# EAE1106 - Métodos Computacionais para Economia

Danilo Souza

# Índice

**Bem-vindo** **5**
Agradecimentos e reprodução . . . . . 5

**Introdução** **6**

**I Alfabetização computacional e lógica básica** **7**

**1 Fundamentos de computação** **9**

**2 Primeiros passos no Python** **10**

**3 Programação e IA** **11**

**4 Tipos primitivos e objetos básicos** **12**
4.1 Strings . . . . . 12
4.1.1 Operações básicas . . . . . 13
4.1.2 Métodos de strings . . . . . 14
4.1.3 Formatação e a instrução print . . . . . 15
4.1.4 Introdução a expressões regulares . . . . . 16
4.2 Listas . . . . . 18
4.2.1 Operações básicas . . . . . 18
4.2.2 Métodos de listas . . . . . 20
4.2.3 Operadores lógicos e variáveis booleanas . . . . . 21
4.2.4 Execução condicional . . . . . 23
4.2.5 Execução alternativa . . . . . 23
4.2.6 Condicionais encadeadas . . . . . 24
4.3 Tuplas . . . . . 24
4.3.1 Operações básicas . . . . . 25
4.3.2 Atribuição de tuplas . . . . . 26
4.3.3 Tuplas como valores de retorno . . . . . 27
4.3.4 Operações nativas com tuplas . . . . . 28
4.4 Dicionários . . . . . 28
4.4.1 Operações básicas . . . . . 30
4.4.2 Métodos de dicionários . . . . . 31
4.5 Exercícios . . . . . 32

**5 Controle de fluxo e iteração** **34**

**6 Funções** **35**
6.1 O que é uma função? . . . 35
6.2 Nossa primeira função . . . 36
6.3 Argumentos de uma função . . . 36
6.3.1 *Keyword arguments* e *default values* . . . 37
6.3.2 Argumentos arbitrários . . . 38
6.3.3 Elementos dentro da função: execução condicional . . . 39
6.3.4 Aplicação: Teorema Central do Limite - Parte 1 . . . 39
6.3.5 Elementos dentro da função: iteração . . . 40
6.3.6 Aplicação: Teorema Central do Limite - Parte 2 . . . 41
6.4 Valor de retorno de uma função . . . 41
6.4.1 Expectativa de vida de variáveis dentro da função . . . 42
6.4.2 Aplicação: Teorema Central do Limite - Parte 3 . . . 43
6.5 Documentação . . . 45
6.6 Funções anônimas . . . 46

**II Computação numérica, análise de dados e visualização** **48**

**7 Arrays, matrizes e álgebra linear** **50**
7.1 O que é o NumPy? . . . 50
7.2 Elementos básicos do NumPy . . . 51
7.2.1 Arrays unidimensionais . . . 51
7.2.2 Arrays multidimensionais . . . 53
7.2.3 Propriedades de arrays . . . 54
7.3 Operações básicas com arrays . . . 56
7.3.1 Operações aritméticas simples . . . 56
7.3.2 Funções universais . . . 61
7.3.3 Arrays e listas . . . 63
7.4 Aplicação: solução de sistema de equações lineares . . . 64
7.4.1 Algoritmo de eliminação de Gauss-Jordan . . . 64
7.4.2 Solução de sistemas exatamente identificados . . . 68

**8 Gestão e análise de dados** **71**
8.1 O que é o NumPy? . . . 71

**9 Visualização** **72**
9.1 O que é o NumPy? . . . 72

**10 Análise empírica integrada** **73**
10.1 O que é o NumPy? . . . 73

**III Temas complementares** **74**

**11 Introdução à programação orientada a objetos (OOP)** **76**

**12 Introdução ao R** **77**

**References** **78**

# Bem-vindo

Este é o material da disciplina **EAE1106 — Métodos Computacionais para Economia**, ministrada no Departamento de Economia da USP.

O curso tem como objetivo introduzir lógica de programação e análise de dados utilizando principalmente **Python**, com uma breve introdução a **R** ao final do semestre.

## Agradecimentos e reprodução

Esse material deve muito a X, Y e Z

Esse material foi construído bla bla bla. Qualquer erro é de responsabilidade exclusiva do autor. Todo e qualquer feedback é muito bem-vindo.

# Introdução

Este é o material da disciplina **EAE1106 — Métodos Computacionais para Economia**, ministrada no Departamento de Economia da USP.

O curso tem como objetivo introduzir lógica de programação e análise de dados utilizando principalmente **Python**, com uma breve introdução a **R** ao final do semestre.

# Parte I

# Alfabetização computacional e lógica básica

Este é o material da disciplina **EAE1106 — Métodos Computacionais para Economia**, ministrada no Departamento de Economia da USP.

O curso tem como objetivo introduzir lógica de programação e análise de dados utilizando principalmente **Python**, com uma breve introdução a **R** ao final do semestre.

# 1 Fundamentos de computação

In summary, this book has no content whatsoever.

# 2 Primeiros passos no Python

In summary, this book has no content whatsoever.

# 3 Programação e IA

In summary, this book has no content whatsoever.

# 4 Tipos primitivos e objetos básicos

In summary, this book has no content whatsoever.

---

## 4.1 Strings

Strings não são como números inteiros ou de ponto flutuate. No Python, uma string nada mais é do que uma sequência ordenada de caracteres unicode. Eles são delimitados sempre por aspas (simplas ou duplas). Relembrando nosso primeiro "programa" da aula passada podemos atribuir a uma variável a string *Hello, World* usando aspas.

```
str1 = 'Hello, World'

print(str1)
print(type(str1))
```

Usando aspas duplas o resultado seria o mesmo

```
str2 = "Hello, World"

print(str2)
print(type(str2))
```

Note que podemos também usar uma sequência de três aspas duplas e escrever strings que percorrem várias linhas.

```
str3 = """Das Utopias

Se as coisas são inatingíveis...ora!
Não é motivo para não querê-las...
Que tristes os caminhos, se não fora
A presença distante das estrelas!
```

```
Mario Quintana
"""

print(str3)
print(type(str3))
```

### 4.1.1 Operações básicas

Por ser uma sequência de caracteres e não um número, operações aritméticas (em geral) não são permitidas, mas outras operações, como *slicing*, o são. É possível acessar um caracter específico da sequência utilizando a posição desse caractere, utilizando o que chamamos de **índice**. No caso do string *'Hello, World'*, para acessar a segunda letra podemos utilizar colchetes após o string e dentro dele o índice referente à posição do 'e' no string:

```
print(str1)
```

```
str1[2]
```

Ué, mas porque que obtivemos como resposta o caractere 'l', que é o 3º na sequência, e não o 'e', que é o 2º elemento? Aqui vai uma particularidade da sintaxe do Python: **em se tratando de índices, o Python sempre começa a contagem em 0!**. Dessa forma, para acessar o primeiro caractere de *'Hello, World'* devemos pedir `str1[0]`, para acessar o segundo é preciso pedir `str1[1]` e assim por diante.

```
str1[1]
```

Podemos acessar contando de trás para frente e usando um índice negativo

```
str1[-11]
```

É possível que o índice seja uma expressão, mas deve sempre ser um valor inteiro.

```
n=0
str1[n+1]
```

```
str1[1.5]
```

Para acessar uma **fatia** (*slice*) do string podemos utilizar um intervalo de índices (inclusive o índice inicial e exclusive o índice final). Por exemplo,

```
str1[0:5]
```

Além de ser possível calcular o número de caracteres em uma sequência string utilizando a função integrada `len()`, podemos "concatenar" ou "somar" strings utilizando apenas o sinal de +

```
len(str1)
```

```
str1 + ' --- ' + str2
```

Note, porém, que strings são **imutáveis**, de modo que caso queira substituir um dos caracteres dentro de um string é preciso utilizar uma função específica para isso ou criar um novo string derivado do 1º. Apenas tentar substituir um dos caracteres de um string já definido não é permitido.

```
# Tentemos substituir o 'e' do str1 por 'a'
str1[1] = 'a'
```

### 4.1.2 Métodos de strings

As strings oferecem métodos que executam várias operações úteis. Um método é em essência uma sequência de instruções encapsuladas dentro de um único comando que recebe argumentos e devolve um valor. Embora a sintaxe seja diferente, a ideia é a mesma quando falamos de funções, objeto do nosso estudo daqui algumas aulas.

No caso dos métodos, temos que passar o nome da string que foi definida anteriormente seguida de '.' e depois do comando relacionado ao método específico. Dentre os principais métodos aplicáveis a strings e suas funcionalidades podemos citar:

1. **str_example.upper()**: devolve a string 'str_example' toda em letras maiúsculas.
2. **str_example.lower()**: devolve a string 'str_example' toda em letras minúsculas.
3. **str_example.strip()**: devolve a string 'str_example' retirando possíveis espaços em branco no início e no fim da string.
4. **str_example.startswith('xyz')**: testa se a string 'str_example' começa com a string 'xyz'.
5. **str_example.endswith('xyz')**: testa se a string 'str_example' termina com a string 'xyz'.
6. **str_example.find('xyz')**: procura a string 'xyz' dentro de 'str_example' e retorna o primeiro índice onde 'xyz' começa ou retorna -1 se nada for encontrado.

7. **str_example.replace('old','new')**: retorna uma string nova onde todas as ocorrências de 'old' encontradas em 'str_example' serão substituídas por 'new'.

Além desses principais, existem outros vários métodos para strings. Esse link é um bom ponto de partida para quem quiser conhecer outros exemplos.

```
str1 = ' Hello, World '

print(str1.upper())
print(str1.lower())
```

```
print(str1.strip())
```

```
print(str1.startswith(' '))
print(str1.endswith('d'))
```

```
print(str1.find(','))
```

```
print(str1.replace('Hello','World'))
```

### 4.1.3 Formatação e a instrução print

Um método sobre o qual não falamos, mas que é bastante interessante quando queremos, por exemplo, printar bonitinho o resultado de determinada operação é `s.format()`. Com esse método podemos converter uma variável numérica para uma formato específico e printá-la dentro de um string maior. Imagine, por exemplo, que estejamos interessados em printar o valor de $\pi$ arredondado para 2 casas decimais apenas dentro de um string que diz isso. Podemos implementar isso da seguinte forma

```
pi = 3.1415926535
print('O valor de pi arredondado para 2 casas decimais é {:.2f}. Interessante, não?'.format(p
```

Os colchetes dentro do string mostram onde que o número deve aparecer. Mais do que isso, definimos dentro do colchete o formato do número. Após os ':', o '.2' significa que queremos 2 casas decimais enquanto 'f' significa que queremos um formato de ponto fixo. Não vou entrar nos detalhes de todas as formatações possíveis, mas podemos ver um pouco mais disso aqui.

