
(Hubert Humphrey)

# Dicionários

Dicionários são estruturas de dados em Python que generalizam o conceito de lista, permitindo a criação de pares do tipo (chave, valor). Em uma lista, a chave é sempre um índice que indica a posição do elemento no conjunto (existe uma relação de ordem intrínseca pois o 0 vem antes do 1, que vem antes do 2, etc). Em um dicionário, a chave pode ser praticamente qualquer tipo de dados (string, float, tupla). É como se fosse uma etiqueta com um identificador único para cada valor armazenado na estrutura.

Para inserir elementos num dicionário, é preciso relacionar uma chave a um valor:

```
d = {}    # cria dicionário vazio
d['a'] = 'programação'
d['b'] = 'em'
d['c'] = 'python'
```

No dicionário d, todos os elementos são armazenados como pares (chave, valor)

```
{('a', 'programação'), ('b', 'em'), ('c', 'python')}
```

Assim, não devemos acessar os elementos como:

```
d[0] → ERRO X
d[1] → ERRO X
d[2] → ERRO X
```

Mas sim

```
d['a']
d['b']
d['c']
```

Para modificar um valor, basta fazer:

```
d['a'] = 'teste'
```

A posição dos pares (chave, valor) no conjunto não segue uma ordem, ou seja, não podemos garantir quem vem antes ou depois de quem. Um dos erros mais comuns que se tem é percorrer um dicionário como uma lista:

```
for i in range(len(d)):
    print(d[i])    → ERRO X
```

## Funções Auxiliares

A seguir veremos algumas funções muito úteis na manipulação de dicionários.

`d.clear()` - limpa dicionário (mesmo que fazer `d = {}`)

`d.keys()` - retorna um conjunto com todas as chaves

Ex: `'a' in d.keys()` - verifica se chave 'a' faz parte do dicionário d

`d.values()` - retorna um conjunto com todos os valores

Ex: `99 in d.values()` - verifica que 99 é um dos valores armazenados no dicionário d

d.items() - retorna um conjunto de todas as tuplas com pares (chave, valor)

Ex: ('a', 99) in d.items()

d.pop(chave) - retira par (chave, valor) com base na chave especificada

Ex: d.pop('a')

Ex60: Crie um dicionário em que as chaves são nomes e os valores são os números de telefones de 5 pessoas. Imprima na tela os nomes e telefones de cada um deles.

```
d = {}
d['José'] = '33079876'
d['Ana'] = '81889275'
d['João'] = '97490128'
d['Maria'] = '34141796'
d['André'] = '33216784'

print('Lista Telefônica')
for chave in d.keys():
    print('%s: %s' %(chave, d[chave]))
```

Outra forma de percorrer dicionário e imprimir os dados:

```
print('Lista Telefônica')
for chave, valor in d.items():
    print('%s: %s' %(chave, valor))
```

Outras consultas incluem:

- a) Retornar os nomes e telefones de todas as pessoas cujo nome inicia com a letra A
- b) Retornar os nomes e telefones de todas as pessoas cujo número inicia com 3 e termina com 6

```
def imprime_A(d):
    for chave in d.keys():
        if chave[0] == 'A':
            print('%s: %s' %(chave, d[chave]))

def imprime_B(d):
    for chave, valor in d.items():
        if valor[0] == '3' and valor[0] == '6':
            print('%s: %s' %(chave, valor))
```

A seguir veremos alguns problemas envolvendo dicionários na forma de exercícios.

Ex61: Seja uma lista L que contém o nome de diversas pessoas:

```
L = ['mark', 'henry', 'matthew', 'paul', 'luke', 'robert', 'joseph',
     'carl', 'michael']
```

Escreva uma função que receba L como entrada e retorne um dicionário em que os nomes são agrupados pelo tamanho, ou seja, na chave 4 deve ser armazenada uma lista com os nomes de 4 letras, e assim por diante.

```
def agrupa_nomes(L):
    d = {}
    for nome em L:
        chave = len(nome)
        if key not in d:      # se não existe grupo, cria vazio
            d[chave] = []
        d[chave].append(nome)
    return d
```

Ex62: Em uma papelaria os produtos a seguir são vendidos considerando a tabela de preços:

caderno	R\$ 10
pasta	R\$ 7.50
lápiz	RS 2.50
caneta	R\$ 3
borracha	R\$ 4

- Monte um dicionário d em que a chave é o nome do produto e o valor é seu preço
- Escreva uma função para determinar o produto mais barato
- Devido a inflação, os produtos que custam menos de R\$ 5 devem sofrer um aumento de preço de 20% enquanto os produtos que custam mais que R\$ 5 devem sofrer um aumento de 10%. Escreva uma função que atualize a tabela de preços.

```
# cria dicionário
d = {}
d['caderno'] = 10.0
d['pasta'] = 7.50
d['lapis'] = 2.50
d['caneta'] = 3.0
d['borracha'] = 4.0

# determina o produto mais barato
def mais_barato(dic):
    menor_preco = max(d.values())
    for chave, valor in d.items():
        if valor < menor_preco:
            menor_preco = valor
            produto = chave
    saida = [produto, menor_preco]
    return saida

# reajusta preços
def reajusta_precos(dic):
    print('NOVA TABELA DE PREÇOS')
    for chave in dic.keys():
        if dic[chave] < 5.0:
            dic[chave] = dic[chave]*1.2
        else:
            dic[chave] = dic[chave]*1.1
    print('%s : R$ %.2f' %(chave, dic[chave]))
```

Ex63: Num jogo do tipo RPG, personagens exploram um vasto mundo adquirindo novos itens para completar sua jornada. Suponha que você esteja desenvolvendo um jogo em que o personagem principal possui um inventário. Considere o seguinte código em Python:

```
inventario = {
    'ouro' : 500,
    'bolso' : ['rocha', 'barbante', 'pedra preciosa'],
    'mochila' : ['gaita', 'adaga', 'saco de dormir', 'corda']
}
```

- a) Suponha que seu personagem encontrou uma sacola. Adicione esse item como uma nova chave no seu inventário (sacola é uma lista inicialmente vazia).
- b) Adicione à sacola uma maçã, um pedaço de pão e carne.
- c) Escreva uma função `verifica_bolso(inventario)` para listar todos os itens do bolso do personagem em ordem alfabética.
- d) Adicione uma chave para o inventário chamada carteira, como uma lista vazia.
- e) Insira na carteira uma concha, uma carta e um bilhete.
- f) Escreva uma função `verifica_mochila(inventario)` para listar todos os itens da mochila do personagem em ordem alfabética.
- g) Escreva uma função `compra_comida(inventario, item, preco)` que adiciona o item a chave sacola do inventário e atualiza a quantidade atual de ouro. Verifique primeiro se há ouro suficiente.
- h) Escreva uma função `vende_item(inventario, item, preco)` que remove o item do inventário e adiciona o valor `preco` à quantidade de ouro.

```
inventario = {
    'ouro' : 500,
    'bolso' : ['rocha', 'barbante', 'pedra preciosa'],
    'mochila' : ['gaita', 'adaga', 'saco de dormir', 'corda']
}

def imprime_inventario(inventario):
    print('INVENTÁRIO ATUAL')
    for chave, valor in inventario.items():
        print('%s: %s' %(chave, valor))

# Imprime inventário inicial
imprime_inventario(inventario)

# a) e b) Adicionando uma chave 'sacola' a atribuindo uma lista a ela
print('Encontrou sacola com alimentos...')
inventario['sacola'] = []
inventario['sacola'].append('maçã')
inventario['sacola'].append('pedaço de pão')
inventario['sacola'].append('carne')

# c) Organizando a lista encontrada sob a chave 'bolso'
inventario['bolso'].sort()

# d) e) f) Adicionando chave carteira
print('Encontrou carteira...')
inventario['carteira'] = ['concha', 'carta', 'bilhete']
inventario['carteira'].sort()

# Organizando mochila
inventario['mochila'].sort()
```



```

# Imprime inventário atualizado
imprime_inventario(inventario)
# Função que compra comida e adiciona na sacola
def compra_comida(inventario, comida, preco):
    print('Comprando comida...')
    if inventario['ouro'] >= preco:
        inventario['sacola'].append(comida)
        inventario['ouro'] -= preco
    else:
        print('Você não tem ouro suficiente')

# Função que vende item do inventário
def vende_item(inventario, item, preco):
    print('Vendendo item...')
    vendeu = False
    for compartimento, conteudo in inventario.items():
        if type(conteudo) is list:
            if item in conteudo:
                conteudo.remove(item)
                inventario['ouro'] += preco
                vendeu = True
    if not vendeu:
        print('Não há item %s no inventário' %item)

compra_comida(inventario, 'frango', 10)
vende_item(inventario, 'pedra preciosa', 200)

# Imprime inventário final
imprime_inventario(inventario)

```

Ex64: Contar o número de ocorrências das palavras em um texto é uma tarefa muito importante na internet, uma vez que plataformas como o Google, o Twitter e o Facebook analisam milhares delas por segundo. Um exemplo são os *Trending Topics* do Twitter que registra os tópicos mais citados na rede. Esse problema visa ilustrar de forma básica como podemos construir um histograma de palavras que compõem um texto. Para isso, iremos utilizar uma cópia do famoso livro “Alice no país das maravilhas”, disponível gratuitamente em formato .txt da seguinte URL

<https://www.gutenberg.org/files/11/11-0.txt>

Pergunta-se:

- Quantas vezes o nome Alice aparece no texto?
- Qual a palavra que mais vezes ocorre no texto?
- Crie uma lista com todas as palavras iniciadas com a letra a e que tenha pelo menos 7 caracteres.

```

import urllib.request
import string

# Faz o download do livro em formato txt
urllib.request.urlretrieve('https://www.gutenberg.org/files/11/11-0.txt', 'alice.txt')

arquivo = open('alice.txt', 'r')    # abre arquivo para leitura

texto = arquivo.read()             # copia todo conteúdo do arquivo numa string

```

```

texto = texto.lower()      # converte string para letras minúsculas

# Substitui caracteres de pontuação por espaços em branco
for c in string.punctuation:
    texto = texto.replace(c, ' ')

# Converte string para uma lista de palavras (separa por espaços)
texto = texto.split()

# Cria dicionário e adiciona cada palavra encontrada no texto

dic = {}
for p in texto:
    if p not in dic:
        dic[p] = 1
    else:
        dic[p] += 1

# Fecha arquivo
arquivo.close()

print('O nome Alice aparece %d vezes no texto' %dic['alice'])

# Imprime palavra que mais ocorre no texto
valor_max = 0
for (chave, valor) in dic.items():
    if valor > valor_max:
        chave_max = chave
        valor_max = valor

print('A palavra que mais aparece no texto é %s com %d ocorrências.' %
(chave_max, valor_max))

# Cria uma lista com todas as palavras iniciadas pela letra B com ao
menso 7 letras
lista_a = []
for (chave, valor) in dic.items():
    if chave[0] == 'a' and len(chave) >= 7:
        lista_a.append(chave)

print('Palavras iniciadas com A e de ao menos 7 letras')
print(lista_a)

```

Pergunta: como faríamos para saber quais as 30 palavras que mais aparecem?  
Criando uma lista ordenada pelo número de ocorrências

```

# Ranking: lista de palavras em ordem decrescente de ocorrências
lista_palavras = sorted(dic, key=dic.get, reverse=True)
for palavras in lista_palavras:
    print('%s : %d' %(palavras, dic[palavras]))

```

Ex65: Crie uma função chamada frequencia que recebe como entrada uma string de caracteres que representam letras do alfabeto (i.e., aaabbbbaabababccaabaacccc) e retorne como saída um dicionário que represente a frequência de cada caractere na cadeia.

```
def frequencia(cadeia):
    d = {}
    for c in cadeia:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d

cadeia = input('Entre com a cadeia de caracteres')
f = frequencia(cadeia)

for chave, valor in f.items():
    print('%s : %d' %(chave, valor))
```

Ex66: Um dicionário recebe esse nome pois é uma estrutura muito utilizada para mapear ou traduzir uma string de entrada numa string de saída. Faça um pequeno tradutor de inglês – português utilizando algumas poucas palavras.

```
d = {'I':'eu', 'the':'o', 'sky':'céu', 'you':'você', 'is': 'é',
     'like':'gosto', 'love':'amo', 'dogs':'cachorros', 'cats':'gatos',
     'in':'em', 'cars':'carros', 'blue':'azul'}

def traducao(frase, dic):
    palavras = frase.split()
    saida = ''
    for p in palavras:
        saida = saida + ' ' + dic[p]
    return saida

print('LISTA DE PALAVRAS')
print(d)
print()
frase = input('Entre com uma frase: ')
print(traducao(frase, d))
```

Ex67: Suponha uma turma com N alunos em que a avaliação final será realizada com base em 3 avaliações (provas P1, P2 e P3). Alunos com média inferior a 6 são reprovados e aqueles com média igual ou superior a 6 são aprovados. Sabendo disso responda:

a) Escreva um trecho de código em Python referente aos dados a seguir. Considere um dicionário inicialmente vazio denominado dados, em que a chave é o nome do aluno e o valor é uma lista contendo as 3 notas que o aluno obteve.

