Curso_Python_Semuni_Jupyter

September 27, 2021

1 Curso de Python: Semuni UnB 2021

O curso vai ter 5 dias de duração. Cada dia vai ter 1 hora e meia de exposição e meia hora para revisão e para responder perguntas no youtube.

Os 5 dias vão ser assim:

- **DIA 1.** Introdução. O que é programação? Por que programar? Linguagens de Programação. Por que escolher o Python? História do Python. Onde Python é utilizado? Instalando o Python. O que são Ambientes de Desenvolvimento Integrados? Sublime, Visual Studio, PyCharm, IDLE, Atom. Anaconda. Spyder. Jupyter Notebook e Google Colab. Exemplo de código. Input e Print. *Tarefa de Casa.*
- **DIA 2.** Variáveis e tipos de dados no Python. Strings. Inteiros e Floats. Listas, Tuplas, Conjuntos e Dicionários. Conversão. Booleanos. Condicionais. Loops. Erros no código. *Tarefa de Casa.*
- **DIA 3.** Funções. Por que usar funções? Parâmetros de entrada e de saída. Parâmetros predefinidos, *args e* kwargs. *Tarefa de Casa.*
- **DIA 4.** Classes e Objetos. Métodos. Classes definidas pelo Python. Classes definidas pelo usuário. Herança. Programação orientada a objetos. *Tarefa de Casa.*
- **DIA 5.** Usando bibliotecas externas. Numpy. Plotando gráficos com o matplotlib. Bibliotecas e Frameworks. Próximos passos. Projetos, Github! *Tarefa de Casa.*

---

2 Curso de Python: Dia 1 (Apresentação, Introdução, Input, Print)

2.1 Apresentação

Apresentação de slides.

Linguagens de Programação:

- Front-End: HTML, CSS, JavaScript
- Back-End: Python, C, C++, C#, Ruby, Java, PHP

- Linguagens utilizadas na computação científica: Fortran, C, C++, Matlab, Octave, Julia, Python, R

Por que escolher o Python? Python é

- open source
- gratuito
- Linguagem Interpretada.
- GPL: General Purpose Language.
- fácil de aprender (excelente para iniciantes)
- códigos mais curtos e melhores de entender
- existem muitas bibliotecas e frameworks disponíveis
- muito utilizado por grandes empresas (Google, Instagram, Amazon, Spotify, Youtube).
- muito utilizado também no meio científico (numpy, scipy, pandas, astropy)
- conversa muito bem com outras linguagens (C, C++ e Fortran)

Principais desvantagens:

- lenta
- consome mais memória do que algumas outras linguagens
- não é muito boa no desenvolvimento de aplicativos de celular

A boa notícia é que temos muitas formas de acelerar um código em python. Vamos ver isso depois.

O que dá pra fazer com Python:

- Scripts (códigos, roteiros, programas) para automatizar tarefas do dia a dia. Mover arquivos, criar arquivos, ler e-mails. Rotinas de otimizações.
- Desenvolvimento para desktop. Aplicativos do computador. Programas com interface para o usuário (usando PyQt, Tkinter, PySimpleGUI).
- Análise de Dados.
- Inteligência Artificial. Algoritmo que aprende com os dados de entrada.
- Desenvolvimento de jogos (Pygame).
- Desenvolvimento web. Desenvolvimento de sites (Django).

História do Python:

- Guido von Rossum criou o Python em 1991.
- Monty Python, grupo de comédia.
- Elegante e intuitiva.
- The Zen of Python

[1]:
```
import this
```

```
The Zen of Python, by Tim Peters
```

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Bonito é melhor que feio.

Explícito é melhor que implícito.

Simples é melhor que complexo.

Complexo é melhor que complicado.

Linear é melhor do que aninhado.

Esparso é melhor que denso.

Legibilidade conta.

Casos especiais não são especiais o bastante para quebrar as regras.

Ainda que praticidade vença a pureza.

Erros nunca devem passar silenciosamente.

A menos que sejam explicitamente silenciados.

Diante da ambiguidade, recuse a tentação de adivinhar.

Dever haver um — e preferencialmente apenas um — modo óbvio para fazer algo.

Embora esse modo possa não ser óbvio a princípio, a menos que você seja holandês.

Agora é melhor que nunca.

Apesar de que nunca normalmente é melhor do que *exatamente* agora

Se a implementação é difícil de explicar, é uma má ideia

Se a implementação é fácil de explicar, pode ser uma boa ideia

Namespaces são uma grande ideia — vamos ter mais dessas!

2.2 Instalando o Python

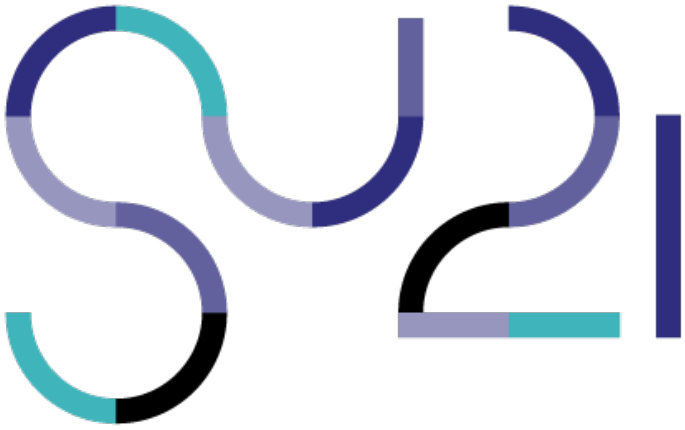
Temos várias maneiras de instalar o Python no computador.