Podemos também usar mais de um número dentro do mesmo método format. Por exemplo,

```
print('O valor de pi arredondado para 2 casas decimais é {:.2f}. Com 4 casas decimais, no en
```

### 4.1.4 Introdução a expressões regulares

Por vezes queremos encontrar um padrão específico de texto (e.g., placas de carro, e-mails ou números de telefone) dentro de um texto maior, para realizar algum tipo de coleta, limpeza ou mesmo substituição que um simples `str.replace()` não dá conta. Para realizar tal ação podemos utilizar as famosas **Expressões Regulares**, também conhecidas como *Regular Expressions* no inglês, ou simplesmente *Regex*.

As expressões regulares são em essência uma potente linguagem para especificar padrões de texto. De forma mais detalhada, é uma composição dos chamados **metacaracteres**, caracteres com funções especiais, que, agrupados entre si e em conjunto com caracteres literais, formam uma sequência, uma expressão. Essa expressão é interpretada como uma regra que indicará sucesso se uma entrada de dados qualquer casar com essa regra, ou seja, obedecer exatamente a todas as suas condições.

Imagine que você tenha o string abaixo, que mostra o texto de um trecho de uma notícia da CNN sobre o resultado da Pesquisa Datafolha para presidente divulgada em 18/08/2022 (link para a matéria completa aqui).

```
pesquisa = """
Pesquisa Datafolha divulgada nesta quinta-feira (18) mostra o ex-presidente Luiz Inácio Lula
com 47% das intenções de voto na corrida pelo Palácio do Planalto. O presidente Jair Bolsona
O primeiro turno das eleições acontece em 2 de outubro.

Na sequência, aparecem Ciro Gomes (PDT), com 7%; Simone Tebet (MDB), com 2%, e Vera Lúcia (P
"""

print(pesquisa)
```

E se quiséssemos, por exemplo, substituir todas as porcentagens de intenção de voto por 'ZZZ'? Note que as porcentagens são diferentes e não há uma repetição dos números que nos permita usar o `str.replace()` de uma vez só. No entanto, todas as porcentagens são representadas por 1 ou 2 números inteiros seguidos do símbolo %. Nesse caso, o mais indicado é utilizar as expressões regulares.

Os módulos e funções nativas do Python não nos trazem muito material para trabalhar com expressões regulares. Para operar com elas utilizaremos uma biblioteca de comandos chamada `re`. Para trazer para dentro do Python as funcionalidades dessa biblioteca precisamos utilizar a função `import` seguida do nome da biblioteca.

```
import re
```

Falaremos mais sobre importação de bibliotecas de comandos e funções mais a frente, mas por hora tenha na cabeça que para utilizar um conjunto de instruções disponível em alguma biblioteca importada é preciso utilizar o nome da biblioteca seguido de ponto e do nome da função dessa biblioteca que você quer utilizar. No caso, para realizar a substituição das porcentagens no string *pesquisa* utilizaremos a função `sub()` de dentro da biblioteca `re`. Mas o que devemos colocar como input dessa função?

O mundo das expressões regulares é um mundo gigante e à parte, com conteúdo suficiente para preencher um outro curso. De forma geral, a combinação entre metacaracteres e caracteres literais é o que dá o padrão do texto pelo qual procuramos. Alguns dos metacaracteres-padrão são `. ? * + ^ | [ ] { } ( ) \`, cada um realizando uma função específica. Para o nosso caso utilizaremos basicamente a expressão regular dada por

```
[0-9]{1,2}%
```

Mas o que essa coisa bizarra diz de fato? A função buscará todo e qualquer elemento dentro dos colchetes (no caso os dígitos numéricos) que apareça uma ou duas vezes (código dentro dos colchetes) e que seja seguido pelo símbolo de porcentagem. Note que esse é o padrão de qualquer uma das porcentagens no nosso string *pesquisa*. Vamos ver o que acontece se usarmos isso dentro de `re.sub()`.

```
pesquisa2 = re.sub('[0-9]{1,2}%','ZZZ',pesquisa)
```

```
print(pesquisa)
```

```
print(pesquisa2)
```

Conseguimos exatamente o que a gente queria. Boa, time!

Pare um pouco e pense sobre a aplicabilidade dessa ferramenta dentro de um mundo repleto de dados não-estruturados, como tweets e notícias. **O potencial de uso é gigante!** Mas como dissemos, o mundo de *regex* é muito grande e existem cursos e livros que se dedicam integralmente a estudar essa linguagem de padrões textuais. Uma boa referência introdutória é o livro Expressões Regulares: Uma Abordagem Divertida.

**Se eu pudesse dar um conselho para o meu eu de 15 anos atrás seria: beba água e estude expressões regulares.**

## 4.2 Listas

Como uma string, uma lista é uma sequência de valores. Em uma string, os valores são caracteres; em uma lista, eles podem ser de qualquer tipo. Podemos ter uma lista de strings, uma lista de valores numéricos ou mesmo uma lista de listas, combinando strings, números e mesmo outros tipos de objetos que ainda veremos, como tuplas, dicionários e dataframes.

Uma lista é delimitada por colchetes e os elementos, ou itens, pertencentes a ela são separados por vírgula. Para definir uma lista com 5 números inteiros, ordenados de forma sequencial e começando em 1 devemos escrever a seguinte linha de código:

```
lista1 = [1,2,3,4,5]

print(lista1)
print(type(lista1))
```

Podemos definir uma lista de strings, uma lista mista de strings e números, e uma lista composto por outras listas (lista aninhada):

```
lista2 = ['Danilo Souza','Claudio Lucinda']
lista3 = ['Turma 2024201',46.0,'Turmas 2024202',49,'Turmas 2024221',81]
lista4 = [lista1, lista2]

print(lista2)
print(lista3)
print(lista4)
```

De forma análoga à listas com elementos não vazios, é possível definir uma lista vazia utilizando apenas os colchetes:

```
lista_vazia = []

print(lista_vazia)
```

### 4.2.1 Operações básicas

A sintaxe para acessar os elementos de uma lista é a mesma que para acessar os caracteres de uma string: o operador de colchete. A expressão dentro dos colchetes especifica o índice ou o intervalo de índices. **Lembrando que o índice no Python começa em ZERO e não em UM**

```
print(lista2[0])
print(lista2[1])
print(lista3[-1])
print(lista3[0:2])
```

Diferente das strings, listas são mutáveis. Quando o operador de colchete aparece do lado esquerdo de uma atribuição, ele identifica o elemento da lista que será atribuído:

```
numbers = [42, 123]
numbers[1] = 5
numbers
```

O segundo elemento de numbers (índice 1), que era 123, agora é 5.

Índices de listas funcionam da mesma forma que os índices de strings:

- Qualquer expressão de números inteiros pode ser usada como índice.
- Se tentar ler ou escrever um elemento que não existe, você recebe um `IndexError`.
- Se um índice tiver um valor negativo, ele conta de trás para a frente, a partir do final da lista.

O operador `in`, que serve ao propósito de testar a existência de determinado elemento dentro de um objeto específico, também funciona com listas:

```
cheeses = ['Cheddar', 'Gorgonzola', 'Gouda']
'Edam' in cheeses
```

```
'Brie' in cheeses
```

Assim como com strings, é possível calcular o número de elementos em uma lista utilizando a função integrada `len()` e "concatenar" ou "somar" listas utilizando apenas o sinal de +

```
print(len(cheeses))
```

```
lista_soma = lista2 + cheeses
print(lista_soma)
```

### 4.2.2 Métodos de listas

As listas também possuem métodos bastante úteis, que nos facilitam a vida em várias dimensões. Dentre os principais métodos aplicáveis a listas e suas funcionalidades podemos citar:

1. **lista_example.append()**: adiciona um novo elemento ao fim da lista "lista_example".
2. **lista_example.extend(lista2)**: toma a lista "lista_example" como argumento e adiciona todos os elementos de lista2 como novos elementos da lista inicial.
3. **lista_example.sort()**: classifica os elementos de "lista_example" em ordem ascendente.
4. **lista_example.remove(x)**: exclui o elemento igual a "x" de "lista_example". É um método bastante útil quando queremos excluir um elemento específico, mas não sabemos sua posição dentro da lista.

A maior parte dos métodos de listas são nulos; eles alteram a lista e retornam None. Se você escrever `t = t.sort()` por acidente, ficará desapontado com o resultado.

```
t = ['a', 'b', 'c']
t.append('d')
print(t)
```

```
t1 = ['a', 'b', 'c']
t2 = ['d', 'e']
t1.extend(t2)
print(t1)
```

```
t = ['d', 'c', 'e', 'b', 'a']
t.sort()
print(t)
```

Assim como já explicamos, uma lista é uma sequência de valores e uma string é uma sequência de caracteres, mas uma lista de caracteres não é a mesma coisa que uma string. Para converter uma string em uma lista de caracteres, você pode usar o comando `list`:

```
s = 'spam'
t = list(s)
print(t)
```

A função `list` quebra uma string em letras individuais. Se você quiser quebrar uma string em palavras, você pode usar o método `split()`. Esse método admite um argumento adicional, chamado *delimiter*, que especifica quais caracteres podem ser usados para demonstrar os limites

das palavras. Isso é muito útil, por exemplo, quando queremos separar um texto em palavras e fazer a contagem de palavras que mais se repetem. Qual caracter deveríamos passar como argumento em `split()` nesse caso?

```
s = 'pining for the fjords'
t = s.split(' ')
print(t)
```

```
s = 'spam-spam-spam'
t = s.split('-')
t
```

O método `join()` é o contrário de `split()`. Ele toma uma lista de strings e concatena os elementos. `join()`, no entanto, é um método de string, então é preciso invocá-lo no string delimitador (por exemplo, " - ") e passar a lista de strings como parâmetro:

```
t = ['spam','spam','spam']
s = ' - '.join(t)
print(s)
```

Além desses principais, existem outros vários métodos para listas. Esse link é um bom ponto de partida para quem quiser conhecer outros exemplos.

### 4.2.3 Operadores lógicos e variáveis booleanas

Agora faremos um pequeno desvio, que será bem útil daqui para frente, para falar de expressões e variáveis **booleanas**. Em resumo, uma expressão booleana é uma expressão que pode ser verdadeira ou falsa e dessa forma assumir apenas dois valores como resultado: **True** e **False**. Os exemplos seguintes usam o operador de igual no Python (==) que compara dois operandos e produz True se forem iguais e False se não forem:

```
5 == 5
```

```
5 == 6
```

```
lista1 = [1,2,3,4,5]
lista2 = [5,4,3,2,1]

lista1 == lista2
```

*True* e *False* são valores especiais que pertencem ao tipo *bool*; não são strings. Além disso, é possível fazer operações aritméticas com variáveis do tipo *bool* já que o Python entende *True* como sendo equivalente ao número 1 e *False* como sendo equivalente ao número 0.

```
print(type(True))
print(type(False))
```

```
print(True + True)
print(True + False)
print(False + False)
```

O operador == é um dos operadores relacionais dentro do Python, os outros são:

1. **x != y**: x não é igual a y
2. **x > y**: x é maior que y
3. **x < y**: x é menor que y
4. **x >= y**: x é maior ou igual a y
5. **x <= y**: x é menor ou igual a y

Embora essas operações provavelmente sejam familiares para você, os símbolos do Python são diferentes dos símbolos matemáticos. Um erro comum é usar apenas um sinal de igual (=) em vez de um sinal duplo (==). Lembre-se de que = é um operador de atribuição e == é um operador relacional. Não existe => ou =<.