Alunos	P1	P2	P3
João	6.5	5.0	7.0
Ana	8.5	4.0	7.5
Carlos	3.0	5.0	7.0
Maria	5.5	6.0	9.0
José	3.5	5.0	6.0

b) Deseja-se saber quais os aluno aprovados e reprovados. Escreva uma função que percorra o dicionário imprimindo na tela os nomes dos alunos e sua condição: aprovado ou reprovado.

c) Os 3 melhores alunos da turma recebem uma bolsa de estudos como incentivo ao bom desempenho. Faça uma função que escreva na tela o nome dos alunos e as médias em ordem decrescente de nota, ou seja, um ranking dos estudantes.

```
# cria dicionário
d = {}
d['João'] = [6.5, 5.0, 7.0]
d['Ana'] = [8.5, 4.0, 7.5]
d['Carlos'] = [3.0, 5.0, 7.0]
d['Maria'] = [5.5, 6.0, 9.0]
d['José'] = [3.5, 5.0, 6.0]

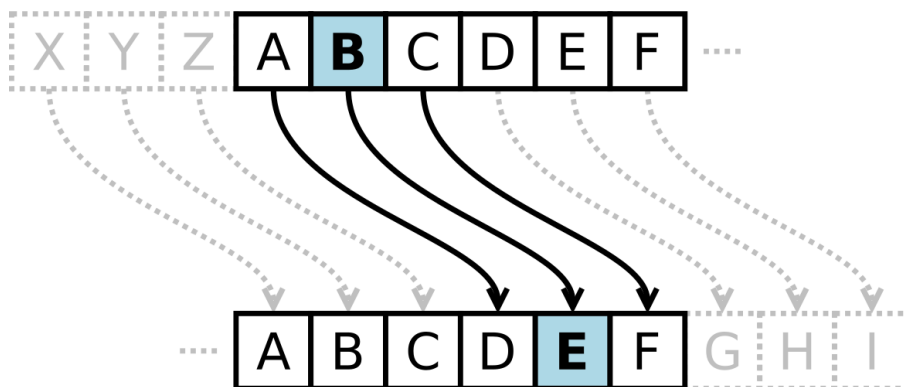
# imprime na tela os aprovados e reprovados
def verifica_aprovados(dic):
    for chave, valor in dic.items():
        media = sum(valor)/len(valor)
        if media >= 6:
            print('%s aprovado(a): média = %.2f' %(chave, media))
        else:
            print('%s reprovado(a): média = %.2f' %(chave, media))

# imprime na tela um ranking dos alunos por desempenho
def gera_ranking(dic):
    medias_finais = {}
    for chave, valor in dic.items():
        media = sum(valor)/len(valor)
        medias_finais[chave] = media    # novo dicionário
```

```
# retorna lista dos alunos em ordem decrescente de médias
nomes = sorted(medias_finais, key=medias_finais.get, reverse=True)

print('RANKING')
for x in nomes:
    print('%s : %.2f' %(x, medias_finais[x]))
```

Ex68: Criptografia é o processo de codificar mensagens utilizando um conjunto de símbolos de modo a dificultar a interpretação do seu conteúdo. Uma forma bastante simples de criptografar uma mensagem de texto consiste em realizar um mapeamento denominado de cifra de César.



A ação de uma cifra de César é mover cada letra do alfabeto um número de vezes fixo abaixo no alfabeto. Este exemplo está com uma troca de três, então o B no texto normal se torna E no texto cifrado.

a) Utilizando um dicionário para implementar a cifra de César, faça uma função encrypt que recebe como entrada 3 parâmetros: uma mensagem, um alfabeto e um valor K (deslocamento) e produz como saída a mensagem criptografada.

b) Utilizando um dicionário para implementar a cifra de César, faça uma função decrypt que recebe como entrada 3 parâmetros: uma mensagem, um alfabeto e um valor K (deslocamento) e produz como saída a mensagem descriptografada.

```
from unicodedata import normalize

# Criptografa mensagem usando alfabeto e chave
def encrypt(mensagem, alfabeto, chave):
    # aplica chave: alfabeto deslocado de k posições
    cifra = alfabeto[k:] + alfabeto[0:k]

    # cria 2 listas com os caracteres
    lista_alfabeto = list(alfabeto)
    lista_cifra = list(cifra)

    # cria mapeamento para os caracteres da mensagem
    # cria tuplas a partir de 2 listas e monta dicionário
    d = dict(zip(lista_alfabeto, lista_cifra)) # mapeia alfabeto

    # Criptografa mensagem
    saida = ''
    for c in mensagem.lower():
        saida = saida + d[c]

    return saida

# Descriptografa mensagem usando alfabeto e chave
def decrypt(mensagem, alfabeto, chave):
    # aplica chave: alfabeto deslocado de k posições
    cifra = alfabeto[k:] + alfabeto[0:k]

    # cria 2 listas com os caracteres
    lista_alfabeto = list(alfabeto)
    lista_cifra = list(cifra)

    # cria mapeamento para os caracteres da mensagem
    # cria tuplas a partir de 2 listas e monta dicionário
    d = dict(zip(lista_cifra, lista_alfabeto)) # mapeia cifra

    original = ''
    for c in mensagem.lower():
        original = original + d[c]

    return original

# Início do script
texto = input('Entre com uma sentença: ')
k = int(input('Entre com a chave (inteiro k): '))

alfabeto = 'abcdefghijklmnopqrstuvwxyz '
```

```
# Remove acentos, til, e caracteres especiais
texto = normalize('NFKD', texto).encode('ascii', 'ignore').decode('ascii')

print('Mensagem original: %s' %texto)

criptografada = encript(texto, alfabeto, k)
print('Mensagem criptografada: %s' %criptografada)

recuperada = decript(criptografada, alfabeto, k)
print('Mensagem descriptografada: %s' %recuperada)
```

"Solutions are not found by pointing fingers; they are reached by extending hands."  
(Aysha Taryam)

# Ordenação de dados

Ser capaz de ordenar os elementos de um conjunto de dados é uma das tarefas básicas mais requisitadas por aplicações computacionais. Como exemplo, podemos citar a busca binária, um algoritmo de busca muito mais eficiente que a simples busca sequencial. Buscar elementos em conjuntos ordenados é bem mais rápido do que em conjuntos desordenados. Existem diversos algoritmos de ordenação, sendo alguns mais eficientes do que outros. Neste curso, iremos apresentar 3 deles: Bubble sort, Selection sort e Insertion sort.

## Bubble sort

O algoritmo *Bubble sort* é uma das abordagens mais simplistas para a ordenação de dados. A ideia básica consiste em percorrer o vetor diversas vezes, em cada passagem fazendo flutuar para o topo da lista (posição mais a direita possível) o maior elemento da sequência. Esse padrão de movimentação lembra a forma como as bolhas em um tanque procuram seu próprio nível, e disso vem o nome do algoritmo (também conhecido como o método bolha)

Embora no melhor caso esse algoritmo necessite de apenas  $n$  operações relevantes, onde  $n$  representa o número de elementos no vetor, no pior caso são feitas  $n^2$  operações. Portanto, diz-se que a complexidade do método é de ordem quadrática. Por essa razão, ele não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados. A seguir veremos uma implementação em Python desse algoritmo.

```
import numpy as np
import time

# Algoritmo Bubblesort
def BubbleSort(vetor):
    # Percorre cada elemento da lista
    for i in range(len(vetor)-1, 0, -1):
        # Flutua o maior elemento para a posição mais a direita
        for j in range(i):
            if vetor[j] > vetor[j+1]:
                aux = vetor[j]
                vetor[j] = vetor[j+1]
                vetor[j+1] = aux

# Início do script
n = int(input('Qual é o tamanho do vetor a ser ordenado? '))
# Início do script
X = np.random.random(n)
# Imprime vetor
print('Vetor não ordenado: ')
print(X)
# Aplica Bubblesort
inicio = time.time()
BubbleSort(X)
fim = time.time()
# Imprime vetor ordenado
print('Vetor ordenado: ')
print(X)
print()
print('Tempo de processamento: %.3f s' %(fim - inicio))
```

Exemplo: mostre os passos necessários para a ordenação do seguinte vetor

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

1ª passagem (levar maior elemento para última posição)

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]  
[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]  
[5, 2, 7, 13, -3, 4, 15, 10, 1, 6]  
[5, 2, 7, -3, 13, 4, 15, 10, 1, 6]  
[5, 2, 7, -3, 4, 13, 15, 10, 1, 6]  
[5, 2, 7, -3, 4, 13, 15, 10, 1, 6]  
[5, 2, 7, -3, 4, 13, 15, 10, 1, 6]  
[5, 2, 7, -3, 4, 13, 10, 15, 1, 6]  
[5, 2, 7, -3, 4, 13, 10, 1, 15, 6]

[5, 2, 7, -3, 4, 13, 10, 1, 6, 15]

2ª passagem (levar segundo maior para penúltima posição)

[2, 5, 7, -3, 4, 13, 10, 1, 6, 15]  
[2, 5, 7, -3, 4, 13, 10, 1, 6, 15]  
[2, 5, -3, 7, 4, 13, 10, 1, 6, 15]  
[2, 5, -3, 4, 7, 13, 10, 1, 6, 15]  
[2, 5, -3, 4, 7, 13, 10, 1, 6, 15]  
[2, 5, -3, 4, 7, 10, 13, 1, 6, 15]  
[2, 5, -3, 4, 7, 10, 1, 13, 6, 15]

[2, 5, -3, 4, 7, 10, 1, 6, 13, 15]

3ª passagem (levar terceiro maior para antepenúltima posição)

[2, 5, -3, 4, 7, 10, 1, 6, 13, 15]  
[2, -3, 5, 4, 7, 10, 1, 6, 13, 15]  
[2, -3, 4, 5, 7, 10, 1, 6, 13, 15]  
[2, -3, 4, 5, 7, 10, 1, 6, 13, 15]  
[2, -3, 4, 5, 7, 10, 1, 6, 13, 15]  
[2, -3, 4, 5, 7, 1, 10, 6, 13, 15]

[2, -3, 4, 5, 7, 1, 6, 10, 13, 15]

4ª passagem

[-3, 2, 4, 5, 7, 1, 6, 10, 13, 15]  
[-3, 2, 4, 5, 7, 1, 6, 10, 13, 15]  
[-3, 2, 4, 5, 7, 1, 6, 10, 13, 15]  
[-3, 2, 4, 5, 7, 1, 6, 10, 13, 15]  
[-3, 2, 4, 5, 1, 7, 6, 10, 13, 15]

[-3, 2, 4, 5, 1, 6, 7, 10, 13, 15]

5ª passagem



[-3, 2, 4, 5, 1, 6, 7, 10, 13, 15]  
[-3, 2, 4, 5, 1, 6, 7, 10, 13, 15]  
[-3, 2, 4, 5, 1, 6, 7, 10, 13, 15]  
[-3, 2, 4, 1, 5, 6, 7, 10, 13, 15]

[-3, 2, 4, 1, 5, 6, 7, 10, 13, 15]

6ª passagem

[-3, 2, 4, 1, 5, 6, 7, 10, 13, 15]  
[-3, 2, 4, 1, 5, 6, 7, 10, 13, 15]  
[-3, 2, 1, 4, 5, 6, 7, 10, 13, 15]

[-3, 2, 1, 4, 5, 6, 7, 10, 13, 15]

7ª passagem

[-3, 2, 1, 4, 5, 6, 7, 10, 13, 15]  
[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

8ª passagem

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

9ª passagem

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

Fim: garantia de que vetor está ordenado só é obtida após todos os passos.