A mais simples delas é através do Pacote Anaconda. Esse pacote possui muitas ferramentas do Python.

((Entre em https://www.anaconda.com/ . Para um passo a passo da instalação no Windows, veja esse vídeo https://youtu.be/_eK0z5QbpKA .))

Temos várias IDEs (Ambientes de Desenvolvimento Integrado) para Python. As mais famosas são: PyCharm, Visual Studio Code, Sublime Text, Atom, IDLE, Spyder.

Jupyter Notebook (essa que estou usando, já vem no pacote Anaconda): permite colocar texto, figuras e códigos no mesmo lugar.



SEMANA
UNIVERSITÁRIA
UnB 27 set · 1º out

100 anos de Paulo Freire

Google Colab: permite usar o Jupyter Notebook de maneira remota, sem precisar instalar. Basta ter uma conta no Google. A vantagem é que o código roda na nuvem, e não no computador. Dá pra usar no celular.

((Aqui no nosso curso vou usar o Jupyter.))

2.3 Funções print e input.

Vamos começar a brincadeira.

Vamos ver rapidamente essas duas funções muito utilizadas no python: *print* e *input*.

A função print é muito utilizada no Python. Ela escreve (imprime ou printa) na tela as informações do código.

Pra dar sorte, sempre o primeiro código tem que ser printar Hello World! na tela.

[1]:
```
print('Hello World!')
```

```
Hello World!
```

Podemos imprimir números também.

[2]:
```
print(42)
```

```
42
```

E variáveis.

[5]:
```
a = 5

b = 9

print(a, b)
```

```
5 9
```

Agora vamos fazer algumas contas.

[5]:
```
a = 5

b = a + 5

print(b)
```

```
10
```

[83]:
```
a = 5

b = 8*a

print(a, b)
```

```
5 40
```

[4]:
```
a = 27

b = a/3
```

```
print('O valor de a é', a, '.')

print('O valor de b é ', b, '.')
```

```
O valor de a é 27 .
O valor de b é  9.0 .
```

Podemos ter variáveis que representam palavras também.

[4]:
```
nome = 'Adriano'

print(nome)
```

```
Adriano
```

((Na próxima aula veremos mais sobre os tipos de dados existentes no Python.))

A função **input** (entrada), por outro lado, pega informações do usuário.

O que o usuário digitar vai aparecer na variável nome, no código abaixo.

[6]:
```
nome = input('Qual é o seu nome? ')

print(nome)
```

```
Qual é o seu nome? Adriano
Adriano
```

Porém, a função input nos dá uma palavra como resposta, mesmo que a entrada seja um número.

[7]:
```
idade = input('Qual é a sua idade? ')

print(idade)
```

```
Qual é a sua idade? 31
31
```

Queremos manipular com o dado de entrada, seja ele um nome ou número.

Ou seja, queremos pegar o dado de entrada, trabalhar com ele, e retornar um dado que tenha algum significado.

A entrada é o *input* e a saída é o *output*.

Um exemplo bem simples: o usuário entra com a data de nascimento e o código retorna o ano de nascimento.

Se tentarmos fazer uma conta com o dado de entrada diretamente, teremos um erro.

[8]:
```
idade = input('Qual é a sua idade hoje? ')

ano_de_nascimento = 2021 - idade
```

```
Qual é a sua idade hoje? 31
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-57395589dfec> in <module>
      1 idade = input('Qual é a sua idade hoje? ')
      2
----> 3 ano_de_nascimento = 2021 - idade

TypeError: unsupported operand type(s) for -: 'int' and 'str'
```

Note que o código não rodou.

O python encontrou um erro e parou o código.

No caso, o erro aconteceu porque eu tentei somar um número com uma palavra.

Para nos ajudar, o python indica **onde está o erro** no código e qual é o **tipo do erro**.

É muito importante saber identificar erros, pois sempre vamos encontrá-los.

((Vamos voltar nesse assunto nas próximas aulas.))

A entrada é sempre uma *string* (palavra), e não podemos fazer uma operação de subtração com um inteiro e uma *string*.

Por isso, temos que transformar o dado de entrada em um inteiro, usando a função *int*.

[9]:
```
idade = input('Qual é a sua idade hoje? ')

ano_de_nascimento = 2021 - int(idade)

print('\nVocê nasceu em', ano_de_nascimento, 'ou em', ano_de_nascimento - 1, '.
 ↪')
```

```
Qual é a sua idade hoje? 31

Você nasceu em 1990 ou em 1989 .
```

Agora funcionou.

Vamos printar agora as três informações: nome, idade e ano de nascimento.

[2]:
```
nome = input('Qual é o seu nome? ')

idade = input('Qual é a sua idade hoje? ')

ano_de_nascimento = 2021 - int(idade)

print('\nO seu nome é', nome, '. Você tem', idade, 'anos e nasceu em',
      ano_de_nascimento, 'ou em', ano_de_nascimento - 1, '.')
```

```
Qual é o seu nome? João
Qual é a sua idade hoje? 31

O seu nome é João  . Você tem 31 anos e nasceu em 1990 ou em 1989 .
```

Pronto, vamos fechar a aula de hoje com esse exemplo.