Há três operadores lógicos: `and`, `or` e `not`. A semântica (significado) destes operadores é semelhante ao seu significado em inglês. Por exemplo, `x>=0 and x<=10` só é verdade se x for maior que 0 **e** menor que 10. `n%2 == 0 or n%3 == 0` é verdadeiro se uma ou as duas condição(ões) for(em) verdadeira(s), isto é, se o número n for divisível por 2 ou 3 (caso você não esteja familiarizado com a divisão pelo piso e o operador módulo, tente brincar um pouco com `//` e com `%`). Finalmente, o operador `not` nega uma expressão booleana, então `not (x > y)` é verdade se `x > y` for falso, isto é, se x for menor que ou igual a y.

Falando estritamente, os operandos dos operadores lógicos devem ser expressões booleanas, mas o Python não é muito estrito. Qualquer número que não seja zero é interpretado como *True*:

```
42 and True
```

Esta flexibilidade tem sua utilidade, mas há algumas sutilezas relativas a ela que podem ser confusas. Assim, pode ser uma boa ideia evitá-la (a menos que você tenha certeza absoluta do que está fazendo).

### 4.2.4 Execução condicional

Para escrever programas úteis, quase sempre precisamos da capacidade de verificar condições e mudar o comportamento do programa de acordo com elas. Instruções condicionais nos dão esta capacidade. A forma mais simples é a instrução `if`:

```
x=5

if x > 0:
    print('x é positivo')
```

A expressão booleana depois do `if` é chamada de condição. Se for verdadeira, a instrução indentada é executada. Se não, nada acontece. *Aqui vale mais um adendo*: uma característica muito importante da sintaxe do Python é justamente a **identação**. Diferentemente de outras linguagens de programação, a identação exerce papel importante aqui, já que é através dela que se determina onde se inicia e onde termina um bloco de código, que pode ser uma expressão condicional ou mesmo uma função, como veremos mais a frente. Além de economizar várias chaves ("{" e "}") e vários "end", a identação exerce um papel estético importante ao permitir uma melhor visualização do código como um todo.

Não há limite para o número de instruções que podem aparecer no corpo de uma instrução if, mas deve haver pelo menos uma. Ocasionalmente, é útil ter um corpo sem instruções (normalmente como um espaço reservado para código que ainda não foi escrito). Neste caso, você pode usar a instrução pass, que não faz nada.

```
if x < 0:
    pass
```

### 4.2.5 Execução alternativa

Uma segunda forma da instrução if é a "execução alternativa", na qual há duas possibilidades e a condição determina qual será executada. A sintaxe pode ser algo assim:

```
x = 5

if x % 2 == 0:
    print('x é par')
else:
    print('x é ímpar')
```

Se o resto quando x for dividido por 2 for 0, então sabemos que x é par e o programa exibe uma mensagem adequada. Se a condição for falsa, o segundo conjunto de instruções é executado. Como a condição deve ser verdadeira ou falsa, exatamente uma das alternativas será executada. As alternativas são chamadas de ramos (branches), porque são ramos no fluxo da execução.

Podemos usar a instrução if para testar o tamanho de uma lista também:

```
lista1 = [1,2,3,4,5]

if len(lista1) > 2:
    print('lista1 tem mais de dois elementos')
else:
    print('lista1 tem menos de dois elementos')
```

### 4.2.6 Condicionais encadeadas

Às vezes, há mais de duas possibilidades e precisamos de mais que dois ramos. Esta forma de expressar uma operação de computação é uma condicional encadeada:

```
x = 5
y = 6

if x < y:
    print('x é menor do que y')
elif x > y:
    print('x é maior do que y')
else:
    print('x e y são iguais')
```

`elif` é uma abreviatura de "else if". Novamente, exatamente um ramo será executado. Não há nenhum limite para o número de instruções elif. Se houver uma cláusula else, ela deve estar no fim, mas não é preciso haver uma. Cada condição é verificada em ordem. Se a primeira for falsa, a próxima é verificada, e assim por diante. Se uma delas for verdadeira, o ramo correspondente é executado e a instrução é encerrada. Mesmo se mais de uma condição for verdade, só o primeiro ramo verdadeiro é executado.

## 4.3 Tuplas

Agora falaremos de mais um tipo de objeto básico do Python, a tupla. Uma tupla é uma sequência de valores. Os valores podem ser de qualquer tipo, e podem ser indexados por

números inteiros, portanto, nesse sentido, as tuplas são muito parecidas com as listas. A diferença importante é que as tuplas são **imutáveis**, assim como os strings.

Em resumo, uma tupla é uma lista de valores separados por vírgulas e em geral delimitado por parênteses (lembre que listas são delimitadas por colchetes):

```
t = ('a', 'b', 'c', 'd', 'e')

print(type(t))
```

Um único valor entre parênteses não é uma tupla. Para criar uma tupla com um único elemento, é preciso incluir uma vírgula final após o elemento, com ou sem os parênteses.

```
t1 = ('a')
t2 = 'a',
t3 = ('a',)

print(type(t1))
print(type(t2))
print(type(t3))
```

Outra forma de criar uma tupla é com a função integrada `tuple`. Sem argumentos, cria uma tupla vazia. Se os argumentos forem uma sequência (string, lista ou tupla), o resultado é uma tupla com os elementos da sequência:

```
t = tuple()
print(t)
```

```
t = tuple('lupins')
print(t)
```

### 4.3.1 Operações básicas

A maior parte dos operadores de lista também funciona em tuplas. O operador de colchetes indexa um elemento e o operador de fatia seleciona vários elementos.

```
t = ('a', 'b', 'c', 'd', 'e')
print(t[0])
print(t[1:3])
```

Entretanto, se tentar alterar um dos elementos da tupla, vai receber um erro:

```
t[0] = 'A'
```

Como tuplas são imutáveis, você não pode alterar os elementos, mas pode substituir uma tupla por outra.

```
t = ('A',) + t[1:]
print(t)
```

Os operadores relacionais funcionam com tuplas e outras sequências. O Python começa comparando o primeiro elemento de cada sequência. Se forem iguais, vai para os próximos elementos, e assim por diante, até que encontre elementos que sejam diferentes. Os elementos subsequentes não são considerados (mesmo se forem muito grandes).

```
(0, 1, 2) < (0, 3, 4)
```

```
(0, 1, 2000000) < (0, 3, 4)
```

### 4.3.2 Atribuição de tuplas

Muitas vezes, é útil trocar os valores de duas variáveis. Com a atribuição convencional, é preciso usar uma variável temporária. Por exemplo, trocar a e b.

```
a=5
b=6

temp = a
a = b
b = temp

print(b)
```

Essa solução é trabalhosa; a atribuição de tuplas é mais elegante:

```
a=5
b=6

a, b = b, a

print(b)
```

O lado esquerdo é uma tupla de variáveis e o lado direito é uma tupla de expressões. Cada valor é atribuído à sua respectiva variável. Todas as expressões no lado direito são avaliadas antes de todas as atribuições.

O número de variáveis à esquerda e o número de valores à direita precisam ser iguais:

```
a, b = 1, 2, 3
```

De forma geral, o lado direito pode ter qualquer tipo de sequência (string, lista ou tupla). Por exemplo, para dividir um endereço de email em um nome de usuário e um domínio, você poderia escrever:

```
addr = 'monty@python.org'
uname, domain = addr.split('@')
```

O valor de retorno do `split()` é uma lista com dois elementos; o primeiro elemento é atribuído a `uname`, o segundo a `domain`:

```
print(uname)
print(domain)
```

### 4.3.3 Tuplas como valores de retorno

Falando estritamente, uma função só pode retornar um valor, mas se o valor for uma tupla, o efeito é o mesmo que retornar valores múltiplos. Por exemplo, se você quiser dividir dois números inteiros e calcular o quociente e resto, não é eficiente calcular `x/y` e depois `x%y`. É melhor calcular ambos ao mesmo tempo. A função integrada `divmod` toma dois argumentos e devolve uma tupla de dois valores: o quociente e o resto da divisão do primeiro termo pelo segundo termo. Você pode guardar o resultado como uma tupla:

```
print(7//3)
print(7%3)
```

```
t = divmod(7, 3)

print(t)
print(type(t))
```

### 4.3.4 Operações nativas com tuplas

`zip` é uma função integrada que recebe duas ou mais sequências e devolve uma lista de tuplas onde cada tupla contém um elemento de cada sequência. O nome da função tem a ver com o zíper, que se junta e encaixa duas carreiras de dentes.

Este exemplo encaixa uma string e uma lista:

```
s = 'abc'
t = [0, 1, 2]
zip(s,t)
```

O resultado é um objeto `zip` que sabe como percorrer os pares. O uso mais comum de `zip` é em um loop `for` (falaremos sobre loops mais adiante no curso):

```
for pair in zip(s, t):
    print(pair)
```

Um objeto `zip` é um tipo de iterador, ou seja, qualquer objeto que percorre ou itera sobre uma sequência. Iteradores são semelhantes a listas em alguns aspectos, mas, ao contrário de listas, não é possível usar um índice para selecionar um elemento de um iterador. Se quiser usar operadores e métodos de lista, você pode usar um objeto `zip` para fazer uma lista:

```
list(zip(s, t))
```

O resultado é uma lista de tuplas. Neste exemplo, cada tupla contém um caractere da string e o elemento correspondente da lista. Se as sequências não forem do mesmo comprimento, o resultado tem o comprimento da mais curta:

```
list(zip('Anne', 'Elk'))
```

## 4.4 Dicionários

Dicionários são um outro tipo de objeto bastante importante e útil em nossas aplicações. Dicionários são um dos melhores recursos do Python, eles são os blocos de montar de muitos algoritmos eficientes e elegantes.

Um dicionário se parece com uma lista, mas é mais geral. Em uma lista os índices têm que ser números inteiros, em um dicionário eles podem ser de (quase) qualquer tipo. Um dicionário contém uma coleção de índices, que se chamam **chaves** e uma coleção de valores. **Cada chave**

**é associada com um único valor**. A associação de uma chave e um valor chama-se par chave-valor ou item.

Em linguagem matemática, um dicionário representa um mapeamento de chaves a valores, para que você possa dizer que cada chave "mostra o mapa" a um valor. Como exemplo, vamos construir um dicionário que faz o mapa de palavras do inglês ao espanhol, portanto as chaves e os valores são todos strings.