## Selection sort

A ordenação por seleção é um método baseado em se passar o menor valor do vetor para a primeira posição mais a esquerda disponível, depois o de segundo menor valor para a segunda posição e assim sucessivamente, com os  $n - 1$  elementos restantes. Esse algoritmo compara a cada iteração um elemento com os demais, visando encontrar o menor. A complexidade desse algoritmo será sempre de ordem quadrática, isto é o número de operações realizadas depende do quadrado do tamanho do vetor de entrada. Algumas vantagens desse método são: é um algoritmo simples de ser implementado, não usa um vetor auxiliar e portanto ocupa pouca memória, é um dos mais rápidos para vetores pequenos. Como desvantagens podemos citar o fato de que ele não é muito eficiente para grandes vetores.

```

import numpy as np
import time

# Algoritmo Selectionsort
def SelectionSort(vetor):
    # Percorre todos os elementos do vetor
    for i in range(len(vetor)):
        menor = i
        # Encontra o menor elemento
        for k in range(i+1, len(vetor)):
            if vetor[k] < vetor[menor]:
                menor = k
        # Troca a posição do elemento i com o menor
        tmp = vetor[menor]
        vetor[menor] = vetor[i]
        vetor[i] = tmp

# Início do script
n = int(input('Qual é o tamanho do vetor a ser ordenado? '))
# Início do script
X = np.random.random(n)
# Imprime vetor
print('Vetor não ordenado: ')
print(X)
# Aplica Selectionsort
inicio = time.time()
SelectionSort(X)
fim = time.time()
# Imprime vetor ordenado
print('Vetor ordenado: ')
print(X)
print()
print('Tempo de processamento: %.3f s' %(fim - inicio))

```

Exemplo: mostre os passos necessários para a ordenação do seguinte vetor

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

1ª passagem: [5, 2, 13, 7, -3, 4, 15, 10, 1, 6] → [-3, 2, 13, 7, 5, 4, 15, 10, 1, 6]  
 2ª passagem: [-3, 2, 13, 7, 5, 4, 15, 10, 1, 6] → [-3, 1, 13, 7, 5, 4, 15, 10, 2, 6]  
 3ª passagem: [-3, 1, 13, 7, 5, 4, 15, 10, 2, 6] → [-3, 1, 2, 7, 5, 4, 15, 10, 13, 6]  
 4ª passagem: [-3, 1, 2, 7, 5, 4, 15, 10, 13, 6] → [-3, 1, 2, 4, 5, 7, 15, 10, 13, 6]  
 5ª passagem: [-3, 1, 2, 4, 5, 7, 15, 10, 13, 6] → [-3, 1, 2, 4, 5, 7, 15, 10, 13, 6]  
 6ª passagem: [-3, 1, 2, 4, 5, 7, 15, 10, 13, 6] → [-3, 1, 2, 4, 5, 6, 15, 10, 13, 7]  
 7ª passagem: [-3, 1, 2, 4, 5, 6, 15, 10, 13, 7] → [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]  
 8ª passagem: [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15] → [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]  
 9ª passagem: [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15] → [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

Fim: garantia de que vetor está ordenado só é obtida após todos os passos.

## Insertion sort

*Insertion sort*, ou ordenação por inserção, é o algoritmo de ordenação que, dado um vetor inicial constrói um vetor final com um elemento de cada vez, uma inserção por vez. Assim como algoritmos de ordenação quadráticos, é bastante eficiente para problemas com pequenas entradas, sendo o mais eficiente entre os algoritmos desta ordem de classificação.

Podemos fazer uma comparação do Insertion sort com o modo de como algumas pessoas organizam um baralho num jogo de cartas. Imagine que você está jogando cartas. Você está com as cartas na mão e elas estão ordenadas. Você recebe uma nova carta e deve colocá-la na posição correta da sua mão de cartas, de forma que as cartas obedeçam a ordenação.

A cada nova carta adicionada a sua mão de cartas, a nova carta pode ser menor que algumas das cartas que você já tem na mão ou maior, e assim, você começa a comparar a nova carta com todas as cartas na sua mão até encontrar sua posição correta. Você insere a nova carta na posição correta, e, novamente, sua mão é composta de cartas totalmente ordenadas. Então, você recebe outra carta e repete o mesmo procedimento. Então outra carta, e outra, e assim por diante, até você não receber mais cartas. Esta é a ideia por trás da ordenação por inserção. Percorra as posições do vetor, começando com o índice zero. Cada nova posição é como a nova carta que você recebeu, e você precisa inseri-la no lugar correto no sub-vetor ordenado à esquerda daquela posição.

```
import numpy as np
import time

# Algoritmo Insertionsort
def InsertionSort(vetor):
    # Percorre cada elemento do vetor
    for i in range(1, len(vetor)):
        k = i
        # Insere o pivô na posição correta
        while k > 0 and vetor[k] < vetor[k-1]:
            tmp = vetor[k]
            vetor[k] = vetor[k-1]
            vetor[k-1] = tmp
            k = k - 1

# Início do script
n = int(input('Qual é o tamanho do vetor a ser ordenado? '))
# Início do script
X = np.random.random(n)
# Imprime vetor
print('Vetor não ordenado: ')
print(X)
# Aplica Insertionsort
inicio = time.time()
InsertionSort(X)
fim = time.time()
# Imprime vetor ordenado
print('Vetor ordenado: ')
print(X)
print()
print('Tempo de processamento: %.3f s' %(fim - inicio))
```

Exemplo: mostre os passos necessários para a ordenação do seguinte vetor

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

1ª passagem: [5, 2, 13, 7, -3, 4, 15, 10, 1, 6] → [2, 5, 13, 7, -3, 4, 15, 10, 1, 6]  
2ª passagem: [2, 5, 13, 7, -3, 4, 15, 10, 1, 6] → [2, 5, 13, 7, -3, 4, 15, 10, 1, 6]  
3ª passagem: [2, 5, 13, 7, -3, 4, 15, 10, 1, 6] → [2, 5, 7, 13, -3, 4, 15, 10, 1, 6]  
4ª passagem: [2, 5, 7, 13, -3, 4, 15, 10, 1, 6] → [-3, 2, 5, 7, 13, 4, 15, 10, 1, 6]  
5ª passagem: [-3, 2, 5, 7, 13, 4, 15, 10, 1, 6] → [-3, 2, 4, 5, 7, 13, 15, 10, 1, 6]  
6ª passagem: [-3, 2, 4, 5, 7, 13, 15, 10, 1, 6] → [-3, 2, 4, 5, 7, 13, 15, 10, 1, 6]  
7ª passagem: [-3, 2, 4, 5, 7, 13, 15, 10, 1, 6] → [-3, 2, 4, 5, 7, 10, 13, 15, 1, 6]  
8ª passagem: [-3, 2, 4, 5, 7, 10, 13, 15, 1, 6] → [-3, 1, 2, 4, 5, 7, 10, 13, 15, 6]  
9ª passagem: [-3, 1, 2, 4, 5, 7, 10, 13, 15, 6] → [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

Fim: garantia de que vetor está ordenado só é obtida após todos os passos.

"True teachers are those who use themselves as bridges over which they invite their students to cross; then, having facilitated their crossing, joyfully collapse, encouraging them to create their own."  
(Nikos Kazantzakis)

# Recursão

Dizemos que uma função é recursiva se ela é definida em termos dela mesma. Em matemática e computação uma classe de objetos ou métodos exibe um comportamento recursivo quando pode ser definido por duas propriedades:

1. Um caso base: condição de término da recursão em que o processo produz uma resposta.
2. Um passo recursivo: um conjunto de regras que reduz todos os outros casos ao caso base.

A série de Fibonacci é um exemplo clássico de recursão, pois:

$F(1) = 1$  (caso base 1)

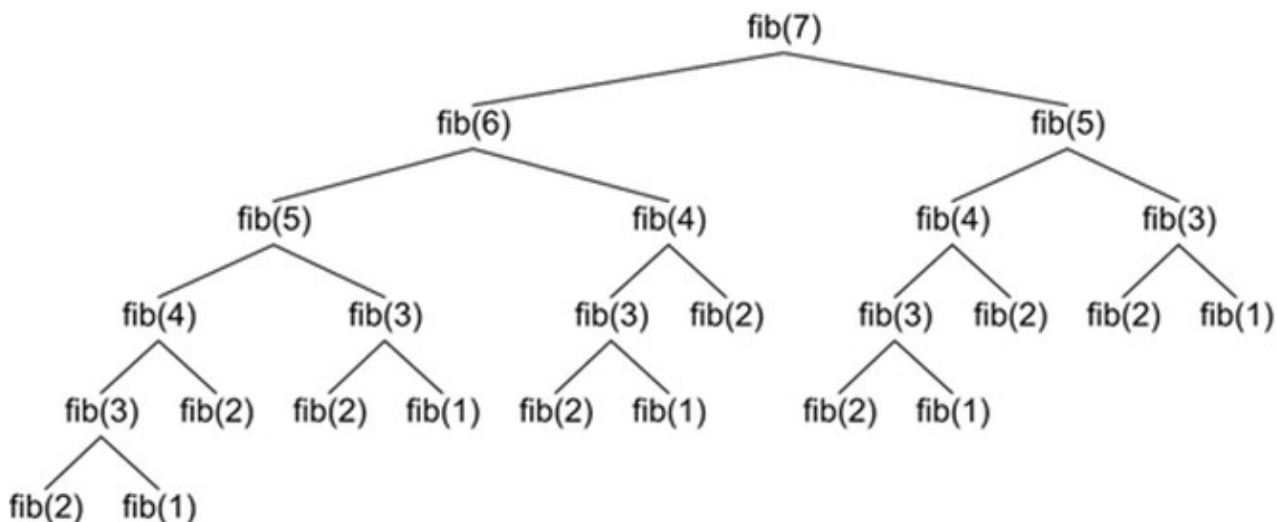
$F(2) = 1$  (caso base 2)

Para todo  $n > 1$ ,  $F(n) = F(n - 1) + F(n - 2)$

Ex69: Faça uma função recursiva em Python para calcular o n-ésimo termo da sequência de Fibonacci.

```
def fib(n):  
    if n == 0 or n == 1:  
        return 1  
    else:  
        return fib(n-1) + fib(n-2)  
  
n = int(input('Enter com o valor de n: '))  
resultado = fib(n)  
  
print('Fibonacci(%d) = %d' %(n, resultado))
```

Note entretanto que a função fibonacci recursiva não é muito eficiente de ponto de vista computacional. Isso ocorre pois durante o cálculo de  $F(n)$ , os valores de  $F(n-2)$ ,  $F(n-3)$ , etc... são calculados várias vezes. O número de chamadas recursivas cresce exponencialmente. O padrão recursivo consiste na expansão de uma árvore binária.



Veja que para computar  $F(3)$  são necessárias duas recursões, para  $F(4)$  são 4 recursões e assim sucessivamente. Veja que o crescimento é praticamente exponencial.

	Número de recursões
$F(3)$	2
$F(4)$	$4 = 2 + 1 + (0 + 1)$
$F(5)$	$8 = (4 + 1) + (2 + 1)$
$F(6)$	$14 = (8 + 1) + (4 + 1)$
$F(7)$	$24 = (14 + 1) + (8 + 1)$
$F(8)$	$40 = (24 + 1) + (14 + 1)$

Quantas recursões são necessárias para calcular  $F(9)$ ?

Há um padrão na sequência que deve estar óbvio agora: 2, 4, 8, 14, 24, 40, ...

O próximo elemento é a soma dos 2 anteriores mais 2. Assim, o número de recursões para calcular  $F(9)$  será  $40 + 24 + 2 = 66$ . Note que é uma outra sequência, mas parecida com a de Fibonacci. E se desejarmos saber como calcular o número de recursões necessárias para computar  $F(100)$ ? O programa a seguir faz esse cálculo de maneira rápida.

```
# Calcula quantas recursões são necessárias para calcular F(n)
# 0, 0, 2, 4, 8, 14, 24, 40,...
n = int(input('Entre com o valor de n (n > 2): '))

if n == 3:
    ops = 2
    print('O número de recursões para calcular F(3) é %d' %ops)
else:
    a, b = 0, 2
    for i in range(3, n):
        a, b = b, a + b + 2
    ops = b
    print('O número de recursões para calcular F(%d) é %d' %(n, ops))
```

Após executar o programa com  $n = 100$ , verificamos que o número de recursões necessárias para calcular  $F(100)$  é 708449696358523830148. Esse número é da ordem de  $10^{20}$ , ou seja, é de uma magnitude imensa. Tente executar com  $n = 100$  e verá que não há condições. Portanto, para o problema específico da sequência de Fibonacci, a versão iterativa é mais eficiente. Porém, existem muitos problemas em computação cuja solução recursiva é mais eficiente que a solução iterativa. Em computação, algoritmos recursivos existem como uma alternativa a algoritmos iterativos, ou seja, métodos que utilizam explicitamente estruturas de repetição. Com a recursão, podemos eliminar iterações. A seguir veremos uma implementação em Python de uma função recursiva para o cálculo do fatorial de um inteiro arbitrário  $n$ .

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

n = int(input('Entre com o valor de n: '))
print('%d! = %d' %(n, fatorial(n)))
```

Ex70: Faça um script recursivo para calcular o somatório a seguir:

$$S = \sum_{i=1}^n i$$

```
def somatorio(n):
    if n == 1:
        return 1
    else:
        return n + somatorio(n-1)

n = int(input('Entre com o valor de n: '))
print('S = %d' %(somatorio(n)))
```

Ex71: Faça um script recursivo para calcular

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

Utilize a função fatorial recursiva.