Na próxima aula vamos ver vários detalhes do código acima.

Vamos ver o que é uma **variável** e quais são os **tipos de dados** que podemos usar no Python (*strings, inteiros, floats, listas…*).

Não se preocupe se você não conseguiu entender tudo que apareceu nos códigos dessa aula. Vamos com calma!!

2.4 Tarefa de Casa

1. Instalar o Python no computador e deixar tudo pronto pra rodar. Faça alguns testes. (De preferência, instale o Anaconda.) Se você for usar o Google Colab não tem problema, mas tenha certeza de que ele está funcionando bem.
2. Peça ao usuário para entrar o seu nome, sobrenome, idade e profissão. Imprima na tela uma frase contendo essas informações.
3. Peça ao usuário para entrar com uma distância em metros. Imprima na tela a distância em quilômetros.
4. Peça ao usuário para entrar com o valor do seu salário. Imprima na tela quanto ele vai ganhar em 7 meses de trabalho.
5. Uma pessoa aplicou um determinado valor x na poupança. Se esse dinheiro ficou parado lá, qual será o valor total depois de um ano? Pesquise qual é o rendimento atual da poupança.

---

3 Curso de Python: Dia 2 (Variáveis e Tipos de Dados)

3.1 Revisão

Pra dar sorte, quando se está aprendendo uma nova linguagem de programação, o primeiro código tem que ser sempre printar *Hello World!* na tela.

print é uma função e tem um argumento entre parênteses.

print significa mostrar na tela.

[12]:
```
print("Hello")
print('World')
print('!')
```

```
Hello
World
!
```

Quando eu executo um código python ele vai lendo linha a linha o que está escrito no *script* (roteiro).

Ele vai interpretando e executando linha por linha.

Agora vamos ler um dado fornecido pelo usuário do código, usando a função **input**.

[14]:
```
nome = input('Qual é o seu nome? ')

sobrenome = input('Qual é o seu sobrenome? ')

print('\nSeu nome completo é', nome + ' ' + sobrenome + '.')
```

```
Qual é o seu nome? Adriano
Qual é o seu sobrenome? Rosa

Seu nome completo é Adriano Rosa.
```

Agora vamos pegar a entrada do usuário, em metros, e vamos printar o correspondente em quilômetros.

[17]:
```
distancia_em_m = int(input('Qual é a distância, em metros? '))

distancia_em_km = distancia_em_m/1000

print('\nA distância, em km, é de', distancia_em_km, '.')
```

```
Qual é a distância, em metros? 2222

A distância, em km, é de 2.222 .
```

3.2 Variáveis e Tipos de Dados

Uma variável armazena dados.

Você está dando um nome ao seu dado.

Isso permite acessar esse dado depois e modificá-lo.

Aqui temos um exemplo.

[18]:
```
x = 5

y = 2

z = x + y

print(z)
```

7

Temos vários tipos de dados no python:

- palavras: **strings**
- numéricos: **int** (números inteiros) e **float** (números reais)
- grupos de dados: **lists** (listas), **tuples** (tuplas), **sets** (conjuntos) e **dicts** (dicionários)
- booleanos: **True** (verdadeiro) e **False** (Falso)

Vamos ver exemplos utilizando esses tipos de dados, começando com strings.

Strings são palavras (a ideia é que você tem várias letras ou caracteres amarrados, formando uma palavra).

Strings tem aspas no início e no fim. Você pode usar aspas simples ou duplas.

Precisamos das aspas para strings porque o Python precisa saber que trata-se de um dado, e de uma variável.

Atenção: "3" é diferente de 3 para o Python. Com o 3 podemos realizar operações matemáticas, com o "3" não.

[22]:
```
a = "3"

b = 3

print(type(a))

print(type(b))
```

```
<class 'str'>
<class 'int'>
```

No exemplo acima, a e b são diferentes.

((A função *type* informa qual é o tipo de dado que a variável está representando.))

Vamos definir o número pi.

[52]:
```
pi = 3.14

print(pi)

print('pi')
```

```
3.14
pi
```

Note que primeiro o *pi* não tem aspas. Eu não estou imprimindo a palavra *pi*.

Na verdade eu estou querendo imprimir o que está guardado na memória no local que eu chamei de *pi*.

Tem uma diferença enorme aí.

Uma variável é um jeito de dar um nome ao dado.

Existe uma convenção em Python para usar letras minúsculas para variáveis, usar o nome completo e usar o underscore para separar palavras.

Na hora de dar nome às variáveis, não pode deixar espaço em branco entre as palavras do nome da variável.

Mas podemos ter espaços em branco nas *strings*.

[32]:
```
time_1 = "Flamengo"

print(time_1)

time_2 = "Vasco da Gama"

print(time_2)

time_2 = "Fluminense"

print(time_2)
```

```
Flamengo
Vasco da Gama
Fluminense
```

Note no código acima que você pode mudar o dado contido em uma variável.

Podemos fazer várias operações com strings.

Podemos 'somar' duas strings.