A função `dict` cria um novo dicionário sem itens. Como `dict` é o nome de uma função integrada, você deve evitar usá-lo como nome de variável.

```
eng2sp = dict()
eng2sp
```

As chaves {} representam um dicionário vazio. Para acrescentar itens ao dicionário, você pode usar colchetes:

```
eng2sp['one'] = 'uno'
```

Esta linha cria um item que mapeia da chave 'one' ao valor 'uno'. Se imprimirmos o dicionário novamente, vemos um par chave-valor com dois pontos entre a chave e o valor:

```
eng2sp
```

Este formato de saída também é um formato de entrada. Por exemplo, você pode criar um dicionário com três itens:

```
eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
eng2sp
```

Ué, mas a ordem parece diferente daquela que definimos, não? Se você digitar o mesmo exemplo no seu computador, pode receber um resultado diferente. Em geral, a ordem dos itens em um dicionário é imprevisível. No entanto, isso não é um problema porque os elementos de um dicionário nunca são indexados com índices de números inteiros. Em vez disso, você usa as chaves para procurar os valores correspondentes. A chave `'two'`, por exemplo sempre mapeia ao valor `'dos'`, assim a ordem dos itens não importa.

```
eng2sp['two']
```

Se a chave não estiver no dicionário, você recebe uma exceção:

```
eng2sp['four']
```

Assim como em strings e listas, a função `len` também é compatível com dicionários. Nesse caso ela devolve o número de pares chave-valor.

```
len(eng2sp)
```

O operador `in` funciona em dicionários também. Ele acusa se algo aparece como chave no dicionário (aparecer como valor não é o suficiente).

```
'one' in eng2sp
```

```
'uno' in eng2sp
```

Para ver se algo aparece como um valor em um dicionário, você pode usar o método `values()`, que devolve uma coleção de valores, e então usar o operador `in`:

```
vals = eng2sp.values()
'uno' in vals
```

### 4.4.1 Operações básicas

Podemos adicionar um novo par de chave e valor ao dicionário usando colchetes, mas caso a chave já exista no dicionário, podemos atualizar o valor.

```
y = {}
y['one'] = 1
y['two'] = 2
print(y)
```

```
y['two'] = 'dos'
print(y)
```

O valor-chave deve ser um tipo de dados imutável, por essa razão, se você tentar definir um dicionário com um tipo de dados mutável, o Python levantará um erro de exceção.

A instrução `del` pode ser usada para remover uma entrada (par de valor de chave) de um dicionário.

```
y = {'one': 1, 'two': 2}
print(y)
```

```
del y['two']
print(y)
```

Mais uma vez, podemos utilizar a função nativa `len` para retornar o número de entradas (pares de valor de chave) em um dicionário.

```
x = {'one': 0, 'two': 2}
print(len(x))
```

Tente acessar uma chave que não está em um dicionário e você receberá um erro de exceção do Python. Para lidar com essa exceção, você pode usar mais uma vez o operador `in` que testa se existe uma chave em um dicionário. Esse operador retorna *True* se o dicionário tiver um valor armazenado sob a chave fornecida e *Falso* caso contrário.

```
y = {'one': 1, 'two': 2}
del y['three']
```

```
'three' in y
```

### 4.4.2 Métodos de dicionários

Dentre os principais métodos aplicáveis a dicionários e suas funcionalidades podemos citar:

1. **dict_example.update(x)**: atualiza o dicionário "dict_example" com todos os pares de valor-chave de um segundo dicionário "x". Os valores de chaves, que são comuns a ambos os dicionários, do segundo dicionário irão se sobrepor aos valores das chaves do primeiro dicionário.
2. **dict_example.keys(x)**: permite que você obtenha todas as chaves no dicionário. Muitas vezes é usado dentro de uma série de instruções repetida várias vezes para iterar sobre o conteúdo de um dicionário.
3. **dict_example.items()**: retorna todas as chaves do dicionário "dict_example" e seus valores associados como uma sequência de tuplas.
4. **dict_example.get(x,y)**: devolve o valor associado a uma chave "x" se o dicionário contiver essa chave. Caso o dicionário não contenha a chave, você pode especificar um segundo argumento opcional "y" para retornar um valor padrão (se o argumento não estiver incluído o método retornará *None*).

5. **dict_example.setdefault(x,y)**: é semelhante ao método `get()`: ele retorna o valor associado a uma chave "x" se o dicionário contiver essa chave, mas caso o dicionário não contenha a chave, este método criará um novo elemento no dicionário (par de valor de chave), onde o primeiro argumento neste método é a chave, e o segundo argumento ("y") é o valor. O segundo argumento é opcional, mas se isso não for incluído, o valor atrelado a essa nova chave será *None*.

Além desses principais, existem outros vários métodos para dicionários. Esse link é um bom ponto de partida para quem quiser conhecer outros exemplos.

```
x = {'one': 0, 'two': 2}
y = {'one': 1, 'three': 3}
print(x)

x.update(y)
print(x)
```

```
x = {'one': 1, 'two': 2}
print(x.keys())
```

```
x = {'one': 1, 'two': 2}
print(x.items())
```

```
y = {'one': 1, 'two': 2}
print(y.get('one'))
print(y.get('three'))
print(y.get('three', 'The key does not exist.'))
```

```
y = {'one': 1, 'two': 2}
print(y.setdefault('three', '3'))
print(y.setdefault('two', 'dos'))
print(y)
```

```
print(y.setdefault('four'))
print(y)
```

## 4.5 Exercícios

1. Considere a seguinte lista: `fruit = ['pear', 'orange', 'apple', 'grapefruit', 'apple', 'pear']`. Use uma função de lista para dizer o índice da primeira ocorrência de `apple`

2. Usando a lista do exercício anterior, use uma função de lista para retornar o número de veze que `apple` ocorre.
3. Existem duas listas abaixo. Escreva um programa que as converta em um dicionário em que o item de `keys` é a chave e o item de `values` é o valor

```
keys = ['Ten', 'Twenty', 'Thirty']
values = [10, 20, 30]
```

**Output esperado do último exercício**

```
{'Ten': 10, 'Twenty': 20, 'Thirty': 30}
```

# 5 Controle de fluxo e iteração

In summary, this book has no content whatsoever.

# 6 Funções

In summary, this book has no content whatsoever.

---

## 6.1 O que é uma função?

No contexto das linguagens de programação, uma função é uma sequência nomeada de instruções, que executa algum tipo de operação específica mas não necessariamente numérica, como é o caso das funções matemáticas. A ideia essencial por trás de uma função é a de juntar algumas tarefas comuns ou repetidas e criar uma função para que, em vez de escrever o mesmo código várias vezes, possamos chama-la pelo nome e reutilizar o conjunto de instruções nela contida sempre que necessário.

Caso o objetivo de dividir um programa em funções ainda não tenha ficado claro, saiba que:

- Criar uma nova função dá a oportunidade de nomear um grupo de instruções, o que deixa o seu programa mais fácil de ler e de depurar
- As funções podem tornar um programa menor, eliminando o código repetitivo. Depois, caso precise fazer alguma alteração, basta fazê-la em um lugar só.
- Dividir um programa longo em funções permite depurar as partes uma de cada vez e então reuni-las em um conjunto funcional.
- As funções bem projetadas muitas vezes são úteis para muitos programas. Uma vez escritas e depuradas, você pode reutilizar as funções em programas fora daquele para o qual elas foram originalmente construídas.

Existem várias funções nativas no Python (e.g., `type()` para encontrar o tipo de um objeto) e mesmo dentro de bibliotecas com as quais já trabalhamos um pouco (e.g., `numpy.random.uniform()` para sortear números aleatórios de acordo com uma distribuição uniforme). À partir de agora veremos como podemos criar nossas próprias funções dentro da linguagem!

No Python, a sintaxe de uma função é dada por:

```
def nome_da_funcao(argumentos):
    <instruções>
```

Assim como nos *loops*, para definir uma nova função no Python é preciso começar com uma palavra-chave, nesse caso `def`. O nome que daremos à função vem logo em seguida. É através desse nome, `nome_da_funcao`, que chamaremos essa função em outras partes do nosso código. Entre parênteses definimos os `argumentos` que a função recebe para realizar o conjunto de instruções em `<instruções>`.

Note, mais uma vez, que todo o bloco de código que estiver **identado** e abaixo da linha de cabeçalho da função fará parte da função. Assim como nos *loops*, a função termina quando passarmos a primeira linha de código não-identada.

## 6.2 Nossa primeira função

Vamos começar criando uma função simples, que tem por objetivo printar uma das frases mais conhecidas da história do cinema:

```
def frase_cinema():
    print('Que a força esteja com você!')
```

Note que nesse caso, o parênteses logo após o nome da função está vazio. Isso quer dizer que essa função não recebe argumentos, apenas printa a frase entre aspas sempre que for chamada. Para chamá-la, basta usar o nome `frase_cinema` seguido dos parênteses vazios:

```
frase_cinema()
```

## 6.3 Argumentos de uma função

Na função anterior, não tínhamos nenhum argumento. Ou seja, toda vez que você chamar a função `frase_cinema`, ela vai fazer a mesma coisa. Pode até ser que seja o seu objetivo fazer exatamente isso – é uma forma de economizar código.

O ponto é que a ideia de função é muito mais poderosa do que simplesmente uma "abreviação" de um monte de linhas de código. A abstração implícita no conceito de função é poderosa o suficiente para garantir que a função retorne coisas diferentes caso você altere algum **argumento**. Vamos falar disso a seguir.

### 6.3.1 *Keyword arguments* e *default values*

Vamos então criar uma função que tenha argumentos.

```
def my_func(name,place):
    print(f"Olá {name}! Você é de {place}?")

my_func("Emily","Paris")
```

O que acontece se você especificar o `place` primeiro e depois o `name`? Vamos descobrir.

```
my_func("Hawaii","Robert")
```

Meio bizarro, não? A razão é que aqui temos os chamados **argumentos posicionais**. Ou seja, a função vai assumir que o primeiro argumento é o `name` e o segundo é o `place` não importa o que tenhamos passado como argumento. Para lidar com isso, a gente pode atribuir um nome a cada um dos argumentos, ou **palavra-chave**, que aí vai ser a palavra-chave, e não a posição, que vai determinar qual o valor atribuído a cada argumento.

```
my_func(place="Hawaii",name="Robert")
```

Vimos aqui que a posição passou a ser irrelevante porque colocamos os nomes de cada um dos argumentos. E se quiséssemos dar mais flexibilidade ainda à função, só especificando um subconjunto dos argumentos? Para que isso funcione, nós precisamos especificar os **valores-padrão** dos argumentos caso eles não sejam fornecidos.

Aqui tem uma função que faz isso.

```
def total_calc(bill_amount,tip_perc=10):

    total = bill_amount*(1 + tip_perc/100)
    total = round(total,2)
    print(f"Please pay ${total}")
```

Nesse caso, o argumento `bill_amount` é obrigatório. Por outro lado, o argumento `tip_perc` vai assumir o valor de 10 toda vez que ele não for explicitamente fornecido na chamada da função.