```
import math

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

def exp(x, n):
    if n == 0:
        return 1
    else:
        return (x**n)/fatorial(n) + exp(x, n-1)

x = float(input('Entre com o valor de x: '))
n = int(input('Número de termos n: '))

print('exp(%.2f) = %.9f' %(x, exp(x, n)))
print('Valor exato: %.9f' %(math.exp(x)))
```

Ex72: O máximo divisor comum entre dois inteiros a e b é o maior inteiro n que divide tanto a quanto b. Seja a = 1071 e b = 462. Primeiramente, devemos calcular quantas vezes b cabe em a e tomar o excesso, ou seja, sabemos que  $1071 // 462 = 2$  e  $1071 \% 462 = 147$ . O processo é então repetido, fazendo com que calculemos quantas vezes 147 cabe em 462 e tomemos o excesso. Sabemos que  $462 // 147 = 3$  e  $462 \% 147 = 21$ . Fazendo a iteração novamente, temos que  $147 // 21 = 7$  e  $147 \% 21 = 0$ . Como o resto é zero, o algoritmo para e temos que  $\text{mdc}(a, b) = 21$ .

Faça duas funções para calcular o MDC entre dois inteiros: uma iterativa e outra recursiva.

```

# Função iterativa
def mdc(a, b):
    while b > 0:
        a, b = b, a % b
    return a

# Função recursiva
def mdc_r(a, b):
    if b == 0:
        return a
    else:
        return mdc_r(b, a % b)

a = int(input('Entre com o valor de a: '))
b = int(input('Entre com o valor de b: '))
print()

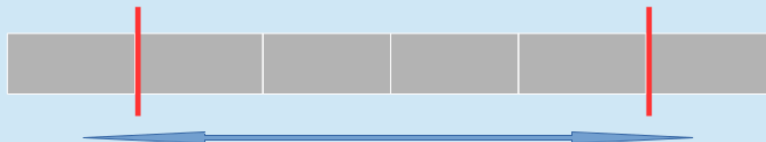
print('Iterativo')
print('mdc(%d, %d) = %d' %(a, b, mdc(a, b)))
print()

print('Recursivo')
print('mdc(%d, %d) = %d' %(a, b, mdc_r(a, b)))

```

Ex73: Faça uma função recursiva para inverter uma lista. Sugestão: utilize a seguinte ideia – inicie com início = 0 e fim = len(L) – 1, depois inverta o primeiro e o último elementos e chame a função recursiva com início incrementado de 1 e fim decrementado de 1. A condição de parada deve ser o fim ser menor ou igual que o início.

- Passo da recursão:



- Troca 1o. e último elementos e inverte resto do array.

```

# Função recursiva
def inverte(L, inicio, fim):
    if fim <= inicio:
        return L
    else:
        L[inicio], L[fim] = L[fim], L[inicio]
        inverte(L, inicio+1, fim-1)

L = [1, 2, 3, 4, 5, 6, 7, 8, 9]
inverte(L, 0, len(L) - 1)
print(L)

```



## Quicksort

O algoritmo Quicksort segue o paradigma conhecido como “Dividir para Conquistar” pois ele quebra o problema de ordenar um vetor em subproblemas menores, mais fáceis e rápidos de serem resolvidos. Primeiramente, o método divide o vetor original em duas partes: os elementos menores que o pivô (tipicamente escolhido como o primeiro ou último elemento do conjunto). O método então ordena essas partes de maneira recursiva. O algoritmo pode ser dividido em 3 passos principais:

1. Escolha do pivô: em geral, o pivô é o primeiro ou último elemento do conjunto.
2. Particionamento: reorganizar o vetor de modo que todos os elementos menores que o pivô apareçam antes dele (a esquerda) e os elementos maiores apareçam após ele (a direita). Ao término dessa etapa o pivô estará em sua posição final (existem várias formas de se fazer essa etapa)

$\leq c$	$\leq c$	$\leq c$	$\leq c$	$c$	$\geq c$	$\geq c$	$\geq c$	$\geq c$	$\geq c$
----------	----------	----------	----------	-----	----------	----------	----------	----------	----------

3. Ordenação: recursivamente aplicar os passos acima aos sub-vetores produzidos durante o particionamento. O caso limite da recursão é o sub-vetor de tamanho 1, que não precisa ser ordenado.

Exemplo: mostre os passos necessários para a ordenação do seguinte vetor

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

1º passo: Definir pivô = 6 (último elemento)

2º passo: Particionar vetor (menores a esquerda e maiores a direita)

[5, 2, -3, 4, 1, 6, 13, 7, 15, 10]

3º passo: Aplicar 1 e 2 recursivamente para as metades

a) 2 metades

Metade 1: [5, 2, -3, 4, 1] → pivô = 1

[-3, 1, 5, 2, 4, 6, 13, 7, 15, 10]

Metade 2: [13, 7, 15, 10] → pivô = 10

[-3, 1, 5, 2, 4, 6, 7, 10, 15, 13]

b) 4 metades

Note que a metade 1 possui um único elemento: [-3] → já está ordenada

Metade 2: [5, 2, 4] → pivô = 4

[-3, 1, 2, 4, 5, 6, 7, 10, 15, 13]

Note que a metade 3 possui apenas um único elemento: [7] → já está ordenadas

Metade 4: [15, 13] → pivô = 13

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

c) 4 metades: Note que cada uma das 4 metades restantes contém um único elemento e portanto já estão ordenadas. Fim.

A seguir veremos uma implementação em Python para o algoritmo *Quicksort*.

```
# Implementa o quicksort recursivo para ordenar uma lista
def quicksort(L):
    if len(L) <= 1:
        return L
    m = L[0]      # pivô é o primeiro elemento da lista
    return quicksort([x for x in L if x < m]) + \
           [x for x in L if x == m] + \
           quicksort([x for x in L if x > m])
```

## Busca binária

Vimos anteriormente que a busca binária requer uma lista ordenada de elementos para funcionar. Basicamente, a ideia consiste em acessar o elemento do meio da lista. Se ele for o que desejamos buscar, a busca se encerra. Caso contrário, se o que desejamos é menor que o elemento do meio, a busca é realizada na metade a esquerda. Senão, a busca é realizada na metade a direita. A seguir mostramos um script em Python que implementa a versão recursiva da busca binária.

```
import random

# Função recursiva
def binary_search(lista, x, ini, fim):
    meio = ini + (fim - ini) // 2
    if ini > fim:
        return -1      # elemento não encontrado
    elif lista[meio] == x:
        return meio
    elif lista[meio] > x:
        print('Buscar na metade inferior')
        return binary_search(lista, x, ini, meio-1)
    else:
        print('Buscar na metade superior')
        return binary_search(lista, x, meio+1, fim)

# Cria lista com valores aleatórios
L = []
for i in range(1000):
    x = random.randint(1, 5000)
    while x in L:
        x = random.randint(1, 10000)
    L.append(x)

print('Lista de números: ')
print(L)

valor = int(input('Entre com o elemento a ser buscado: '))
```

```
# Ordena a lista
L.sort()
# Realiza a busca binária
y = binary_search(L, valor, 0, len(L)-1)

if y == -1:
    print('Elemento %d não encontrado.' %valor)
else:
    print('Elemento %d está na posição %d' %(valor, y))
```

Apesar da recursão ser uma ferramenta computacional muito poderosa e elegante para resolver uma série de problemas, há um problema que pode ocorrer em instâncias muito grandes dos problemas. Por exemplo, ao tentar ordenar um vetor com 10 milhões de elementos, a recursão será tão profunda que a pilha de recursão não caberá na memória do computador, causando a interrupção abrupta do processo e várias mensagens de erro. Esse fenômeno é conhecido como Stack Overflow. Existem técnicas computacionais para evitar que isso ocorra, mas elas estão fora do escopo dessa discussão.

“Não procure saber as respostas, procure compreender as perguntas.”  
(Confúcio)

## Aplicação: A recorrência logística e os sistemas caóticos

Um modelo matemático simples, porém capaz de gerar comportamentos caóticos e imprevisíveis é a recorrência logística (*logistic map*). Imagine que desejamos criar uma equação para modelar o número de indivíduos de uma população de coelhos a partir de uma população inicial. A equação mais simples seria algo do tipo:

$$x_{n+1} = r x_n$$

onde  $r > 0$  denota a taxa de crescimento. Porém, na natureza sabemos que devido a limitação de espaço e a disputa pelos recursos, populações não tendem a crescer indefinidamente. Há um ponto de equilíbrio em que o número de indivíduos tende a se estabilizar ao redor. Sendo assim, podemos definir a seguinte equação:

$$x_{n+1} = r x_n (1 - x_n)$$

em que o  $x_n \in [0,1]$  a porcentagem de indivíduos vivos e o termo  $(1 - x_n)$  tende a zero quando essa porcentagem se aproxima do valor máximo de 100%. Esse modelo é conhecido como recorrência logística. Veremos a seguir que fenômenos caóticos emergem desse simples modelo, que aparentemente possui um comportamento bastante previsível.

Primeiramente, note que o número de indivíduos no tempo  $n+1$  é uma função quadrática do número de indivíduos no tempo  $n$ , pois:

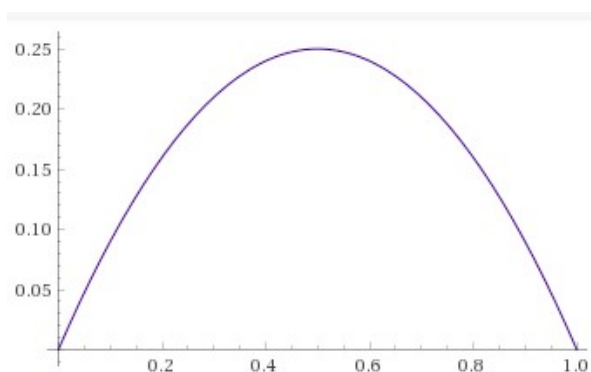
$$x_{n+1} = f(x_n) = -r x_n^2 + r x_n$$

ou seja, temos uma equação do segundo grau com  $a = -r$ ,  $b = r$  e  $c = 0$ . Como  $a < 0$ , a concavidade da parábola é para baixo, ou seja, ela admite um ponto de máximo. Derivando  $f(x_n)$  em relação a  $x_n$  e igualando a zero, temos o ponto de máximo:

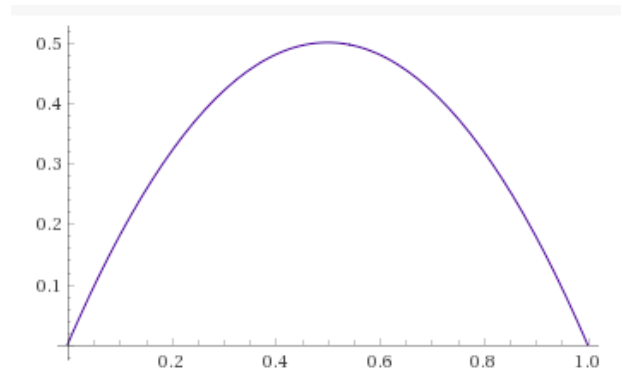
$$-2r x_n + r = 0$$

o que nos leva a  $x_n^* = \frac{1}{2}$ . Note que nesse ponto o valor da função vale:  $f(x_n^*) = -\frac{r}{4} + \frac{r}{2} = \frac{r}{4}$

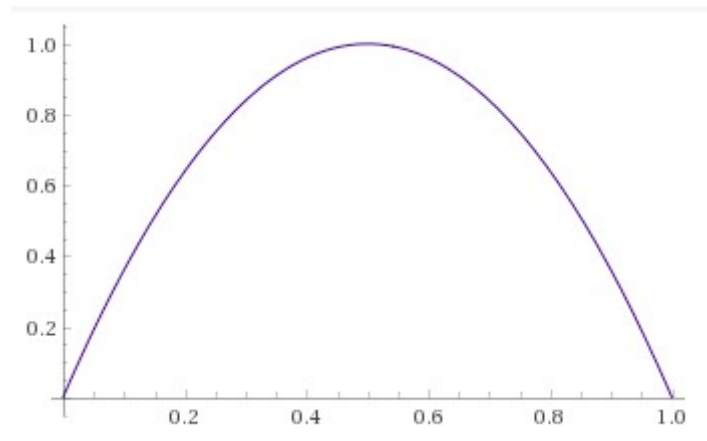
Note também que como  $c = 0$ ,  $f(0) = 0$ , ou seja, a parábola passa pela origem. Note ainda que  $f(1) = 0$ , ou seja a parábola corta o eixo  $x$  no ponto  $x = 1$ . De forma gráfica temos para alguns valores de  $r$  as seguintes parábolas:



$r = 1$



$r = 2$



$r = 4$

Vamos simular várias iterações do método em Python para analisar o comportamento do tamanho da população em função do tempo  $t$ . O script em Python a seguir mostra uma implementação computacional do modelo utilizando 100 iterações.