[34]:
```
nome = 'Forrest'
sobrenome = 'Gump'

nome_completo = nome + sobrenome

print(nome_completo)
```

```
ForrestGump
```

Essa operação é chamada de **concatenação**.

Estamos unindo, juntando ou concatenando duas strings.

Mas ainda não ficou muito legal, pois o nome completo precisa de espaço.

Espaço em branco também é uma string.

Vamos concatená-lo também. Vou aproveitar para inserir um ponto de exclamação.

[35]:
```
nome = 'Forrest'
sobrenome = 'Gump'
```

```
espaco = ' '

exclamacao = '!'

nome_completo = nome + espaco + sobrenome + exclamacao

print(nome_completo)
```

```
Forrest Gump!
```

Mais um exemplo: podemos passar todas as letras para maiúsculas.

[36]:
```
nome_caixa_alta = nome.upper()

print(nome_caixa_alta)
```

```
FORREST
```

[37]:
```
print(nome_completo.upper())
```

```
FORREST GUMP!
```

Podemos capitalizar uma *string*.

[39]:
```
a = "adriano"

a.capitalize()
```

[39]:
```
'Adriano'
```

Dá repetir uma *string* várias vezes.

[42]:
```
a = 'nome '*4

print(a)
```

```
nome nome nome nome
```

Existem várias outras maneiras de manipular *strings*. Você pode dividir *strings*, contar o número de letras, acrescentar letras, ….

Vamos ver mais algumas ao longo do curso.

Show!

Vamos passar para os números agora.

Inteiro é diferente de **float**.

O tipo de dado **int** (*integer*, ou inteiro) no python é usado para representar números inteiros.

O **float**, ou número de ponto flutuante, é usado para representar números reais, com parte decimal.

[47]:
```
a = 1
b = 1.0

print('Aqui temos um inteiro: ', a)

print('Aqui temos um float: ', b)

c = 1.553

print('Aqui temos outro float: ', c)

print('\n')

print(type(a))
print(type(b))
```

```
Aqui temos um inteiro:  1
Aqui temos um float:  1.0
Aqui temos outro float:  1.553


<class 'int'>
<class 'float'>
```

Também é possível atribuir valores a mais de uma variável em uma mesma linha de código.

[53]:
```
a, b, c = 1, 1.0, 1.553

print('Aqui temos um inteiro: ', a)

print('Aqui temos um float: ', b)

print('Aqui temos outro float: ', c)
```

```
Aqui temos um inteiro:  1
Aqui temos um float:  1.0
Aqui temos outro float:  1.553
```

O Python armazena inteiro e float de formas diferentes na memória.

O Python tem uma precisão de 16 casas decimais para floats.

[54]:
```
x = 1.22222222222222222222222222222222222222

print(x)

#print(f'{x:.25f}')
```

```
1.2222222222222223
```

Vamos somar um **int** (inteiro) e um **float**.

[51]:
```
# Quando somamos int e float o resultado é um float.

a = 1       # Inteiro.

b = 1.0     # Float.

c = a + b   # O resultado é um float.

print(c)

print(type(c))
```

```
2.0
<class 'float'>
```

No código acima, o resultado da soma de um inteiro com um float é um float.

Note, no código acima, que o que está depois do jogo da velha # não é lido pelo Python.

Podemos fazer comentários no código dessa maneira.

É sempre importante comentar o código, para explicar os passos.

Outra maneira é usando 3 aspas.

Veja abaixo.

[55]:
```
# Isso é um comentário.

print('Aprendendo')     # Posso comentar na mesma linha também.

# print('Isso não vai aparecer.')

'''
Isso também é um comentário.
print('Isso também não vai aparecer.')
'''

print('Python!')
```

```
Aprendendo
Python!
```

Voltando, podemos fazer contas com as variáveis, usando operadores do Python.

[56]:
```
a = 3

b = 5

soma = a + b
```

```
subtracao = a - b

multiplicacao = a*b

divisao = a/b

potencia = a**b

potencia2 = pow(a,b)

print(soma)
print(subtracao)
print(multiplicacao)
print(divisao)
print(potencia)
print(potencia2)
```

```
8
-2
15
0.6
243
243
```

Abaixo temos um código que converte temperatura em graus Celsius para Kelvin.

[58]:
```
temperatura_em_graus_Celsius = 27

temperatura_em_Kelvin = temperatura_em_graus_Celsius + 273.15

print('A temperatura, em Kelvin, é de ', temperatura_em_Kelvin, '.')
```

```
A temperatura, em Kelvin, é de  300.15 .
```

Atenção: não podemos somar strings com números.

[160]:
```
a = "67"

b = 100

c = a + b
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-160-730a908753fb> in <module>
      3 b = 100
      4
----> 5 c = a + b
```

```
TypeError: can only concatenate str (not "int") to str
```

Para o Python não tem como somar um inteiro com um string.

Então ele imprime uma mensagem de erro porque ele não sabe o que fazer.

Quando há ambiguidade ou quando o python não sabe o que fazer, ele vai parar o programa e exibir uma mensagem de erro.

Acostume-se: sempre vamos ter erros no código.

Só se aprende a programar errando.

Uma habilidade muito importante é saber encontrar esses erros.

Agora, veja o exemplo abaixo. Temos duas strings e vamos tentar somá-las.