### 6.3.2 Argumentos arbitrários

Vamos começar fazendo algumas perguntas: * E se não soubermos o número exato de argumentos de antemão? * Podemos criar funções que funcionem com um número variável de argumentos?

A resposta é *sim*! E vamos criar essa função imediatamente. Vamos criar uma função simples `my_var_sum()` que retorna a soma de todos os números passados como argumento. No entanto, o número de argumentos pode ser potencialmente diferente cada vez que chamamos a função.

```
def my_var_sum(*args):

    sum = 0
    for arg in args:
        sum += arg

    print(f"The numbers that you have add up to {sum}")
```

Observe como a definição da função agora tem `*args` em vez de apenas o nome do parâmetro. No corpo da função, fazemos um loop em `args` até usarmos todos os argumentos. A função `my_var_sum` retorna a soma de todos os números passados como argumentos. Olha só o que acontece quando chamamos a função com diferentes números de argumentos:

```
my_var_sum(99,10,54,23)
my_var_sum(9,87)
my_var_sum(5,21,36,79,45,65)
my_var_sum(1)
```

Mas e se eu quiser passar não apenas uma sequência de valores, mas uma sequência de valores *com nomes*? Não se preocupe, existe uma possibilidade de fazer isso, usando o `**kwargs` na definição dos argumentos. Qual é a diferença entre `*args` e `**kwargs`? A diferença é que você vai passar uma sequência de tamanho arbitrário de parâmetros, cada um deles nomeado.

O Python vai entender o `**kwargs` como um dicionário. Cada elemento passado é um par chave-valor, e dentro da função você tem que desempacotar o valor da chave. Vamos seguir alguns exemplos.

```
def myFun(**kwargs):

    for key, value in kwargs.items():
        print("%s == %s" % (key, value))


myFun(first='Geeks', mid='for', last='Geeks')
```

Podemos misturar os tipos de argumentos (posicionais versus `*args`/`*kwargs`)? Sim, como podemos ver no exemplo seguinte:

```
def myFun(arg1, **kwargs):

    for key, value in kwargs.items():
        print(arg1+" %s == %s" % (key, value))


myFun("Hi - ", first='Geeks', mid='for', last='Geeks')
```

### 6.3.3 Elementos dentro da função: execução condicional

As funções podem retornar objetos booleanos também, o que pode ser conveniente para esconder testes complicados dentro de funções. Por exemplo:

```
def is_divisible(x, y):

    if x % y == 0:
        print(True)
    else:
        print(False)

is_divisible(6, 4)
```

### 6.3.4 Aplicação: Teorema Central do Limite - Parte 1

O primeiro passo para replicarmos, na forma de uma função, a aplicação do Teorema Central do Limite da aula anterior é definir uma função que recebe como argumentos a distribuição da varíavel aleatória X, o intervalo no qual a variável está definida e o número de sorteios que faremos em cada amostra. Com o que aprendemos até agora podemos definir a função `func_tcl` abaixo:

```
def func_tcl(dist=None,intervalo=(0,1),n=100):

    if dist == None:
        print('Você esqueceu de carregar uma função que defina a distribuição de X! Volte 2 c
    else:
        x = dist(intervalo[0],intervalo[1],n)
        print(x)
```

Para todos os 3 argumentos definimos valores *default* que, caso não sejam alterados a função vai utilizá-los em suas operações. Vamos rodar a função com todos os valores *default* e ver o que acontece:

```
func_tcl()
```

Nesse caso, a execução condicional `if` dentro da função acende já que não passamos nenhuma distribuição como default, apenas o valor `None`. Vamos então repetir o que fizemos na aula passada e utilizar a distribuição uniforme, no intervalo $[-40, 40]$ e sorteando apenas 10 números desta distribuição:

```
func_tcl(dist=np.random.uniform,intervalo=(-40,40),n=10)
```

### 6.3.5 Elementos dentro da função: iteração

Da mesma forma que podemos ter condições lógicas dentro da função, podemos ter blocos de repetição (`for` e `while`) dentro da função, assim como no caso de `*args`. Vamos olhar para um outro exemplo em que a função recebe uma lista de strings como argumento e faz operações em cada um dos elementos, um a um:

```
states = [' Alabama ','Georgia!','Georgia','georgia','FlOrIda','south carolina##','West virgi

def clean_strings(lista_strings):
    result=[]

    for value in lista_strings:
        value = value.strip()
        value = value.title()
        value = value.replace('#','')
        value = value.replace('?','')
        value = value.replace('!','')
        result.append(value)

    print(result)

print(states)
print()
clean_strings(states)
```

### 6.3.6 Aplicação: Teorema Central do Limite - Parte 2

Agora podemos dar mais um passo e incorporar o loop dentro da função, passando como novo argumento da função o número de amostras com o qual queremos trabalhar:

```
def func_tcl(dist=None,intervalo=(0,1),n=100, samples=10):

    if dist == None:
        print('Você esqueceu de carregar uma função que defina a distribuição de X!')
    else:
        means = []
        for j in range(0,samples):
            x_func_tcl = dist(intervalo[0],intervalo[1],n)
            mean_x = sum(x_func_tcl)/len(x_func_tcl)

            means.append(mean_x)

        print('Essa é a lista de médias:')
        print(means)
        print('\nE essa é a média das médias:')
        print(sum(means)/len(means))
```

```
# utilizamos a função seed mais uma vez para tornar os resultados previsíveis
np.random.seed(1)

func_tcl(dist=np.random.uniform,intervalo=(-40,40),n=50,samples=10)
```

Que bacana!!!

Mas note que o valor da média que nos interessa nesse caso só aparece printado quando chamamos a função. O que podemos fazer para conseguir guardá-lo como uma nova variável, por exemplo?

## 6.4 Valor de retorno de uma função

O comando `return` é usado para a saída de uma função de volta ao lugar no código de onde ela foi chamada. A sintaxe desse comando é simples:

```
return <expression_list>
```

Essa declaração pode ter um comando que é executado e o valor resultante devolvido. Se esse comando não for anexado ou não tiver nada além do `return`, a função é devolvida com o valor de `None`. Por exemplo:

```
def my_func(name,place):

    print(f"Olá {name}! Você é de {place}?")
    return

print(my_func("Jane","Paris"))
```

Funções que não contenham o comando `return` sempre retornarão um valor vazio. Quando queremos utilizar uma função com o objetivo de realizar algum tipo de operação e **guardar** o valor dessa operação, não podemos esquecer do `return`. Vejamos o exemplo da função que calcula a raiz quadrada de um número:

```
def raiz(x):

    fx = x**0.5

y = raiz(100)
print(y)
```

```
def raiz(x):

    fx = x**0.5
    return fx

y = raiz(100)
print(y)
```

### 6.4.1 Expectativa de vida de variáveis dentro da função

Quando você cria uma variável **dentro** de uma função ela é local, ou seja, só existe dentro da própria função. Por exemplo:

```
def concat_strings(str1,str2):

    texto_concatenado = str1 + ' ' + str2
    print(texto_concatenado)
```

```
texto1 = 'Que a força esteja com você,'
texto2 = 'jovem Padawan.'

concat_strings(texto1,texto2)
```

Essa função recebe dois argumentos, concatena-os e exibe o resultado em uma única linha de texto. No entanto, assim que a função é encerrada, a variável `texto_concatenado` é deletada. O que acontece se tentarmos acessá-la?

```
print(texto_concatenado)
```

### 6.4.2 Aplicação: Teorema Central do Limite - Parte 3

Retomando a função da parte 2:

```
def func_tcl(dist=None,intervalo=(0,1),n=100, samples=10):

    if dist == None:
        print('Você esqueceu de carregar uma função que defina a distribuição de X!')
    else:
        means = []
        for j in range(0,samples):
            x_func_tcl = dist(intervalo[0],intervalo[1],n)
            mean_x = sum(x_func_tcl)/len(x_func_tcl)

            means.append(mean_x)
            mean_of_means = sum(means)/len(means)

        print('Essa é a lista de médias:')
        print(means)
        print('\nE essa é a média das médias:')
        print(mean_of_means)
```

```
np.random.seed(1)

func_tcl(dist=np.random.uniform,intervalo=(-40,40),n=50,samples=10)
```

Como acabamos de ver, se quisermos acessar a variável `mean_of_means` o Python retornará um erro. As variáveis definidas dentro de uma função tem vida curta: elas existem apenas dentro da função!

```
print(mean_of_means)
```

No entanto, agora já sabemos o que a palavra-chave `return` faz dentro de uma função. Vamos utilizá-la!

```
def func_tcl(dist=None,intervalo=(0,1),n=100, samples=10):

    if dist == None:
        print('Você esqueceu de carregar uma função que defina a distribuição de X!')
    else:
        means = []
        for j in range(0,samples):
            x_func_tcl = dist(intervalo[0],intervalo[1],n)
            mean_x = sum(x_func_tcl)/len(x_func_tcl)

            means.append(mean_x)
            mean_of_means = sum(means)/len(means)

        return mean_of_means
```

```
np.random.seed(1)

func_tcl(dist=np.random.uniform,intervalo=(-40,40),n=50,samples=10)
```

Maravilha, agora a função nos retorna apenas o que é do nosso interesse: a média das médias. Para chegar, finalmente, no mesmo resultado da aula passada basta colocar essa função dentro de um loop e tacar-lhe pau nesse carrinho, Marcos!

```
expoentes = [1,2,3,4,5,6,7,8,9,10]
Y = [2**exp for exp in expoentes]

means_of_means = []

np.random.seed(1)
for y in Y:

    mean_of_means = func_tcl(dist=np.random.uniform,intervalo=(-40,40),n=100,samples=y)
    means_of_means.append(mean_of_means)
```

```
import numpy as np
for m in means_of_means:

    print(np.round(m,2))
```

*Voilá*!

Como exercício para casa, tente fazer alterações nessa função de modo que ela receba não um valor com o número de amostras, mas uma lista de números de amostras. A função deve cuspir como resultado a lista `means_of_means` e não apenas o valor `mean_of_means`. Pratique!

## 6.5 Documentação

Uma **docstring** é uma string no início de uma função, definida pelo usuário, que serve como documentação do que a função faz. A docstring vem logo depois da primeira linha que define a função e é delimitada por aspas triplas, o que permite que a string se estenda por várias linhas, como vocês devem se lembrar.