```
import matplotlib.pyplot as plt

# Número de iterações para atingir equilíbrio
MAX = 100

r = float(input('Entre com a constante r: '))
x = float(input('Entre com x0: '))

population = [x]

for i in range(1, MAX):
    x = r*x*(1 - x)
    population.append(x)

print('População no longo prazo: ', population[-1])

# Plota gráfico da população pelo tempo
eixox = list(range(MAX))
plt.figure(1)
plt.plot(eixox, population)
plt.show()
```

Execute o script e veja o que acontece para as entradas a seguir:

- a)  $r = 1$  e  $x_0 = 0.4$  (extinção)
- b)  $r = 2$  e  $x_0 = 0.4$  (equilíbrio em 50%)
- c)  $r = 2.4$  e  $x_0 = 0.6$  (pequena oscilação, mas atinge equilíbrio em 58%)
- d)  $r = 3$  e  $x_0 = 0.4$  (não há equilíbrio, população oscila, mas em torno de uma média)
- e)  $r = 4$  e  $x_0 = 0.4$  (comportamento caótico, totalmente imprevisível)

Em seguida, iremos estudar o que acontece com a população de equilíbrio conforme variamos o valor do parâmetro  $r$ . A ideia é que no eixo  $x$  iremos plotar os possíveis valores de  $r$  e no eixo  $y$  iremos plotar a população de equilíbrio para aquele valor de  $r$  específico. Iremos considerar que a população do equilíbrio é obtida depois de 1000 iterações. O script em Python a seguir mostra a implementação computacional dessa análise.

```

import matplotlib.pyplot as plt
import numpy as np

# Cria um vetor com todos os possíveis valores de r
R = np.linspace(0.5, 4, 20000)

m = 0.5

# Inicializa os eixos x e y vazios
X = []
Y = []
# Loop principal (iterar para todo r em R)
for r in R:
    # Adiciona r no eixo x
    X.append(r)

    # Escolhe um valor aleatório entre 0 e 1
    x = np.random.random()

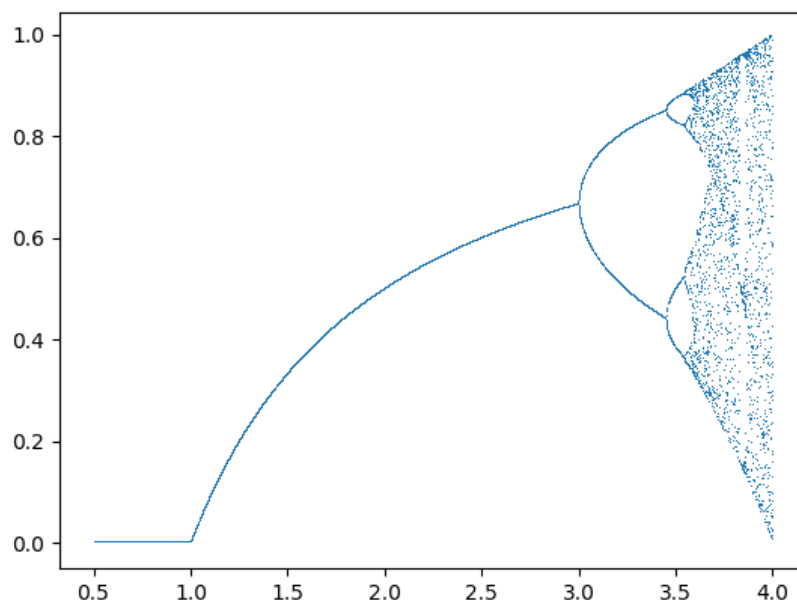
    # Gera população de equilíbrio
    for l in range(1000):
        x = r*x*(1-x)

    Y.append(x)

# Plota o gráfico sem utilizar retas ligando os pontos
plt.plot(X, Y, ls='', marker=',')
plt.show()

```

O gráfico plotado pelo script acima é conhecido como *bifurcation map*. Esse fenômeno da bifurcação ocorre como uma manifestação do comportamento caótico da população de equilíbrio para valores de  $r$  maiores que 3. Na prática, o que temos é que para um valor de  $r = 3.49999$ , a população de equilíbrio é muito diferente daquela obtida para  $r = 3.50000$  por exemplo. Pequenas perturbações no parâmetro  $r$  causam um efeito devastador na população de equilíbrio. Esse é o lema da teoria do caos, que pode ser parafraseado pela célebre sentença: o simples bater de asas de uma borboleta pode levar ao surgimento de um furacão, conhecido também como o efeito borboleta.



Uma das propriedades do caos é que é possível encontrar ordem e padrões em comportamentos caóticos. Por exemplo, a seguir iremos desenvolver um script em Python para plotar uma sequência de populações, começando de uma população inicial arbitrária e utilizando o valor de  $r = 3.99$ .

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

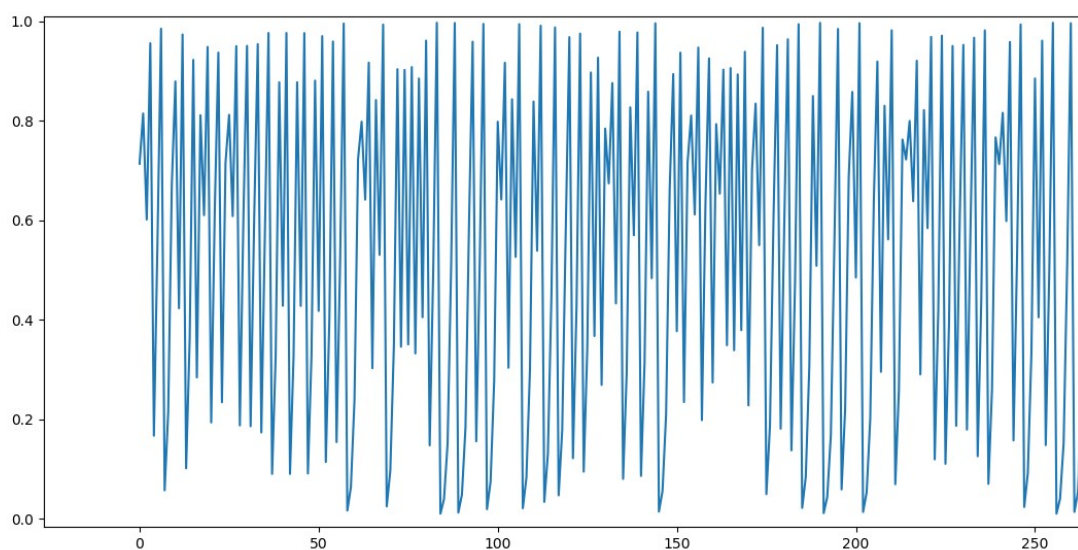
def atrator(X):
    A = X[:len(X)-2]
    B = X[1:len(X)-1]
    C = X[2:]
    #Plota atrator em 3D
    fig = plt.figure(2)
    ax = fig.add_subplot(111, projection='3d')
    ax.plot(A, B, C, '.', c='red')
    plt.show()

# Início do script
r = 3.99
x = np.random.random()
X = [x]

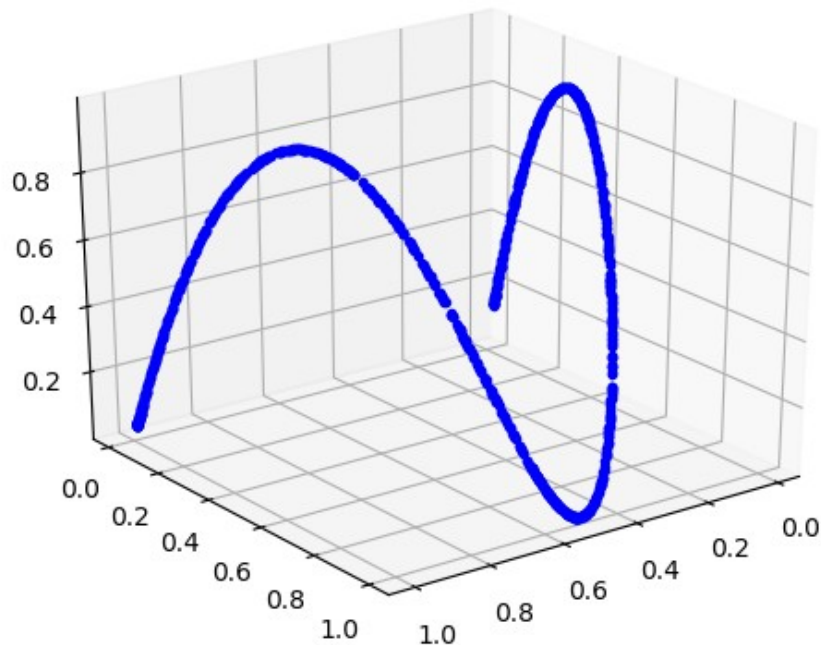
for i in range(1000):
    x = r*x*(1 - x)
    X.append(x)

# Plota o gráfico da sequência
plt.figure(1)
plt.plot(X)
plt.show()
```

Note que o gráfico mostrado na figura 1 parece o de uma sequência totalmente aleatória.



A seguir, iremos plotar cada subsequência  $(x_n, x_{n+1}, x_{n+2})$  como um ponto no  $\mathbb{R}^3$ . Na prática, isso significa que no eixo X iremos plotar a sequência original, no eixo Y iremos plotar a sequência deslocada de uma unidade e no eixo Z iremos plotar a sequência deslocada de duas unidades. Qual será o gráfico formado? Se de fato a sequência for completamente aleatória, nenhum padrão deverá ser observado, apenas pontos dispersos aleatoriamente pelo espaço. Mas, surpreendentemente, temos a formação do seguinte padrão, conhecido como o atrator do modelo.



Podemos repetir a análise anterior, mas agora plotando o ponto  $(x_n, x_{n+1})$  no plano. Conforme discutido anteriormente, vimos que  $x_{n+1} = f(x_n)$  é uma função quadrática, ou melhor, uma parábola. O experimento prático apenas comprova a teoria. Note que o gráfico obtido pelo script a seguir é exatamente a parábola. Conforme a teoria, note que o ponto de máximo ocorre em  $x_n = 0.5$ , e o valor da função nesse ponto,  $f(x_n)$ , é praticamente 1 ( $r/4 = 3.99/4$ ).

```
import matplotlib.pyplot as plt
import numpy as np

r = 3.99
x = np.random.random()      # a população é x minúsculo!
X = [x]                      # a lista é X maiúsculo!

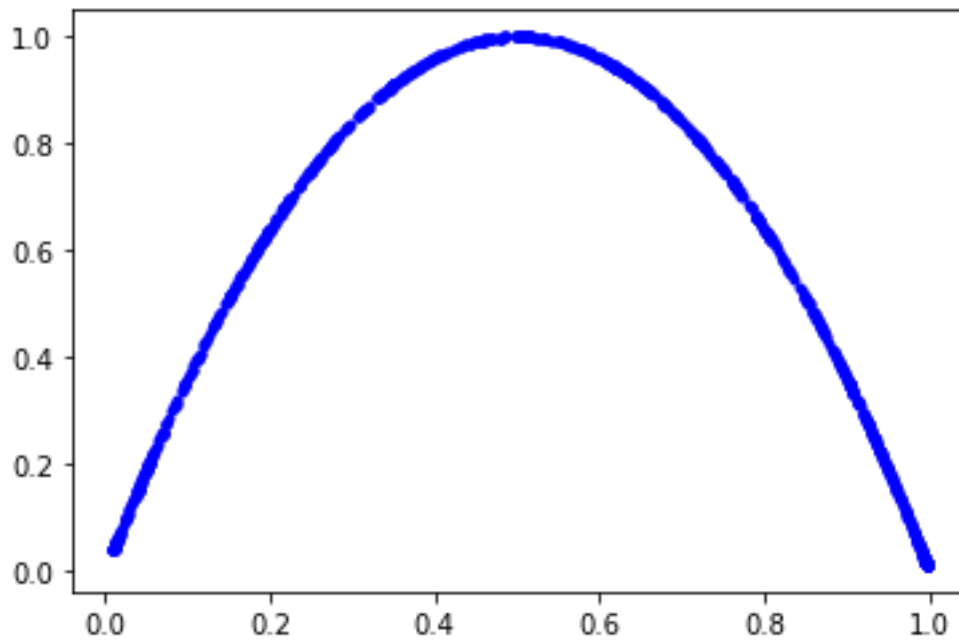
for i in range(1000):
    x = r*x*(1 - x)
    X.append(x)

# Plota o gráfico da sequência
plt.figure(1)
plt.plot(X)
plt.show()
```



```
A = X[:len(X)-1]
B = X[1:len(X)]

#Plota atrator em 3D
plt.figure(2)
plt.plot(A, B, '.', c='blue')
plt.show()
```



Interessante, não é mesmo? Dentro do caos, há ordem. Muitos fenômenos que observamos no mundo real parecem ser aleatórios, mas na verdade exibem comportamento caótico. A pergunta que fica é justamente essa: como distinguir um sistema aleatório de um sistema caótico? Como identificar os padrões que nos permitem enxergar a ordem em um sistema caótico? Para responder a esse questionamento precisamos mergulhar fundo na matemática dos sistemas complexos e da teoria do caos.