[157]:
```
a = "67"

b = "100"

c = a + b

print(c)
```

```
67100
```

O que aconteceu aqui?

A operação de + acima foi realizada em strings.

Foi feita uma concatenação de duas strings, ou seja, elas foram coladas.

Mas não exatamente isso que a gente queria.

Como eu faço pra somar 67 e 100, no exemplo acima?

Para isso, temos que transformar as strings em inteiros.

Isso pode ser feito com a função **int**.

Abaixo temos duas formas de fazer isso. Qual é a diferença entre elas?

[74]:
```
a = "67"

b = "100"

c = int(a) + int(b)

print(c)
```

```
167
```

Ou:

[59]:
```
a = int("67")

b = int("100")

c = a + b

print(c)
```

```
167
```

Um último exemplo interessante.

[61]:
```
a = 5         # Criando a variável a. Essa variável recebe o valor 5.

a = a + 5     # A variável a recebe o valor atual de a somado a 5.

print(a)

a = a + 5     # A variável a recebe o valor atual de a somado a 5.

print(a)
```

```
10
15
```

Esse exemplo é interessante para mostrar que a expressão

$a = a + 5$

tem significados diferentes na **matemática** e na **programação**.

Na matemática, esse sinal = é a igualdade. Ele indica que as quantidades dos dois lados são iguas (em valor, em propriedades, ...)

Na programação, esse sinal = significa **atribuição**: ele define ou redefine o que está armazenado em uma variável.

Na programação, esse sinal = indica que vamos pegar o que está do lado direito e vamos armazenar isso na variável indicada no lado esquerdo.

Mais alguns exemplos.

[77]:
```
x = 15

x = x + 15

print(x)
```

```
30
```

[79]:
```
x = 15
```

```
x += 15

print(x)
```

```
30
```

[80]:
```
x = 15

x -= 15

print(x)
```

```
0
```

[82]:
```
x = 15

x *= 3
print(x)
```

```
30
```

[85]:
```
x = 15

x /= 5

print(x)
```

```
3.0
```

E se tentarmos dividir por zero?

[62]:
```
x = 4

y = 0

print(x/y)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-62-5d56be054415> in <module>
      3 y = 0
      4
----> 5 print(x/y)

ZeroDivisionError: division by zero
```

Vamos ver agora as famosas **listas**.

Até aqui todas as variáveis contêm apenas um dado, uma informação.

((Na verdade as *strings* possuem vários caracteres, mas vamos voltar nisso depois.))

E se eu quiser armazenar vários dados (números ou palavras) em uma única variável?

Em Python podemos ter listas, que contêm **vários dados ao mesmo tempo**.

Para criar uma lista usamos colchetes.

A primeira pergunta é: **por que usar listas?**

Vamos ver o exemplo abaixo, com os números da mega-sena.

[64]:
```
numero_1_da_mega_sena = 2

numero_2_da_mega_sena = 29

numero_3_da_mega_sena = 39

numero_4_da_mega_sena = 49

numero_5_da_mega_sena = 52

numero_6_da_mega_sena = 58
```

Não parece meio ineficiente fazer isso?

Além de tudo, é bem chato também ficar repetindo as linhas.

É aí que as listas entram.

[65]:
```
numeros_da_mega_sena = [2, 29, 39, 49, 52, 58]

print(numeros_da_mega_sena)
```

```
[2, 29, 39, 49, 52, 58]
```

Agora estamos armazenando 6 números em uma única variável.

Podemos ter strings e números.

[66]:
```
minha_lista = ['Forrest', 'Gump', 1993]

print(minha_lista)
```

```
['Forrest', 'Gump', 1993]
```

Como fazer para acessar cada elemento?

[67]:
```
print(minha_lista[0])

print(minha_lista[1])
```

```
print(minha_lista[2])
```

```
Forrest
Gump
1993
```

[68]:
```
print('Run, ', minha_lista[0], ' run!')
```

```
Run,  Forrest  run!
```

Atenção: em Python toda contagem começa do zero.

O primeiro elemento de uma lista é o elemento na posição zero.

Note que os colchetes têm funções diferentes nos dois casos.

Primeiro, se eu estou criando uma lista, então os colchetes delimitam os termos que estarão nessa lista.

No segundo caso, se eu uso colchetes colados a uma variável que representa uma lista, então eu estou acessando um elemento dessa lista.

[69]:
```
print(minha_lista[2])
```

```
1993
```

[70]:
```
print(minha_lista[3])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-70-427cb84d8934> in <module>
----> 1 print(minha_lista[3])

IndexError: list index out of range
```

Note que apareceu um novo tipo de erro.

Eu tentei acessar um elemento da lista que não existe.

Como a lista tem apenas 3 elementos, temos apenas as posições 0, 1 e 2.

O Python vê o erro e para de rodar o código.

O tamanho da lista pode ser obtido com a função abaixo.

[71]:
```
len(minha_lista)    # len vem de length, que é extensão ou tamanho.
```

[71]:
```
3
```

O Python permite o uso de posições negativas.

[167]:
```
print(minha_lista[-1])
```

```
1993
```

[168]:
```
print(minha_lista[-2])
```

```
Gump
```

A posição -1 acessa o último elemento. A posição -2 acessa o penúltimo, e assim por diante.