Vamos criar como exemplo uma função que printa o tipo do objeto que é passado como argumento:

```
def imprime_tipo(x):

    '''
    Função criada para a matéria EAE1106 - Métodos Computacionais para Economia
    Objetivo: função simples que imprime o tipo do objeto recebido como argumento.
    '''

    print(type(x))
```

Embora opcional, a documentação é uma **boa prática de programação**. A menos que você consiga se lembrar qual foi o cardápio do bandejão na semana passada, sempre documente seu código. Podemos acessar a documentação de determinada função utilizando o atributo `__doc__`.

```
print(imprime_tipo.__doc__)
```

Isso vale também para funções nativas e definidas em outros pacotes do Python. Por exemplo,

```
# documentação da função nativa len()
print(len.__doc__)
```

```
import time

# documentação da função time() dentro do pacote time
print(time.time.__doc__)
```

```
import numpy as np

# documentação da função uniform() dentro do pacote NumPy
print(np.random.uniform.__doc__)
```

Note do exemplo acima que uma docstring pode conter uma descrição detalhada do funcionamento de uma função, inclusive com exemplos de aplicação. Legal, né? Esse tipo de documentação também está disponível para as bibliotecas como um todo. Isso pode nos ajudar, por exemplo, a conhecer o conteúdo de uma determinada biblioteca.

```
import numpy as np

# documentação do NumPy
print(np.__doc__)
```

Podemos usar sempre esse atalho caso desejemos conhecer as funcionalidades que uma biblioteca guarda por trás de suas cortinas!

## 6.6 Funções anônimas

Em Python, uma função anônima é uma função definida sem nome. Quem poderia imaginar, não é mesmo?

Enquanto as funções normais são definidas usando a palavra-chave `def` em Python, as funções anônimas são definidas usando a palavra-chave `lambda`. Portanto, funções anônimas também são chamadas de **funções lambda**. A estrutura usual de uma função lambda é a seguinte:

```
lambda <argumentos>: <expressão>
```

Usualmente a gente precisa de uma função lambda porque precisamos de uma função rápida por um período de tempo e/ou quando a gente usa técnicas mais poderosas que possuem funções como argumento, como `filter` e `map`. Vou fazer um exemplo com cada uma delas.

### Usando filter

A função `filter()` em Python recebe uma função e uma lista como argumentos. A função é chamada com todos os itens da lista e uma nova lista é retornada contendo itens para os quais a função avalia `True`. Aqui está um exemplo de uso da função para filtrar apenas números pares de uma lista.

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))

print(new_list)
```

### Usando map

A função `map()` em Python recebe uma função e uma lista. A função é chamada com todos os itens da lista e uma nova lista é retornada contendo os itens retornados por essa função para cada item. Aqui está um exemplo de uso da função `map()` para dobrar todos os itens em uma lista.

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))

print(new_list)
```

# Parte II

# Computação numérica, análise de dados e visualização

Este é o material da disciplina **EAE1106 — Métodos Computacionais para Economia**, ministrada no Departamento de Economia da USP.

O curso tem como objetivo introduzir lógica de programação e análise de dados utilizando principalmente **Python**, com uma breve introdução a **R** ao final do semestre.

# 7 Arrays, matrizes e álgebra linear

Uma das principais aplicações para o Python bla bla bla

---

## 7.1 O que é o NumPy?

NumPy (**Num**erical **Py**thon) é o pacote fundamental para computação científica em Python. Essa biblioteca é peça fundamental em outras bibliotecas igualmente importantes, como o Pandas e o Matplotlib. É uma biblioteca Python que tem como principal objeto o `ndarray`, um array multidimensional que guarda bastante semelhança com a ideia de vetores e matrizes, embora seja um objeto específico dentro da linguagem, com suas características e métodos próprios. O pacote contém também uma variedade de rotinas para operações rápidas em arrays, incluindo matemática, lógica, álgebra linear básica, operações estatísticas básicas e muito mais.

Mas o que é de fato um `ndarray`? É um objeto multidimensional que nos permite armazenar dados de forma sequencial e que podem ser acessados via indexação. Ué, mas isso é muito parecido com uma lista (ou um conjunto de listas). Qual a diferença então?

- NumPy arrays têm um tamanho fixo na criação, ao contrário das listas, que podem crescer. Alterar o tamanho de um ndarray criará um novo array e excluirá o original.
- Todos os elementos em um array devem ser do mesmo tipo de dados, diferentemente de listas, que são objetos mais genéricos. Isso facilita a gestão de memória e torna operações com esse tipo de objeto ordens de magnitude mais rápidas do que se utilizássemos listas.
- A maior velocidade e eficiência de armazenamento fazem do NumPy uma das bibliotecas mais utilizadas em aplicações matemáticas e científicas. Saber apenas as ferramentas nativas do Python, como listas, hoje já não é mais suficiente.

São muitas as qualidades do `NumPy` que fazem dele a melhor escolha quanto o assunto é lidar com objetos sequenciais, multidimensionais, e com os quais queremos operar tal qual vetores e matrizes. Mas chega de lenga lenga, vamos ao trabalho!

---

## 7.2 Elementos básicos do NumPy

Antes de tudo, é preciso importar o `NumPy`, já que se trata de uma biblioteca não nativa do Python.

```
import numpy as np
```

### 7.2.1 Arrays unidimensionais

Comecemos criando um `numpy.ndarray` do zero, contendo os números 1, 2 e 3. Podemos fazê-lo da seguinte forma:

```
a = np.array([1, 2, 3])
print(a)
print(type(a))
```

```
[1 2 3]
<class 'numpy.ndarray'>
```

A sintaxe é essa mesma: um par de colchetes dentro dos parênteses. Se tentarmos passar sem os colchetes, o Python retornará um erro.

```
a = np.array(1, 2, 3)
```

```
TypeError: array() takes from 1 to 2 positional arguments but 3 were given
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[3], line 1
----> 1 a = np.array(1, 2, 3)
TypeError: array() takes from 1 to 2 positional arguments but 3 were given
```

Da mesma forma que uma sequência de 3 números inteiros, podemos criar um `numpy.ndarray` que repete 3 vezes o número 0 ou 3 vezes o número 1.

```
a = np.zeros(3)
b = np.ones(3)

print(a)
print(b)
```

```
[0. 0. 0.]
[1. 1. 1.]
```

Note que em ambos os casos os números aparecem com o ponto da casa decimal, o que é um indicativo de que estão armazenados como valores do tipo `float`. E se quiséssemos criar esses mesmos arrays, mas especificando que os números são inteiros, i.e., do tipo `int`?

```
a = np.zeros(3, dtype=int)
b = np.ones(3, dtype=int)

print(a)
print(b)
```

```
[0 0 0]
[1 1 1]
```

A função `numpy.linspace(x,y,z)` nos permite criar um array que vai de `x` até `y`, com `z` elementos igualmente espaçados.

```
a = np.linspace(0,8,5, dtype=int)

print(a)
```

```
[0 2 4 6 8]
```

Podemos acessar os elementos de um array qualquer utilizando a mesma ideia de indexação de listas:

```
print('Array a =',a)
print('\nPrimeiro elemento de a = ',a[0])
print('Segundo elemento de a = ',a[1])
print('Último elemento de a = ',a[-1])
print('Dois primeiros elementos de a = ',a[0:2])
```

```
Array a = [0 2 4 6 8]

Primeiro elemento de a =  0
Segundo elemento de a =  2
Último elemento de a =  8
Dois primeiros elementos de a =  [0 2]
```

### 7.2.2 Arrays multidimensionais

Como falamos anteriormente, um objeto do tipo `numpy.ndarray` é um array *n-dimensional* (por isso o `nd` em `ndarray`). Até agora trabalhamos apenas com uma dimensão, mas para as nossas aplicações é particularmente interessante o caso em que o número de dimensões é igual a 2, i.e., para o caso em que o array assume a forma de uma **matriz**.

Podemos criar esse tipo de array usando a mesma lógica de antes, com pequenas alterações na sintaxe:

```
A_22 = np.array([[1, 2], [3, 4]], dtype=int)

print('Matriz A =\n',A_22)
```

```
Matriz A =
 [[1 2]
 [3 4]]
```

O NumPy nos oferece algumas funções interessantes para criarmos matrizes específicas:

```
print('Matriz identidade 2x2 =\n',np.eye(2,dtype=int))
print('\nMatriz diagonal 3x3 =\n',np.diag([1,2,3]))
```

```
Matriz identidade 2x2 =
 [[1 0]
 [0 1]]

Matriz diagonal 3x3 =
 [[1 0 0]
 [0 2 0]
 [0 0 3]]
```

Podemos criar a matriz **transposta** de uma dada matriz utilizando a função `np.transpose()` (ou apenas o método `.T`):

```
print('Matriz A =\n',A_22)
print('\nTransposta da matriz A =\n',np.transpose(A_22))
print('\nTransposta da matriz A =\n',A_22.T)
```

```
Matriz A =
 [[1 2]
 [3 4]]

Transposta da matriz A =
 [[1 3]
 [2 4]]

Transposta da matriz A =
 [[1 3]
 [2 4]]
```

Assim como em arrays unidimensionais, podemos acessar os elementos de uma matriz utilizando indexação, mas agora em 2 dimensões:

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

print('Matriz A =\n',A)
print('\nElemento 11 de A = ',A[0,0])
print('Elemento 23 de A = ',A[1,2])
print('Primeira linha de A = ',A[0,:])
print('Segunda coluna de A = ',A[:,1])
print('\nSubmatriz de A delimitada pelos elementos 22 e 33 =\n',A[1:,1:])
```

```
Matriz A =
 [[1 2 3]
 [4 5 6]
 [7 8 9]]

Elemento 11 de A =  1
Elemento 23 de A =  6
Primeira linha de A =  [1 2 3]
Segunda coluna de A =  [2 5 8]

Submatriz de A delimitada pelos elementos 22 e 33 =
 [[5 6]
 [8 9]]
```

### 7.2.3 Propriedades de arrays

O NumPy fornece alguns métodos úteis que, apesar de não receberem nenhum argumento, nos permitem acessar algumas das características dos arrays que criamos.

- `ndim` retorna o número de dimensões do array;
- `shape` retorna o tamanho do array em cada uma de suas dimensões;
- `dtype` retorna o tipo de dado contido no array;
- `size` retorna o número total de elementos contidos no array.