"Real difficulties can be overcome. It is only the imaginary ones that are unconquerable."  
(Theodore N. Vail)

## Aplicação: o dilema do prisioneiro

O Dilema dos Prisioneiros é um jogo muito famoso que representa bem o dilema entre cooperar e trair. Resumidamente, a estória é a seguinte. Dois suspeitos, A e B, são presos pela polícia. A polícia não tem provas suficientes para os condenar, então separa os prisioneiros em salas diferentes e oferece a ambos o mesmo acordo:

1. Se um dos prisioneiros confessar (trair o outro) e o outro permanecer em silêncio, o que confessou sai livre enquanto o cúmplice silencioso cumpre 10 anos.
2. Se ambos ficarem em silêncio (colaborarem um com o outro), a polícia só pode condená-los a 1 ano cada um.
3. Se ambos confessarem (traírem o comparsa), cada um leva 5 anos de cadeia.

Cada prisioneiro faz a decisão sem saber a escolha do outro - eles não podem conversar. Como o prisioneiro vai reagir? Existe alguma decisão racional a tomar? Qual seria a sua decisão?

### Matriz de ganhos

Uma forma esquemática para mostrar uma interação humana, ou seja, um jogo, é através de uma "matriz de resultados". Embora o enunciado do problema seja simples e intuitivo para entender de forma verbal, a representação gráfica oferece uma grande ajuda para visualizar o cenário de forma completa e entender as opções e implicações para cada jogador.

		Prisioneiro A	
		Colaborar (silêncio)	Trair (confessar)
Prisioneiro B	Colaborar (silêncio)	1 ano 1 ano	Livre 10 anos
	Trair (confessar)	10 anos Livre	5 anos 5 anos

Nesta figura você visualiza as duas opções de cada prisioneiro e o resultado de cada combinação de ação. Para cada célula, os valores vermelhos a direita referem-se ao Prisioneiro A; os azuis a esquerda referem-se ao Prisioneiro B. Estão descritos quantos anos cada prisioneiro ficará na cadeia. Neste cenário, quando menor o valor da pena, melhor para o prisioneiro.

Os prisioneiros não podem combinar a decisão (estão em salas isoladas e sem comunicação) e devem escolher simultaneamente. Cada jogador quer ficar preso o menor tempo possível, ou seja, maximizar seu resultado individual. Qual a melhor decisão? Considerando os incentivos deste jogo (os valores das penas para cada combinação de decisões na matriz), existe uma única decisão racional a tomar: trair. A explicação é a seguinte:

Imagine que você é o prisioneiro A. Assim, você raciocina nas duas hipóteses:

- Suponha que o Prisioneiro B escolha Colaborar. Então, se você escolher Colaborar, leva 1 ano de prisão. Se escolher Trair, sai livre. Neste caso, Trair é a melhor opção.

- Suponha que o Prisioneiro B escolha Trair. Então, se você escolher Colaborar, leva 10 anos de prisão. Se escolher Trair, fica com 5 anos. Neste caso, Trair é a melhor opção.

Perceba que Trair é a melhor opção em ambos os casos. Em outras palavras, Trair é a melhor opção independente da decisão do Prisioneiro B.

Agora, imagine o que o Prisioneiro B está pensando: se ele é racional como você, provavelmente a mesma coisa.

- Ele supõe que você vai escolher Colaborar. Então, se ele escolher Colaborar, leva 1 ano de prisão. Se escolher Trair, sai livre. Neste caso, Trair é a melhor opção.

- Ele supõe que você vai escolher Trair. Então, se ele escolher Colaborar, leva 10 anos de prisão. Se escolher Trair, fica com 5 anos de prisão. Neste caso, Trair é a melhor opção.

De novo, perceba que Trair é a melhor opção em ambos os casos.

### **O dilema: a escolha individual não é o melhor para ambos**

Em Teoria dos Jogos, chamamos que Trair é a Estratégia Dominante, ou seja, aquela que apresenta o melhor resultado independente da decisão do outro jogador. Quando em um certo jogo, devido o esquema de incentivos (a matriz de resultados) você não precisa se preocupar com a decisão alheia porque existe uma opção melhor independente do seu competidor, então você deve escolher a estratégia dominante.

Neste exemplo dos prisioneiros, como ambos vão escolher Trair devido a estratégia dominante, cada um é preso por 5 anos. Assim, dizemos que Trair-Trair é a solução de equilíbrio, equilíbrio do jogo ou Equilíbrio de Nash. O Equilíbrio de Nash é a solução (combinação de decisões) em que nenhum jogador pode melhorar seu resultado com uma ação unilateral. Ou seja, dado que Trair-Trair é a solução de equilíbrio (o resultado racional do jogo), se o Prisioneiro A mudar unilateralmente para Colaborar ele sai perdendo (15 anos), o mesmo ocorrendo para o Prisioneiro B.

O grande problema no Dilema dos Prisioneiros é que o equilíbrio (Trair-Trair) não é o melhor resultado pois existe um outro possível e melhor: se ambos escolherem Colaborar (ficar em silêncio) cada um ficaria com apenas um ano de prisão. Assim, o Dilema dos Prisioneiros é uma abstração de situações comuns onde a escolha do melhor individual conduz à traição mútua, enquanto que a colaboração proporcionaria melhores resultados. Por isso dizemos que o Dilema dos Prisioneiros resulta em um "equilíbrio ineficiente" pois o esquema de incentivos e racionalidade induz a um resultado pior.

Você poderia imaginar que este equilíbrio só ocorre porque as pessoas não podem conversar e combinar as ações, e que se pudessem fazer um acordo prévio, tudo se resolveria. Isso não é necessariamente verdade. Você quer colaborar (ficar em silêncio), mas quem garante que o seu parceiro fará o mesmo? O quanto você confia no outro jogador?

Você é o prisioneiro e sua vida está em jogo. Você combina antes que vai colaborar e quer cumprir sua palavra. Seu comparsa sabe isso. Então, o que garante que, no último instante, ele não vai te trair, justamente sabendo que você vai colaborar? Para ele é simples, ele sai livre e você pega 15 anos de prisão... Daí é tarde. Provavelmente, o seu comparsa pensará da mesma forma a seu

respeito. Por isso, o Dilema dos Prisioneiros se torna, na verdade, num Dilema da Confiança. Como resolver esse dilema?

### **Um exemplo ilustrativo**

O conflito típico dos jogos da categoria Dilema dos Prisioneiros é aquele em que cada jogador escolhe sua estratégia dominante e o resultado do jogo é pior para o grupo como um todo—é o conflito entre o interesse individual e o coletivo. Na prática, esse jogo-modelo é uma das metáforas mais poderosas da ciência do comportamento humano, pois inúmeras interações sociais e econômicas têm a mesma estrutura de incentivos (a matriz de resultados).

Imagine uma cidade com apenas dois postos de gasolina. Você é dono de um deles, chamado GASOIL, que fica ao lado do posto do seu concorrente, o AUTOGAS. Devido à proximidade dos dois, quando uma pessoa precisa abastecer o carro, ela vai até eles, confere os preços e escolhe o menor. Embora existam outras características que diferenciam os postos, como a cordialidade e a velocidade dos frentistas, considere por um momento que o preço é o fator mais relevante. Assim, se o critério é preço, alguns centavos a menos podem induzir parte dos clientes a preferir o posto que cobra o menor valor. Por exemplo, quem abaixar o preço em 5% ganha cerca de 30% dos clientes do concorrente. Esse aumento de volume de clientes compensa o preço reduzido, melhorando a rentabilidade, enquanto o outro perde faturamento. Por isso, você pensa: “Que tal abaixar o preço do litro de \$3 para \$2,90?”. Isso fará com que os habituais clientes do AUTOGAS (concorrente) passem a abastecer no GASOIL (o seu posto).

A vida empresarial seria mais fácil se as decisões fossem assim, isoladas. Entretanto, como o seu concorrente vai reagir? Ao notar que você abaixou o preço e ele perdeu clientes, ele também vai abaixar o preço para \$2,90. Como resultado, os dois postos terão preço igual (\$2,90 no lugar de \$3) e o mesmo volume de clientes, como antes, mas ambas as empresas perdem faturamento e lucro. Essa é a essência da guerra de preços, que prejudica o negócio dos dois postos.

Suponha que vocês tomem a decisão simultaneamente. Se hoje é domingo, vocês vão decidir o preço da segunda-feira. Durante o dia não é possível alterar o preço, mas apenas de um dia para outro. Vocês não se conversam e não sabem qual preço o outro vai adotar. Você ficará sabendo apenas no dia seguinte, e qualquer arrependimento será tarde demais—você terá de esperar pelo menos um dia inteiro para tomar qualquer providência, isto é, até o dia seguinte.

Considerando essa dinâmica de mercado com clientes sensíveis ao preço, os dois postos têm incentivos para abaixar o preço e ganhar mais momentaneamente. Entretanto, se os dois o fizerem, ambos saem perdendo. Assim, preventivamente, você conversa com o dono do AUTOGAS, e vocês combinam de não abaixar os preços. Ele concorda, mas você vai dormir com a dúvida: será que posso confiar nele? Se ele abaixar o preço à noite, você perderá toda a clientela do dia seguinte. Você está num dilema — o dilema da confiança, ou Dilema dos Prisioneiros.

Embora seja intuitivo, podemos representar, a seguir, a matriz de resultados dos postos de gasolina. Em cada célula (combinação de escolhas), o valor da esquerda refere-se aos ganhos do GASOIL, e o valor da direita aos ganhos do AUTOGAS. O valor em si é meramente ilustrativo, mas a proporção entre eles é relevante para a decisão.

		AUTOGAS	
		Manter preço	Reduzir preço
GASOIL	Manter preço	<div>\$50</div> <div>\$50</div>	<div>\$60</div> <div>\$30</div>
	Reduzir preço	<div>\$30</div> <div>\$60</div>	<div>\$40</div> <div>\$40</div>

Se ambos colaborarem (manterem o preço original), os dois ganham \$50 por dia. Se um deles abaixar o preço, recebe \$60, enquanto o que mantém recebe apenas \$30. Já se ambos reduzirem o preço, o resultado para cada um será \$40, pois significa abaixar o preço sem aumentar o volume de clientes. De acordo com a metodologia de análise no Dilema dos Prisioneiros, reduzir-reduzir é o ponto de equilíbrio (\$40, \$40), pois abaixar o preço é a estratégia dominante em cada um, resultando em valor pior se comparado àquele inicial.

Eles caíram na armadilha, e muitos chamam essas situações de dilema social—o interesse individual e a análise estritamente matemática e racional induzem a resultados piores do que opções que consideram o interesse coletivo. Como já foi mencionado, é difícil sair dessa armadilha—quem vai arriscar a colaborar (manter o preço), se há chance de o outro trair (reduzir o preço) e ganhar sozinho?

Para o leitor interessado, recomendamos acessar o link do livro “Estratégias de Decisão”, que é a fonte do conteúdo mostrado nessa seção em:

<http://estrategiasdedecisao.com/dilema-dos-prisioneiros/>

### Dilema do prisioneiro iterativo

O Dilema do Prisioneiro Iterativo (DPI) é uma referência bem conhecida para o estudo dos comportamentos de longo prazo de agentes racionais, como a cooperação que pode surgir entre agentes egoístas e independentes que precisam coexistir a longo prazo. Muitas estratégias bem conhecidas foram estudadas, formando a estratégia simples tit-for-tat (TFT), tornada famosa por Axelrod após seus torneios influentes, para outras mais envolvidas, como estratégias sem determinantes estudadas recentemente por Press e Dyson.

Na prática, o Dilema do Prisioneiro Iterativo é uma repetição do dilema do prisioneiro com as seguintes restrições na matriz de ganhos:

	C	T
C	(R, R)	(S, T)
T	(T, S)	(P, P)

a)  $T > R > P > S$ : isso torna (T, T) a estratégia dominante no dilema do prisioneiro.

b)  $2R > T + S$ : isso faz (C, C) a melhor escolha para os jogadores a longo prazo.  
A seguir definimos algumas estratégias que podem ser adotadas no DPI.

## Estratégias

1. Colaboração incondicional: sempre colabora.
2. Traição incondicional: sempre trai.
3. Aleatório: 50% de chance para colaborar, 50% de chance para trair.
4. Colaboração com probabilidade  $P$  = Traição com probabilidade  $(1 - P)$
5. Tit for Tat (TFT): coopera no 1º round e depois copia a ação anterior do oponente.
6. Suspicious TFT (STFT): trai no 1º round e depois copia a ação anterior do oponente.
7. Tit for Two Tats (TFTT): coopera sempre a menos que seja traído duas vezes em sequência.
8. Two Tits for Tats (TTFT): trai duas vezes seguidas após ser traído, caso contrário coopera.
9. Grim Trigger: Coopera, até que o oponente traia uma vez e então trai para o resto das jogadas.
10. Pavlov: Coopera se você e seu oponente tiveram estratégias iguais na jogada anterior, e trai se as jogadas anteriores foram diferentes.