Podemos criar uma lista vazia e adicionar elementos a essa lista.

[74]:
```
nova_lista = []

print(nova_lista)
```

```
[]
```

[75]:
```
nova_lista = nova_lista + [1]

print(nova_lista)
```

```
[1]
```

[76]:
```
nova_lista.append(4)

print(nova_lista)
```

```
[1, 4]
```

No código acima usamos o método *append* do Python.

Podemos modificar um elemento de uma lista.

[77]:
```
lista = ['UnB', 1963, 'Abril', 21]

print(lista)
```

```
['UnB', 1963, 'Abril', 21]
```

Eita, o ano está errado, vamos corrigir.

[78]:
```
lista[1] = 1962

print(lista)
```

```
['UnB', 1962, 'Abril', 21]
```

Agora sim!

E essa é a diferença entre listas e **tuplas**.

Tuplas não podem ser modificadas.

Imagine que você está criando uma nova variável e ela não deve ser modificada ao longo do código.

((Sim, você entendeu bem: nós não queremos variar a variável. Acontece.))

Nesse caso podemos criar uma **tupla**.

Usamos parênteses, e não colchetes, para criar uma tupla.

[79]:
```
tupla = (89, 11, 'oito')

print(tupla)
```

```
(89, 11, 'oito')
```

Também usamos colchetes para acessar o elemento de uma tupla.

[80]:
```
print(tupla[2])
```

```
oito
```

Um erro aparece quando tentamos modificar um elemento de uma tupla.

Não é possível modificar uma tupla.

[81]:
```
tupla[2] = 'nove'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-81-ab83b872e396> in <module>
----> 1 tupla[2] = 'nove'

TypeError: 'tuple' object does not support item assignment
```

Um **set** é um conjunto de dados que não permite dados duplicados.

Usamos chaves para criar um set.

[83]:
```
lista = [7, 8, 35, 1, 2, 3, 1, 2, 3, 35]

conjunto = {7, 8, 35, 1, 2, 3, 1, 2, 3}

print(lista)

print(conjunto)
```

```
[7, 8, 35, 1, 2, 3, 1, 2, 3, 35]
{1, 2, 35, 3, 7, 8}
```

A ordem não importa no set.

Adicionando um elemento.

[84]:
```
conjunto.add(9)
```

```
print(conjunto)
```

```
{1, 2, 35, 3, 7, 8, 9}
```

Removendo um elemento.

[86]:
```
conjunto.discard(3)

print(conjunto)
```

```
{1, 2, 35, 7, 8, 9}
```

Set pode ser interessante quando você quer ter certeza de que não vai ter dados repetidos em sua variável.

Podemos fazer operações com set também.

Por exemplo, temos união e interseção.

[88]:
```
a = {1, 2, 3, 4}

b = {3, 4, 5, 6}

uniao = a | b

interseccao = a & b

print('A união dos conjuntos a e b é o conjunto', uniao, '.')

print('\nA intersecção dos conjuntos a e b é o conjunto', interseccao, '.')
```

```
A união dos conjuntos a e b é o conjunto {1, 2, 3, 4, 5, 6} .

A intersecção dos conjuntos a e b é o conjunto {3, 4} .
```

Um **dicionário**, ou **dict**, também serve para armazenar vários dados em uma mesma variável.

A diferença é que não usamos números para acessar os elementos de um dicionário.

Usamos as chaves, ou keys.

Para criar um dicionário nós entramos com as informações entre chaves.

[89]:
```
saramago = {'nome': 'José', 'sobrenome': 'Saramago', 'profissão': 'Escritor',␣
 ↪'ano': 1923}

print(saramago)
```

```
{'nome': 'José', 'sobrenome': 'Saramago', 'profissão': 'Escritor', 'ano': 1923}
```

Para acessar um elemento eu utilizo a **chave** desse elemento, e não a posição.

Funciona como um dicionário mesmo: você procura pela palavra (chave) e ali você encontra o significado da palavra (valor).

Você não tenta descobrir o significado de uma palavra, no dicionário, procurando pelo número da posição que essa palavra ocupa.

O número da posição em um dicionário não importa.

Vamos printar o ano de nascimento.

[113]:
```
print(saramago['ano'])
```

```
1923
```

Opa, o ano está errado. Vamos corrigir.

[91]:
```
saramago['ano'] = 1922

print(saramago)
```

```
{'nome': 'José', 'sobrenome': 'Saramago', 'profissão': 'Escritor', 'ano': 1922}
```

Vamos acessar a profissão.

[218]:
```
print(saramago['profissão'])
```

```
Escritor
```

Esquecemos de incluir a nacionalidade.

[92]:
```
saramago['nacionalidade'] = 'Portuguesa'

print(saramago)
```

```
{'nome': 'José', 'sobrenome': 'Saramago', 'profissão': 'Escritor', 'ano': 1922,
'nacionalidade': 'Portuguesa'}
```

Agora sim, está completo.

Podemos printar as chaves apenas.

[8]:
```
print(saramago.keys())
```

```
dict_keys(['nome', 'sobrenome', 'profissão', 'ano', 'nacionalidade'])
```

Podemos também printar apenas os valores.

[93]:
```
print(saramago.values())
```

```
dict_values(['José', 'Saramago', 'Escritor', 1922, 'Portuguesa'])
```

Vamos escrever uma frase usando o nosso dicionário.