```
X1 = np.array([[1,2,3], [4,5,6], [7,8,9]])
X2 = X1.flatten() # O método flatten reduz um array de n-dimensões em um array de uma única d

print('Array X1 =\n',X1)
print('\nDimensões de X1 = ', X1.ndim)
print('Shape de X1 = ', X1.shape)
print('Tipo de dado em X1 = ', X1.dtype)
print('Número de elementos em X1 = ', X1.size)

print('\n\nArray X2 =\n',X2)
print('\nDimensões de X2 = ', X2.ndim)
print('Shape de X2 = ', X2.shape)
print('Tipo de dado em X2 = ', X2.dtype)
print('Número de elementos em X2 = ', X2.size)
```

```
Array X1 =
 [[1 2 3]
 [4 5 6]
 [7 8 9]]

Dimensões de X1 =  2
Shape de X1 =  (3, 3)
Tipo de dado em X1 =  int64
Número de elementos em X1 =  9


Array X2 =
 [1 2 3 4 5 6 7 8 9]

Dimensões de X2 =  1
Shape de X2 =  (9,)
Tipo de dado em X2 =  int64
Número de elementos em X2 =  9
```

Um método interessante de arrays é o `reshape(x,y)` que reorganiza um array existente de acordo com os argumentos `x` e `y`:

```
X = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16]])

print('Array X1 4x4 =\n',X)
print('\n X1 reorganizado em 2x8 =\n',X.reshape(2,8))
print('\n X1 reorganizado em 8x2 =\n',X.reshape(8,2))
```

```
Array X1 4x4 =
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]

 X1 reorganizado em 2x8 =
 [[ 1  2  3  4  5  6  7  8]
 [ 9 10 11 12 13 14 15 16]]

 X1 reorganizado em 8x2 =
 [[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]
 [13 14]
 [15 16]]
```

## 7.3 Operações básicas com arrays

### 7.3.1 Operações aritméticas simples

Dois tipos de operações que serão úteis para arrays de qualquer dimensão são:

1. Operações entre um array e um único número.
2. Operações entre dois arrays da mesma forma.

Quando realizamos operações em um array usando um único número, simplesmente aplicamos essa operação a cada elemento do array. Isso vale tanto para arrays unidimensionais (vetores) quanto multidimensionais (por ex., matrizes).

```
x = np.array([1,2,3], dtype=int)
print("x =\n", x)
print("\n2 + x =\n", 2 + x)
print("\n2 - x =\n", 2 - x)
print("\n2 * x =\n", 2 * x)
print("\nx / 2 =\n", x / 2)
```

```
x =
 [1 2 3]

2 + x =
 [3 4 5]

2 - x =
 [ 1  0 -1]

2 * x =
 [2 4 6]

x / 2 =
 [0.5 1.  1.5]
```

```
X = np.ones((2, 2), dtype=int)

print("X =\n", X)
print("\n2 + X =\n", 2 + X)
print("\n2 - X =\n", 2 - X)
print("\n2 * X =\n", 2 * X)
print("\nX / 2 =\n", X / 2)
```

```
X =
 [[1 1]
 [1 1]]

2 + X =
 [[3 3]
 [3 3]]

2 - X =
 [[1 1]
 [1 1]]
```

```
2 * X =
 [[2 2]
 [2 2]]

X / 2 =
 [[0.5 0.5]
 [0.5 0.5]]
```

Para operações entre dois arrays de mesmo tamanho, basta aplicar a operação elemento a elemento (*elementwise*, em inglês) entre os arrays.

```
x = np.array([2, 4, 6], dtype=int)
y = np.array([2, 2, 1], dtype=int)

print("x =\n", x)
print("\ny =\n", y)
print("\nx + y =\n", x + y)
print("\nx - y =\n", x - y)
```

```
x =
 [2 4 6]

y =
 [2 2 1]

x + y =
 [4 6 7]

x - y =
 [0 2 5]
```

```
X = np.array([[2, 4], [6, 8]], dtype=int)
Y = np.array([[2, 2], [2, 2]], dtype=int)

print("X =\n", X)
print("\nY =\n", Y)
print("\nX + Y =\n", X + Y)
print("\nX - Y =\n", X - Y)
```

```
X =
```

```
 [[2 4]
 [6 8]]

Y =
 [[2 2]
 [2 2]]

X + Y =
 [[ 4  6]
 [ 8 10]]

X - Y =
 [[0 2]
 [4 6]]
```

As operações de multiplicação e divisão entre arrays são um pouco diferentes. São duas as possibilidades:

- Multiplicação e divisão elemento a elemento.
- Multiplicação entre arrays usando a lógica de matriz e "divisão" usando a lógica de matriz inversa.

Para realizar as operações *elementwise* basta utilizar os sinais usuais de `*` e `/`, independentemente do número de dimesões do array.

```
print('Vetores x e y:')
print('x =',x)
print('y =',y)
print("\n x * y =\n", x * y)
print("\n x / y =\n", x / y)

print('\nMatrizes X e Y:')
print('X =\n',X)
print('Y =\n',Y)
print("\n X * Y =\n", X * Y)
print("\n X / Y =\n", X / Y)
```

```
Vetores x e y:
x = [2 4 6]
y = [2 2 1]

 x * y =
```

```
 [4 8 6]

 x / y =
 [1. 2. 6.]

Matrizes X e Y:
X =
 [[2 4]
 [6 8]]
Y =
 [[2 2]
 [2 2]]

 X * Y =
 [[ 4  8]
 [12 16]]

 X / Y =
 [[1. 2.]
 [3. 4.]]
```

A **multiplicação** de matriz com matriz, do jeito que a gente conhece do Ensino Médio é feita usando o símbolo `@` (ou através da função `np.dot()`):

```
X= np.array([[1, 2], [3, 4]], dtype=int)
Y= np.array([[10, 20], [30, 40]], dtype=int)

print('X =\n',X)
print('Y =\n',Y)
print('\nX * Y =\n', X @ Y)
print('\nX * Y =\n', np.dot(X,Y))
```

```
X =
 [[1 2]
 [3 4]]
Y =
 [[10 20]
 [30 40]]

X * Y =
 [[ 70 100]
 [150 220]]
```

```
X * Y =
 [[ 70 100]
 [150 220]]
```

Para calcular a inversa de uma matriz utilizando o NumPy é preciso um pouco mais de estrutura e de conhecimento acerca do subpacote `linalg` que nos traz funções para realizar operações de álgebra linear. Falaremos disso já já!

### 7.3.2 Funções universais

As funções universais no Numpy são funções matemáticas simples. É apenas um termo que demos às funções matemáticas na biblioteca Numpy, que cobrem uma ampla variedade de operações. Essas funções incluem funções trigonométricas padrão, funções para operações aritméticas, manipulação de números complexos, funções estatísticas, etc. Essas funções possuem como características principais:

- Elas executam operações de array elemento por elemento.
- Elas suportam vários recursos, como conversão de tipos.
- As funções universais são objetos que pertencem à classe `numpy.ufunc`.
- As funções do Python também podem ser criadas como uma função universal usando a função da biblioteca `frompyfunc`.
- Algumas funções universais são chamadas automaticamente quando o operador aritmético correspondente é usado em arrays. Por exemplo, quando a adição de dois arrays é executada em elementos usando o operador '+', então `np.add()` é chamado internamente.

Dentre as principais funções universais matemáticas estão:

- `sin`, `cos`, `tan`: calcular seno, cosseno e tangente de ângulos.
- `hypot`: calcule a hipotenusa do triângulo retângulo dado.
- `arcsinh`, `arcosh`, `arctanh`: calcular seno hiperbólico inverso, cosseno e tangente.
- `deg2rad`: converter grau em radianos.
- `rad2deg`: converter radianos em graus.

Dentre as principais funções universais estatísticas estão:

- `amin`, `amax`: retorna o mínimo ou máximo de um array ou ao longo de um eixo.
- `ptp`: retorna o intervalo de valores (máximo-mínimo) de um array ou ao longo de um eixo.
- `sum`: retorna a soma de valores de um array ao longo de um eixo.
- `percentile(a, p, eixo)`: calcular o p-ésimo percentil da matriz ou ao longo do eixo especificado.
- `median`: calcular a mediana dos dados ao longo do eixo especificado.

- `mean`: calcular a média dos dados ao longo do eixo especificado.
- `var`: calcular a variância de dados ao longo do eixo especificado.
- `log`: calcular o log dos dados ao longo do eixo especificado.

Alguns exemplos:

```
angulos_notaveis_deg = np.array([30,45,60])
angulos_notaveis_rad = np.deg2rad(angulos_notaveis_deg)

seno_notaveis = [np.round(elem,2) for elem in np.sin(angulos_notaveis_rad)]
cosseno_notaveis = [np.round(elem,2) for elem in np.cos(angulos_notaveis_rad)]
tangente_notaveis = [np.round(elem,2) for elem in np.tan(angulos_notaveis_rad)]

print('Seno, cosseno e tangente de 30 graus: ',seno_notaveis[0],', ',cosseno_notaveis[0],' e
print('Seno, cosseno e tangente de 45 graus: ',seno_notaveis[1],', ',cosseno_notaveis[1],' e
print('Seno, cosseno e tangente de 60 graus: ',seno_notaveis[2],', ',cosseno_notaveis[2],' e
```

```
Seno, cosseno e tangente de 30 graus:  0.5 ,  0.87  e  0.58
Seno, cosseno e tangente de 45 graus:  0.71 ,  0.71  e  1.0
Seno, cosseno e tangente de 60 graus:  0.87 ,  0.5  e  1.73
```

```
x = np.array([1,2,3,4,5])

print('Array x = ',x)
print('\nMínimo de x = ',np.amin(x))
print('Máximo de x = ',np.amax(x))
print('Intervalo de x = ',np.ptp(x))
print('Soma de x = ',np.sum(x))
print('Média de x = ',np.mean(x))
print('Log de x = ',[np.round(elem,2) for elem in np.log(x)])
```

```
Array x =  [1 2 3 4 5]

Mínimo de x =  1
Máximo de x =  5
Intervalo de x =  4
Soma de x =  15
Média de x =  3.0
Log de x =  [np.float64(0.0), np.float64(0.69), np.float64(1.1), np.float64(1.39), np.float6
```

### 7.3.3 Arrays e listas

Agora que já vimos um pouco de operações básicas, podemos entender um pouco melhor a diferença de desempenho entre arrays e listas. Considere um array de dez milhões de números inteiros e uma lista equivalente:

```
my_array = np.arange(10000000)
my_list = list(range(10000000))
```

Agora vamos multiplicar, elemento a elemento, todos os números por 2 e salvar o resultado correspondente. Utilizaremos a função `time` do pacote `time`para fazer a medição do tempo utilizado pelos dois métodos.

```
import time

# arrays
start_array = time.time()
my_array2 = my_array * 2
end_array   = time.time()


# listas
start_lista = time.time()
my_list2 = [x * 2 for x in my_list]
end_lista   = time.time()

ratio = (end_lista - start_lista) / (end_array - start_array)
```

Qual abordagem será que levou menos tempo?