A seguir mostramos uma implementação em Python do DPI com algumas estratégias selecionadas.

```
import numpy as np

MAX = 100

#####
# Definição das estratégias
#####

def always_cooperate():
    return 0

def always_defect():
    return 1

def random_player():
    return np.random.randint(2)

# Coopera na primeira escolha, depois copia a última escolha do oponente
def tit_for_tat(jogadas, i):
    if i == 0:
        return 0
    else:
        # última jogada
        return jogadas[-1]

# Coopera até que adversário traia, e depois sempre trai
def grim_trigger(jogadas):
    if 1 in jogadas:
        return 1
    else:
        return 0

# Coopera na primeira jogada ou se você e seu oponente tiveram
# estratégias iguais na jogada anterior e trai se as jogadas anteriores
# foram diferentes.
```

```

def pavlov(jogadas1, jogadas2, i):
    if i == 0 or jogadas1[-1] == jogadas2[-1]:
        return 0
    else:
        return 1

#####
# Início do script

# 0 = cooperar e 1 = trair
# Definição das matrizes de ganho
MA = np.array([[3,0],[5,1]])
MB = np.array([[3,5],[0,1]])

# Gera estratégias e computa os ganhos
# Lista de estratégias do playerA
playerA = []
# Lista de estratégias do playerA
playerB = []
# Ganhos iniciais são nulos
ganhoA = 0
ganhoB = 0

# Realiza MAX disputas entre os jogadores
for i in range(MAX):
    # Calcula estratégia do jogador A
    sp1 = random_player()
    playerA.append(sp1)
    # Calcula a estratégia do jogador B
    sp2 = grim_trigger(playerA)
    playerB.append(sp2)
    # Calcula os ganhos dos jogadores
    ganhoA += MA[sp1, sp2]
    ganhoB += MB[sp1, sp2]

print('Resultado')
if ganhoA > ganhoB:
    print('Jogador A venceu')
elif ganhoA < ganhoB:
    print('Jogador B venceu')
else:
    print('Empate')

print('Jogador A: %d ' %ganhoA)
print('Jogador B: %d ' %ganhoB)
print('Ganho total: %d' %(ganhoA+ganhoB))

```

Note que do ponto de vista individual, a estratégia `always_defect()`, vence qualquer outra estratégia. Porém, não é a melhor em termos de maximizar o ganho global. Por exemplo, se o jogador A adota a estratégia `random_player()` e o jogador adota a estratégia `tit_for_tat()`, em média o ganho global é superior aquele obtido quando um deles escolhe a estratégia `always_defect()`.

Execute o código para realizar a disputa entre dois jogadores que adotam as seguintes estratégias considerando 100 iterações:

- a) always\_defect x always\_defect
- b) always\_defect x always\_cooperate
- c) always\_defect x random\_player
- d) always\_defect x tit\_for\_tat
- e) always\_defect x grim\_trigger
- f) always\_defect x pavlov
- g) random\_player x tit\_for\_tat
- h) random\_player x grim\_trigger
- i) random\_player x pavlov

Para maiores detalhes sobre estratégias para o DPI, o leitor interessado pode acessar:

<http://jasss.soc.surrey.ac.uk/20/4/12.html>

"The major value in life is not what you get. The major value in life is what you become."  
(Jim Rohn)



## Aplicação: autômatos celulares

Modelos de autômatos celulares definem ferramentas computacionais muito importantes para a simulação e estudo de sistemas complexos. Um sistema é dito complexo quando suas propriedades não são uma consequência natural de seus elementos constituintes vistos isoladamente. As propriedades emergentes de um sistema complexo decorrem em grande parte da relação não-linear entre as partes. Costuma-se dizer de um sistema complexo que o todo é mais que a soma das partes. Exemplos de sistemas complexos incluem redes sociais, colônias de animais, o clima e a economia.

Uma pergunta recorrente no estudo de tais sistemas é: porque e como padrões complexos emergem a partir da interação entre os elementos? Como esses padrões evoluem com o tempo? Respostas a essas perguntas não são totalmente conhecidas, mas o estudo de modelos de autômatos celulares nos auxiliam no estudo e análise de tais sistemas. Aplicações práticas são muitas e incluem:

- Autômatos celulares e composição musical
- Autômatos celulares e modelagem urbana
- Autômatos celulares e propagação de epidemias
- Autômatos celulares e crescimento de câncer

Os primeiros modelos de autômatos celulares foram propostos originalmente na década de 40 por John Von Neumann e tinham como objetivos principais:

- Representar matematicamente a evolução natural em sistemas complexos
- Desenvolver máquinas de auto-replicação, através de um conjunto de regras matemáticas objetivas
- Estudar a auto-organização em sistemas complexos

Segundo Wolfram, autômatos celulares são formados por uma rede de células que possuem seus estados alterados num tempo discreto de acordo com seu estado anterior e o estado de suas células vizinhas. Algumas características importantes e comuns a todos os autômatos celulares são:

- Homogeneidade: as regras são iguais para todas as células
- Estados discretos: cada célula pode estar em um dos finitos estados
- Interações locais: o estado de uma célula depende apenas das células mais próximas (vizinhas)
- Processo dinâmico: a cada instante de tempo as células podem sofrer uma atualização de estado

Def: Um autômato celular é definido por uma 5-tupla de elementos

$$A = (R, S, S_0, V, F) \quad \text{onde}$$

$R$  é a grade de células (pode ser 1D, 2D, 3D,...)

$S$  é o conjunto de estados de uma célula  $c_i$

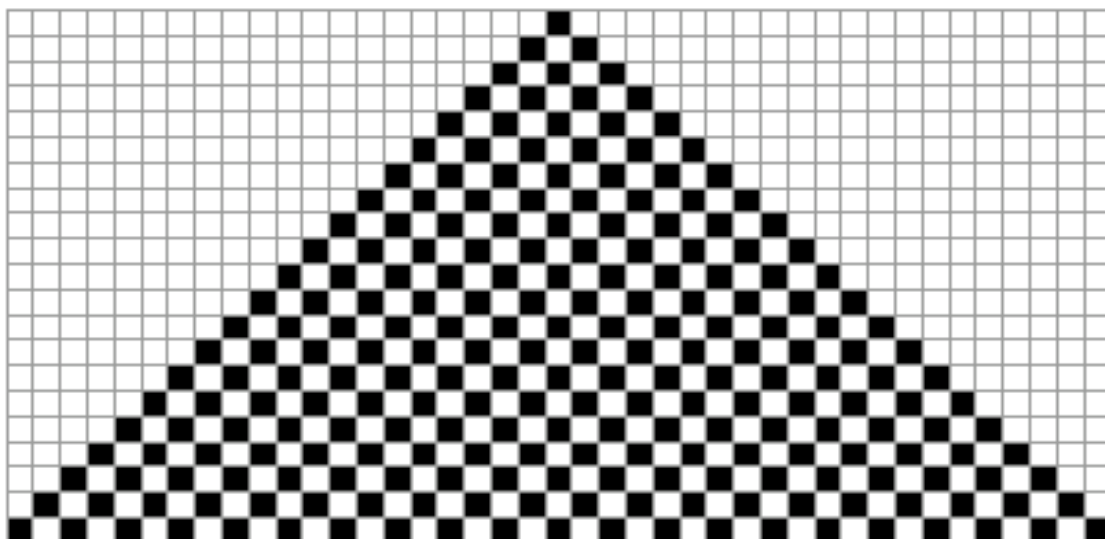
$S_0$  é o estado inicial do sistema

$V$  é conjunto vizinhança (define quem são os vizinhos de cada célula)

$F$  é a função de transição (regras de governam a evolução do sistema no tempo)

### Autômatos Celulares Elementares

Um autômato celular é dito elementar se o reticulado de células  $R$  é unidimensional (ou seja, pode ser representado por um vetor), o conjunto de estados  $S = \{0, 1\}$  e o conjunto vizinhança engloba apenas duas células: a anterior ( $i-1$ ) e a posterior ( $i+1$ ). Tipicamente, se uma célula assume estado 0 dizemos que ela está morta e se ela assume estado 1 dizemos que está viva. A figura a seguir ilustra as primeiras 20 gerações de um autômato celular elementar, em que no início apenas uma célula está viva (preto = vivo, branco = morto)

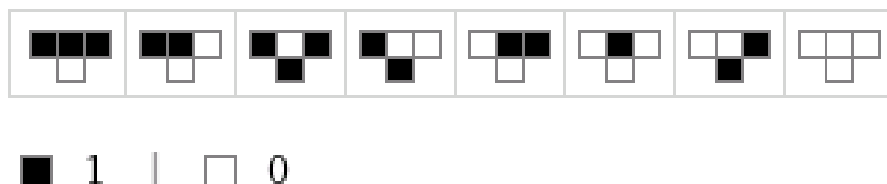


A questão é: como evoluir uma configuração de forma a construir esse padrão? Que regras são aplicadas para definir quais células vivem ou morrem na próxima geração?

A função de transição do autômato da figura é dada pela seguinte tabela.

$x_t(i-1)$	$x_t(i)$	$x_t(i+1)$	$x_{t+1}(i)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Uma forma de resumir toda essa tabela é através da seguinte representação:



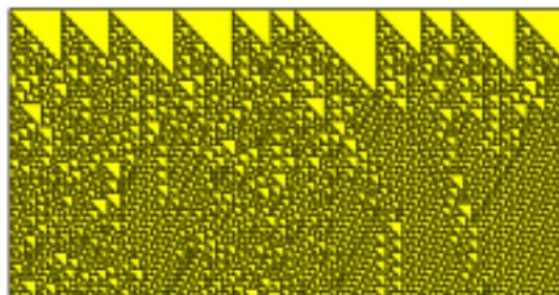
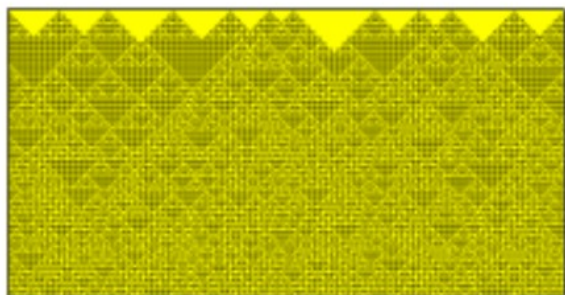
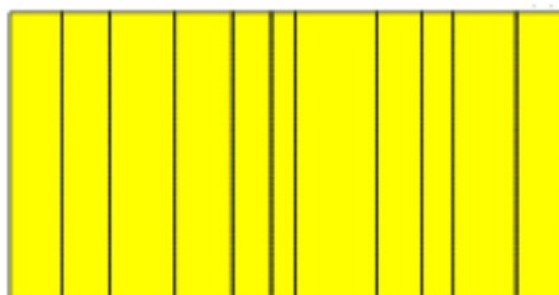
O que essa regra nos diz pode ser sumariado em 8 fatos:

- 1) sempre que a célula  $i$  for morta e a  $(i-1)$  e  $(i+1)$  também forem, a célula  $i$  permanecerá morta na próxima geração.
- 2) sempre que a célula  $i$  for morta, a  $(i-1)$  for morta e a  $(i+1)$  for viva, a célula  $i$  viverá na próxima geração.
- 3) sempre que a célula  $i$  for viva e a  $(i-1)$  e a  $(i+1)$  forem mortas, a célula  $i$  morrerá na próxima geração.
- 4) sempre que a célula  $i$  for viva, a  $(i-1)$  for morta e a  $(i+1)$  for viva, a célula  $i$  morrerá na próxima geração.

- 5) sempre que a célula  $i$  for morta, a  $(i-1)$  for viva e a  $(i+1)$  for morta, a célula  $i$  viverá na próxima geração.
- 6) sempre que a célula  $i$  for morta e ambas  $(i-1)$  e  $(i+1)$  forem vivas, a célula  $i$  viverá na próxima geração.
- 7) sempre que a célula  $i$  for viva, a  $(i-1)$  for viva e a  $(i+1)$  for morta, a célula  $i$  morrerá na próxima geração.
- 8) sempre que a célula  $i$  for viva e ambas  $(i-1)$  e  $(i+1)$  forem vivas, a célula  $i$  morrerá na próxima geração.

Note que não existem mais combinações possíveis de 0's e 1's usando apenas 3 bits, pois conseguimos contar em binário de 0 a 7, o que resulta em 8 possibilidades. Essa regra tem um nome: é a regra 50, pois o número binário correspondente a última coluna da função de transição vale 00110010, que em binário é justamente o número 50. Sendo assim, quantas possíveis regras existem para um autômato celular elementar? Basta computar  $2^8$ , que resulta em 256. Portanto, o número total de regras distintas é 256. Por essa razão dizemos que existem 256 autômatos celulares elementares distintos, um para cada regra. O interessante é estudar e simular o que acontece com cada um desses autômatos durante sua evolução. De acordo com Wolfram, existem 4 classes de regras para um autômato celular elementar:

- Classe 1: Estado Homogêneo  
Todas as células chegarão num mesmo estado após um número finito de estados
- Classe 2: Estável simples  
As células não possuem todas o mesmo estado, mas eles se repetem com a evolução temporal
- Classe 3: Padrão irregular  
Não possui padrão reconhecível
- Classe 4: Estrutura complexa  
Estruturas complexas que evoluem imprevisivelmente



As regras mais interessantes são as da classe 4, pois definem um sistema complexo com propriedades dinâmicas interessantes, sendo algumas delas capazes até de simular máquinas de Turing, que são modelos computacionais capazes de serem programadas para realizar diferentes tarefas computacionais. Um exemplo de regra com essa característica é a regra 110. (Referências: [https://en.wikipedia.org/wiki/Rule\\_110](https://en.wikipedia.org/wiki/Rule_110), <http://www.complex-systems.com/pdf/15-1-1.pdf>)

Exercício: Construa a função de transição do autômato celular elementar definido pela regra 30. Aplique a regra para evoluir a condição inicial idêntica a da figura da regra 50 (apenas uma célula viva) por 20 gerações. Repita o exercício mas agora para a regra 110.