[94]:
```
print('O', saramago['profissão'], saramago['nome'] + ' ' +␣
↪saramago['sobrenome'],
      'nasceu em', saramago['ano'])
```

```
O Escritor José Saramago nasceu em 1922
```

Vamos criar um novo dicionário. As keys são as mesmas, mas vamos preencher os values com os dados de outro escritor.

[95]:
```
amado = {'nome': 'Jorge', 'sobrenome': 'Amado',
                'profissão': 'Escritor', 'ano': 1912}

print(amado)
```

```
{'nome': 'Jorge', 'sobrenome': 'Amado', 'profissão': 'Escritor', 'ano': 1912}
```

Agora vamos reescrever a mesma frase acima, adaptando-a.

[96]:
```
print('O', amado['profissão'], amado['nome'] + ' ' + amado['sobrenome'],
      'nasceu em', amado['ano'])
```

```
O Escritor Jorge Amado nasceu em 1912
```

Legal. Muito bonito.

Mas é muito chato ficar dando ctrl c e ctrl v no código, e mudando as informações manualmente.

Além de chato isso pode levar a **erros**.

Devemos evitar a todo custo copiar partes do código.

Seria melhor se tivesse um jeito de imprimir a frase, dado o dicionário (como entrada).

Isso é possível com a criação de **funções**.

[98]:
```
def imprimir_dados(dicionario):
    print('O', dicionario['profissão'], dicionario['nome'] + ' ' +␣
↪dicionario['sobrenome'],
      'nasceu em', dicionario['ano'], '.')

imprimir_dados(saramago)

imprimir_dados(amado)

moraes = {'ano': 1913, 'sobrenome': 'de Moraes', 'profissão': 'Poeta', 'nome':␣
↪'Vinicius'}

imprimir_dados(moraes)
```

```
O Escritor José Saramago nasceu em 1922 .
O Escritor Jorge Amado nasceu em 1912 .
O Poeta Vinicius de Moraes nasceu em 1913 .
```

Funções são extremamente importantes.

Mas vamos ver mais sobre elas nas próxima aulas.

Vamos fechar a parte de dicionários com um último exemplo.

[99]:
```
escritor = {'nome': ['José', 'Saramago'], 'nascimento': [1922, 11, 16]}

nascimento_dia = escritor['nascimento'][2]

print(nascimento_dia)
```

```
16
```

O código acima mostra que podemos ter listas dentro de dicionários.

Um valor de um dicionário pode ser uma lista.

3.3 Conversão entre tipos de dados

Algumas vezes, em nossos códigos, é necessário converter um tipo de dado em outro.

Pra isso podemos usar funções intrínsecas do Python.

Transformando *float* em inteiro.

[27]:
```
int(3.98)    # Transforma em inteiro, truncando.
```

[27]:
```
3
```

Transformando *float* em inteiro, com aproximação.

[28]:
```
round(3.98)    # Transforma em inteiro, aproximando.
```

[28]:
```
4
```

Transformando inteiro em *float*.

[100]:
```
float(9)    # Transforma em float.
```

[100]:
```
9.0
```

Transformando *string* em *float*.

[101]:
```
float('99')
```

[101]:
```
99.0
```

Mas temos que ter apenas números.

[102]:
```
float('99a')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-102-86c915b8b8c7> in <module>
----> 1 float('99a')

ValueError: could not convert string to float: '99a'
```

Podemos transformar listas em tuples, e vice-versa.

[103]:
```
print(list((1, 2, 3)))

print(tuple([1, 2, 3]))
```

```
[1, 2, 3]
(1, 2, 3)
```

Para sabermos o tipo de uma variável, podemos usar a função *type*.

[104]:
```
type(9)
```

[104]:
```
int
```

[42]:
```
type(9.0)
```

[42]:
```
float
```

[43]:
```
type('9')
```

[43]:
```
str
```

[44]:
```
type([9])
```

[44]:
```
list
```

[49]:
```
type((9,))
```

[49]:
```
tuple
```

[107]:
```
type({'nome': 'José'})
```

[107]:
```
dict
```

3.4 Condicionais

Faltou falar de um tipo de dado no Python: o **booleano** (bool).

Um booleano pode ser ***True*** (verdadeiro) or ***False*** (falso).

E acabou.

[108]:
```
a = True

b = False

print(a, b)

print(a + b)    # Nessa operação, True é transformado em 1 e False em 0.
```

```
True False
1
```

Os booleanos são usados nos **condicionais**.

Em muitas situações, você quer executar algumas linhas do seu código somente se determinadas condições são obedecidas.

Ou seja, você pode querer que uma linha de seu código seja ou não executada, dependendo da situação.

[110]:
```
a = True

if a:
    print('Printou!')
```

```
Printou!
```

[111]:
```
a = False

if a:
    print('Não printou!')
```

[112]:
```
my_number = 128

if my_number > 100:
    print(my_number, "é maior")

if my_number < 100:
    print(my_number, "é menor")

#> maior
#< menor
#>= maior ou igual
#<= menor ou igual
#== igual
#!= diferente
```

```
128 é maior
```

[113]:
```
my_number = 30

if my_number != 35:
    print('Meu número não é 35.')
else:
    print('Meu número é 35.')
```

```
Meu número não é 35.
```

[3]:
```
my_number = 100

if my_number > 100:
    print("Meu número é maior do que 100.")
elif my_number == 100:
    print("Meu número é igual a 100.")
else:
    print("Meu número é menor do que 100.")
```

```
Meu número é igual a 100.
```

É importante notar que o espaço em branco antes do código (**indentação**) identifica os blocos do código que devem ser executados de diferentes maneiras.