```
print('Tempo necessário para a realização dos cálculos utilizando arrays: {:.4f} segundos'.fo
print('Tempo necessário para a realização dos cálculos utilizando listas: {:.4f} segundos'.fo
print('\nA abordagem de listas demorou {:.0f}x mais tempo! Esqueça listas e use arrays ;)'.fo
```

```
Tempo necessário para a realização dos cálculos utilizando arrays: 0.0129 segundos
Tempo necessário para a realização dos cálculos utilizando listas: 0.2792 segundos

A abordagem de listas demorou 21x mais tempo! Esqueça listas e use arrays ;)
```

## 7.4 Aplicação: solução de sistema de equações lineares

Chega de exemplos vazios, vamos usar o NumPy para resolver um problema concreto e que nos é muito familiar: a solução de sistemas de equações lineares. Considere o sistema linear de equações dado por

$$A \cdot \vec{x} = \vec{b}$$

tal que $A$ é a matriz de coeficientes, $\vec{x}$ é o vetor de incógnitas e $\vec{b}$ o vetor de constantes.

### 7.4.1 Algoritmo de eliminação de Gauss-Jordan

Segundo Hubbard e Hubbard (2015), uma matriz de coeficientes $A$ é representada em sua forma escalonada reduzida por linhas (*reduced row echelon form*, em inglês) se

- Em toda e qualquer linha, a primeira entrada diferente de zero é igual a 1 (1 *pivotal*).
- O 1 *pivotal* de uma linha mais abaixo está sempre à direita de um outro 1 *pivotal* de alguma linha acima.
- Em toda e qualquer coluna que contém um 1 *pivotal*, todas as outras entradas são iguais a zero.
- Toda linha contendo apenas zeros está no fim da matriz.

À partir dessa definição, é possível mostrar que para qualquer matriz $A$, existe uma matriz $\tilde{A}$ na forma escalonada reduzida por linhas que pode ser obtida à partir de operações elementares nas linhas de A. Além disso, é possível mostrar também que $\tilde{A}$ é única. Ao algoritmo utilizado para encontrar $\tilde{A}$ é dado o nome de **Algoritmo de Eliminação de Gauss-Jordan**.

ALGORITMO DE ELIMINAÇÃO DE GAUSS-JORDAN

Para levar uma matriz $A$ a sua forma escalonada reduzida por linhas $\tilde{A}$ devemos seguir os seguintes passos:

1. Encontre a primeira coluna que não é composta apenas de zeros, chame isso de primeira coluna pivotal e chame sua primeira entrada diferente de zero de pivô. Se o pivô não for na primeira linha, mova a linha que a contém para o topo da matriz.

2. Divida a primeira linha inteira pelo pivô, de modo que a primeira entrada da primeira coluna pivotal seja igual a 1.

3. Adicione múltiplos apropriados da primeira linha às outras linhas para garantir que todas as outras entradas da primeira coluna pivotal sejam iguais a 0. O 1 na primeira coluna é agora um pivô 1.

4. Escolha a próxima coluna que contém pelo menos uma entrada diferente de zero abaixo da primeira linha e coloque a linha que contém o novo pivô na posição da segunda linha. Faça do pivô um pivô 1: divida a linha inteira pelo pivô e adicione múltiplos apropriados desta linha às outras linhas abaixo, para tornar todas as outras entradas desta coluna iguais a 0.

5. Repita o processo até que a matriz esteja em sua forma escalonada reduzida por linhas.

Assuma o caso em que a matriz de coeficiente $A$ e o vetor de constantes $b$ são tais que a matriz ampliada é dada por:

$$\left[\begin{array}{ccc|c} 2 & 2 & 1 & 1 \\ 1 & 3 & 1 & 2 \\ 1 & 2 & 2 & -1 \end{array}\right]$$

- Passo 1: Defina a matriz M

```
M = np.array([(2, 2, 1, 1),
              (1, 3, 1, 2),
              (1, 2, 2, -1)],dtype=float)
print('Matriz ampliada =\n',M)
```

```
Matriz ampliada =
 [[ 2.  2.  1.  1.]
 [ 1.  3.  1.  2.]
 [ 1.  2.  2. -1.]]
```

- Passo 2: Divida a linha 1 pelo primeiro elemento da primeira linha

```
M[0,:] = M[0,:] / M[0,0]

print(M)
```

```
[[ 1.   1.   0.5  0.5]
 [ 1.   3.   1.   2. ]
 [ 1.   2.   2.  -1. ]]
```

- Passo 3: Subtraia a linha 1 da linha 2 e da linha 3

```
M[1,:] = M[1,:] - M[0,:]
M[2,:] = M[2,:] - M[0,:]

print(M)
```

```
[[ 1.   1.   0.5  0.5]
 [ 0.   2.   0.5  1.5]
 [ 0.   1.   1.5 -1.5]]
```

- Passo 4: Divida a linha 2 pelo segundo elemento da segunda linha

```
M[1,:] = M[1,:] / M[1,1]

print(M)
```

```
[[ 1.    1.    0.5   0.5 ]
 [ 0.    1.    0.25  0.75]
 [ 0.    1.    1.5  -1.5 ]]
```

- Passo 5: Subtraia a linha 2 da linha 3

```
M[2,:] = M[2,:] - M[1,:]

print(M)
```

```
[[ 1.    1.    0.5   0.5 ]
 [ 0.    1.    0.25  0.75]
 [ 0.    0.    1.25 -2.25]]
```

- Passo 6: Divida a linha 3 pelo terceiro elemento da terceira linha

```
M[2,:] = M[2,:] / M[2,2]

print(M)
```

```
[[ 1.    1.    0.5   0.5 ]
 [ 0.    1.    0.25  0.75]
 [ 0.    0.    1.   -1.8 ]]
```

- Passo 7: Multiplique a linha 3 pelo terceiro elemento da segunda linha e subtraia da linha 2

```
M[1,:] = M[1,:] - M[1,2] * M[2,:]

print(M)
```

```
[[ 1.   1.   0.5  0.5]
 [ 0.   1.   0.   1.2]
 [ 0.   0.   1.  -1.8]]
```

- Passo 8: Multiplique a linha 3 pelo terceiro elemento da primeira linha e subtraia da linha 1

```
M[0,:] = M[0,:] - M[0,2] * M[2,:]

print(M)
```

```
[[ 1.   1.   0.   1.4]
 [ 0.   1.   0.   1.2]
 [ 0.   0.   1.  -1.8]]
```

- Passo 9: Subtraia a linha 2 da linha 1

```
M[0,:] = M[0,:] - M[1,:]

print(M)
```

```
[[ 1.   0.   0.   0.2]
 [ 0.   1.   0.   1.2]
 [ 0.   0.   1.  -1.8]]
```

Temos a nossa matriz em sua forma escalonada reduzida por linhas! A solução do sistema é tal que:

```
print('A solução de x1 é: {:.2f}'.format(M[0,3]))
print('A solução de x2 é: {:.2f}'.format(M[1,3]))
print('A solução de x3 é: {:.2f}'.format(M[2,3]))
```

```
A solução de x1 é: 0.20
A solução de x2 é: 1.20
A solução de x3 é: -1.80
```

Uma forma mais simples de chegar na forma escalonada reduzida por linhas é através do pacote `SymPy` e das funções `Matrix` e `rref`:

```
import sympy

M_sympy = sympy.Matrix([(2, 2, 1, 1),
                        (1, 3, 1, 2),
                        (1, 2, 2, -1)])

M_sympy.rref()[0]
```

$$\begin{bmatrix} 1 & 0 & 0 & \frac{1}{5} \\ 0 & 1 & 0 & \frac{6}{5} \\ 0 & 0 & 1 & -\frac{9}{5} \end{bmatrix}$$

### 7.4.2 Solução de sistemas exatamente identificados

Uma outra forma de resolver sistemas de equações exatamente identificados, quando o número de incógnitas é igual ao número de equações (i.e., matriz $A$ é quadrada) é através da matriz inversa de $A$. Para o caso em que $A^{-1}$ existe, o sistema é tal que

$$\vec{x} = A^{-1} \cdot \vec{b}$$

Para que seja possível fazer dessa forma, $A$ deve ser uma matriz quadrada e seu determinante ser diferente de 0. Mais uma vez assuma o caso em que $A$ e $\vec{b}$ são dados por:

$$A = \begin{bmatrix} 2 & 2 & 1 \\ 1 & 3 & 1 \\ 1 & 2 & 2 \end{bmatrix} \qquad \vec{b} = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix}$$

O primeiro passo é conferir se a matrix $A$ é quadrada. Podemos fazer isso usando a função `numpy.shape()`

```
b = np.array([1, 2, -1])
A = np.array([(2, 2, 1), (1, 3, 1), (1, 2, 2)])

np.shape(A)[0] == np.shape(A)[1]
```

```
True
```

O próximo passo é ver se o determinante da matriz é igual a 0 e para isso usamos um função do subpacote de álgebra linera do numpy, `numpy.linalg.det()`:

```
print('A =')
print(A)

print('\nDeterminante = ',np.round(np.linalg.det(A)))
```

```
A =
[[2 2 1]
 [1 3 1]
 [1 2 2]]

Determinante =  5.0
```

Como ambas as condições são satisfeitas, por fim basta calcular a inversa da matriz $A$ com `numpy.linalg.inv()` e multiplicar por $\vec{b}$ para chegar nos valores de `x`, `y` e `z` que solucionam o sistema de equações.

```
print('A =')
print(A)

print("\nInversa de A = ")
print(np.linalg.inv(A))
```

```
A =
[[2 2 1]
 [1 3 1]
 [1 2 2]]

Inversa de A =
[[ 0.8 -0.4 -0.2]
 [-0.2  0.6 -0.2]
 [-0.2 -0.4  0.8]]
```

```
solution = np.linalg.inv(A) @ b
solution
```

```
array([ 0.2,  1.2, -1.8])
```

```
print('A solução de x1 é: {:.2f}'.format(solution[0]))
print('A solução de x2 é: {:.2f}'.format(solution[1]))
print('A solução de x3 é: {:.2f}'.format(solution[2]))
```

```
A solução de x1 é: 0.20
A solução de x2 é: 1.20
A solução de x3 é: -1.80
```

Uma última alternativa para resolver o sistema $A \cdot \vec{x} = \vec{b}$, é utilizar a função `solve` do subpacote de álgebra linear do `NumPy`. Essa função faz o processo de resolução do sistema de forma direta e nos cospe o vetor de resultado.

```
np.linalg.solve(A,b)
```

```
array([ 0.2,  1.2, -1.8])
```

# 8 Gestão e análise de dados

Uma das principais aplicações para o Python bla bla bla

---

## 8.1 O que é o NumPy?

NumPy (**Num**erical **Py**thon) é o pacote fundamental

# 9 Visualização

Uma das principais aplicações para o Python bla bla bla

---

## 9.1 O que é o NumPy?

NumPy (**Num**erical **Py**thon) é o pacote fundamental

# 10 Análise empírica integrada

Uma das principais aplicações para o Python bla bla bla

---

## 10.1 O que é o NumPy?

NumPy (**Num**erical **Py**thon) é o pacote fundamental

# Parte III

# Temas complementares

Este é o material da disciplina **EAE1106 — Métodos Computacionais para Economia**, ministrada no Departamento de Economia da USP.

O curso tem como objetivo introduzir lógica de programação e análise de dados utilizando principalmente **Python**, com uma breve introdução a **R** ao final do semestre.

# 11 Introdução à programação orientada a objetos (OOP)

In summary, this book has no content whatsoever.

# 12 Introdução ao R

In summary, this book has no content whatsoever.

# References