A seguir é apresentado um algoritmo para a simulação de autômatos celulares elementares.

```
geração = vetor(N) (N é o número de células)

nova_geração = vetor(N)

evolução = matriz(MAX, N) (MAX é o número de gerações)

Inicializar geração (setar configuração inicial)

para i = 1 até MAX
    evolução[i,:] = geração
    # Percorre cada célula da geração atual
    para j = 1 até N
        Aplicar regra de transição na célula j, gerando nova_geração
    geração = nova_geração

Plotar resultados
```

Ex: Baseado no algoritmo anterior, implementar um script em Python que, dado uma regra (número de 0 a 255), evolua uma configuração inicial de tamanho N = 1000 até a geração 500.

```
import numpy as np
import matplotlib.pyplot as plt

# converte um número inteiro para sua representação binária (0-255)
def converte_binario(numero):
    binario = bin(numero)
    binario = binario[2:]
    if len(binario) < 8:
        zeros = [0]*(8-len(binario))
        binario = zeros + list(binario)
    return list(binario)

# Início do script
MAX = 500
g = np.zeros(1000)
ng = np.zeros(1000)

regra = int(input('Entre com o número da regra: '))
codigo = converte_binario(regra)
```

```

# Matriz em que cada linha armazena uma geração do autômato
matriz_evolucao = np.zeros((MAX, len(g)))

# Define geração inicial
g[len(g)//2] = 1

# Laço principal: atualiza as gerações
for i in range(MAX):

    matriz_evolucao[i,:] = g

    # Percorre células da geração atual
    for j in range(len(g)):

        if (g[j-1] == 0 and g[j] == 0 and g[(j+1)%len(g)] == 0):
            ng[j] = int(codigo[7])
        elif (g[j-1] == 0 and g[j] == 0 and g[(j+1)%len(g)] == 1):
            ng[j] = int(codigo[6])
        elif (g[j-1] == 0 and g[j] == 1 and g[(j+1)%len(g)] == 0):
            ng[j] = int(codigo[5])
        elif (g[j-1] == 0 and g[j] == 1 and g[(j+1)%len(g)] == 1):
            ng[j] = int(codigo[4])
        elif (g[j-1] == 1 and g[j] == 0 and g[(j+1)%len(g)] == 0):
            ng[j] = int(codigo[3])
        elif (g[j-1] == 1 and g[j] == 0 and g[(j+1)%len(g)] == 1):
            ng[j] = int(codigo[2])
        elif (g[j-1] == 1 and g[j] == 1 and g[(j+1)%len(g)] == 0):
            ng[j] = int(codigo[1])
        elif (g[j-1] == 1 and g[j] == 1 and g[(j+1)%len(g)] == 1):
            ng[j] = int(codigo[0])

    g = ng.copy() # se não usar copy ambos vetores tornam-se o mesmo

# plota matriz resultante como imagem
plt.figure(1)
plt.axis('off')
plt.imshow(matriz_evolucao, cmap='gray')
plt.savefig('Automata.png', dpi=300)
plt.show()

```

## Autômatos celulares 2D

### O Jogo da Vida

O autômato 2D mais conhecido sem dúvida é o jogo da vida, criado por Conway para simular a evolução de sistemas complexos a partir de regras determinísticas. O reticulado 2D é representado computacionalmente por uma matriz geralmente quadrada de células que podem estar vivas ou mortas. A função de vizinhança é definida pela vizinhança de Moore, ou seja, pelas 8 células mais próximas a uma dada célula  $i$ , conforme ilustra a figura a seguir.

A função de transição do jogo da vida tem como conceito imitar processos de nascimento e morte. A ideia básica é que um ser vivo necessita de outros seres vivos para sobreviver e procriar, mas um excesso de densidade populacional provoca a morte do ser vivo devido à escassez de recursos.

## Two-dimensional cellular automata

1	0	1	0	1	0
0	0	1	0	1	1
1	1	1	0	1	1
1	0	1	0	1	0
0	0	0	1	1	0
1	1	0	0	1	0
1	1	1	0	0	0
1	0	1	1	1	1

a neighborhood  
of 9 cells

São 4 regras básicas:

R1 (Sobrevivência) – uma célula viva com 2 ou 3 células vizinhas vivas, permanece viva na próxima geração.

R2 (Morte por isolamento) – uma célula viva com 0 ou 1 vizinho vivo morre de solidão na próxima geração.

R3 (Morte por sufocamento) – uma célula viva com 4 ou mais vizinhos vivos morre por sufocamento na próxima geração.

R4 (Renascimento) – uma célula morta com exatamente 3 vizinhos vivos, renasce na próxima geração.

Isso nos leva a regra de transição conhecida como B3S23, uma vez que no código proposto B significa born (nascer) e S significa survive (sobreviver). Em outras palavras, nesse autômato em particular, uma célula nasce sempre que possui 3 vizinhas vivas ao redor e sobrevive se possui 2 ou 3 vizinhas vivas ao redor. Outras variantes de regras incluem: B15S257, B147S256, B34S4567, etc. Cada uma das regras define um autômato diferente. Ao autômato cuja regra é B3S23 dá-se o nome de Jogo da Vida.

É interessante perceber que a regra B3S23 a partir de diversas inicializações simples exibe um comportamento altamente complexo, onde padrões complexos e “seres vivos” passam a interagir de maneira bastante inesperada. Trata-se de um conjunto de regras totalmente determinísticas que levam a um comportamento completamente imprevisível (ordem x caos).

Para uma coletânea de condições iniciais verifique o link:

<https://jakevdp.github.io/blog/2013/08/07/conways-game-of-life/>

Um problema comum que afeta simulações computacionais do jogo da vida é o chamado problema de valor de contorno. Isso nada mais é que uma falha ao se definir a função de transição para células na borda do sistema. Para se evitar essa indefinição, considera-se que o reticulado é na verdade um toro. Isso significa que não existem bordas, uma vez que a borda da esquerda é ligada a borda da direita, assim como a inferior é ligada a superior.

Ex: Implementar um script em Python que, dada uma configuração inicial, simule o jogo da Vida num tabuleiro de dimensões 100 x 100 por 200 gerações.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import time

# cria inicialização: rpentomino nas coordenadas i, j
def init_config(tabuleiro, padrao, i, j):
    linhas = padrao.shape[0]
    colunas = padrao.shape[1]
    tabuleiro[i:i+linhas, j:j+colunas] = padrao

# Início do script
inicio = time.time()
MAX = 200
SIZE = 100

geracao = np.zeros((SIZE, SIZE))
nova_geracao = np.zeros((SIZE, SIZE))

# Cubo de dados em que cada fatia representa uma geração
matriz_evolucao = np.zeros((SIZE, SIZE, MAX))

# Define geração inicial

unbounded = np.array([[1, 1, 1, 0, 1],
                      [1, 0, 0, 0, 0],
                      [0, 0, 0, 1, 1],
                      [0, 1, 1, 0, 1],
                      [1, 0, 1, 0, 1]])

glider = np.array([[1, 0, 0],
                  [0, 1, 1],
                  [1, 1, 0]])

r_pentomino = np.array([[0, 1, 1],
                       [1, 1, 0],
                       [0, 1, 0]])

diehard = np.array([[0, 0, 0, 0, 0, 0, 1, 0],
                    [1, 1, 0, 0, 0, 0, 0, 0],
                    [0, 1, 0, 0, 0, 1, 1, 1]])

acorn = np.array([[0, 1, 0, 0, 0, 0, 0],
                 [0, 0, 0, 1, 0, 0, 0],
                 [1, 1, 0, 0, 1, 1, 1]])

init_config(geracao, r_pentomino, SIZE//2, SIZE//2)
```

```

# Laço principal (atualiza gerações)
for k in range(MAX):

    print('Processando geração %d...' %k)
    matriz_evolucao[:, :, k] = geracao

    # Laço principal: atualiza as gerações
    for i in range(SIZE):

        for j in range(SIZE):

            vivos = geracao[i-1, j-1] + geracao[i-1, j] + \
                    geracao[i-1, (j+1)%SIZE] + geracao[i, j-1] + \
                    geracao[i, (j+1)%SIZE] + geracao[(i+1)%SIZE, j-1] + \
                    geracao[(i+1)%SIZE, j] + geracao[(i+1)%SIZE, (j+1)%SIZE]

            if (geracao[i,j] == 1):
                if (vivos == 2 or vivos == 3):
                    nova_geracao[i,j] = 1
                else:
                    nova_geracao[i,j] = 0
            else:
                if (vivos == 3):
                    nova_geracao[i,j] = 1

        geracao = nova_geracao.copy()

fim = time.time()
print('Tempo gasto na simulação: %.2f s' %(fim-inicio))

# Gera animação da evolução do sistema
fig = plt.figure(1)
plt.axis('off')
lista = []
for i in range(MAX):
    im = plt.imshow(matriz_evolucao[:, :, i], cmap='gray')
    lista.append([im])

ani = animation.ArtistAnimation(fig, lista, interval=100, blit=True,
                                repeat_delay=1000)

ani.save('r_pentomino.gif', writer='imagemagick')

plt.show()

```

"Fear has a large shadow, but he himself is small."  
(Ruth Gendler)



## Bibliografia

MENEZES, N. N. C. Introdução à programação com Python: algoritmos e lógica de programação para iniciantes. 2. ed. São Paulo: Novatec, 2014.

BORGES, L. E.; Python para Desenvolvedores. 2. ed., 2010. Disponível em [https://ark4n.files.wordpress.com/2010/01/python\\_para\\_desenvolvedores\\_2ed.pdf](https://ark4n.files.wordpress.com/2010/01/python_para_desenvolvedores_2ed.pdf)

MILLER, B.; RANUM, D.; Como Pensar como um Cientista da Computação: Versão Interativa. Disponível em: <https://panda.ime.usp.br/pensepy/static/pensepy/index.html>

MILLER, B., RANUM, D. Problem Solving with Algorithms and Data Structures using Python. Disponível em: <https://runestone.academy/runestone/books/published/pythonds/index.html>

MASANORI, F.; Python para Zumbis (curso online). Material didático em: <https://github.com/fmasanori/PPZ>

Comunidade Python Basil  
<https://wiki.python.org.br/ListaDeExercicios>

Curso online de Python em Codecademy - <http://www.codecademy.com/pt-BR/tracks/python>

PILGRIM, M.. Dive into Python, Apress, 2004 - disponível em <http://www.diveintopython.net/>

Repositório online de livros sobre programação Python - <http://pythonbooks.revolunet.com/>

MATTHES, E.; Curso Intensivo de Python: Uma Introdução Prática e Baseada em Projetos à Programação, Novatec, 2016.

RAMALHO, L; Python Fluente: Programação Clara, Concisa e Eficaz, Novatec, 2015.

LUTZ, M.; ASCHER, D.; Aprendendo Python. 2. ed., Bookman, 2007.

BARRICHELO, F. Estratégias de decisão: decida melhor com insights da Teoria dos Jogos  
<http://estrategiasdedecisao.com/>

WOLFRAM, S. A New Kind of Science, Wolfram Research, 2002.

Game of Life in Python  
<https://jakevdp.github.io/blog/2013/08/07/conways-game-of-life/>

Autômatos celulares  
[https://en.wikipedia.org/wiki/Rule\\_110](https://en.wikipedia.org/wiki/Rule_110),  
<http://www.complex-systems.com/pdf/15-1-1.pdf>

MAY, R. M. "Simple mathematical models with very complicated dynamics". *Nature*. **261** (5560): 459–467, 1976. Available at: [http://abel.harvard.edu/archive/118r\\_spring\\_05/docs/may.pdf](http://abel.harvard.edu/archive/118r_spring_05/docs/may.pdf)

## Sobre o autor

Alexandre L. M. Levada é bacharel em Ciências da Computação pela Universidade Estadual Paulista “Júlio de Mesquita Filho” (UNESP), mestre em Ciências da Computação pela Universidade Federal de São Carlos (UFSCar) e doutor em Física Computacional pela Universidade de São Paulo (USP). Atualmente é professor adjunto no Departamento de Computação da Universidade Federal de São Carlos e seus interesses em pesquisa são: filtragem de ruído em imagens e aprendizado de métricas via redução de dimensionalidade para problemas de classificação de padrões. Para maiores detalhes: <https://sites.google.com/view/alexandre-levada/>