((Em algumas linguagens a indentação não é importante. No Python ela é fundamental.))

Essas expressões de comparação são, na verdade, *booleanos*.

[123]:
```
a = 1

print(a == 1)
```

```
True
```

[124]:
```
print(a != 1)
```

```
False
```

[125]:
```
1 == 2
```

[125]: False

[40]:
```
1 != 2
```

[40]: True

[41]:
```
1 < 2
```

[41]: True

[42]:
```
1 > 2
```

[42]:
```
False
```

[126]:
```
if 1 < 2:
    print('Olá!')
```

```
Olá!
```

[127]:
```
if 1 > 2:
    print('Olá!')
```

Nos condicionais podemos ter mais de uma condição no if, usando o *and* e o *or*.

[45]:
```
a = 2

b = 4

if a == 2 and b == 3:
    print('Printou!')
```

[47]:
```
a = 2

b = 4

if a == 2 or b == 3:
    print('Printou!')
```

```
Printou!
```

Entendeu a diferença entre *and* e *or*?

O *and* exige as duas condições. O *or* exige apenas uma.

Podemos ter mais de duas condições.

[128]:
```
a = 1

b = 2

c = 4

if a == 1 and b == 2 and c == 3:
    print('Printou!')
```

[131]:
```
a = 1

b = 2
```

```
c = 4

if a == 1 and b == 2 or c == 3:
    print('Printou!')
```

```
Printou!
```

Faça alguns testes com esses códigos acima para entender bem como o if funciona.

Os condicionais aparecem MUITO nos códigos em Python (em todas as linguagens, na verdade).

3.5 *Loops: for* e *while*

Às vezes precisamos repetir partes do código.

Quase sempre é possível fazer isso usando ***loops***.

Com *loops* (laços de repetição) podemos realizar uma mesma operação várias vezes.

Ou seja, podemos repetir um mesmo conjunto de comandos (mudando alguns valores).

Abaixo temos um exemplo com *for*.

[133]:
```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

O *for* (para), junto com o *range*, vai fazer a variável i mudar de 0 a 4.

A variável i assume o valor zero e todos os comandos dentro do *loop* são executados, considerando i como sendo zero.

Depois, chegando ao final do código que está indentado, a variável i assume o valor 1 e as operações que estão dentro do *loop* recomeçam.

Depois i assume valor 2 e assim por diante....

Podemos mudar o nome da varíavel.

[134]:
```
for j in range(7):
    print(j)
```

```
0
1
2
3
4
5
6
```

Podemos definir, no *range*, o valor inicial, o valor final e o intervalor entre os valores da variável.

[135]:
```
for i in range(3, 25, 4):
    print(i)
```

```
3
7
11
15
19
23
```

Note que a função *range* pode ter um argumento, dois ou três (faça alguns testes para entender isso bem).

Abaixo temos uma soma dos inteiros de 1 a 10.

Note que o *range* vai até 11, porque o *range* exclui o argumento de entrada. Vai de 0 até 11-1.

[292]:
```
soma = 0

for i in range(11):
    soma += i

print(soma)
```

```
55
```

Agora temos uma soma dos quadrados desses números.

[293]:
```
soma = 0

for i in range(11):
    soma += i**2

print(soma)
```

```
385
```

Vamos somar apenas os quadrados dos números pares.

[136]:
```
soma = 0

for i in range(0, 11, 2):
    soma += i**2

print(soma)
```

```
220
```

Na verdade, o *for* do Python pode iterar em uma sequência de itens, como uma lista ou uma string.

Usamos o *range* no caso particular em que queremos iterar em uma sequência de números. O *range* é uma função que gera essa lista.

Veja os exemplos abaixo.

[137]:
```
nome = 'Universidade de Brasília'

for i in nome:
    print(i)
```

```
U
n
i
v
e
r
s
i
d
a
d
e

d
e

B
r
a
s
í
l
i
a
```

[302]:
```
numeros = [4, 8, -7, 13, 1.1]

for i in numeros:
    print(i)
```

```
4
8
-7
13
1.1
```

Também podemos iterar em dicionários. Mas para isso temos que transformar seus elementos em listas.

[138]:
```
ingles_portugues = {'mechanics': 'mecânica', 'engineering': 'engenharia',
                    'university': 'universidade'}

for key, value in ingles_portugues.items():
    print(key,'->', value)
```

```
mechanics -> mecânica
engineering -> engenharia
university -> universidade
```

Sempre tem o *for*, o *in* e os dois pontos.

Tudo que está dentro do *loop*, ou seja, que deve ser repetido, deve estar indentado, que é o espaço em branco antes dos comandos.

Podemos também ter loop dentro de loop.

[140]:
```
for i in range(3):
    for j in range(3):
        print(i, j)
```

```
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

[141]:
```
for i in range(3):
    for j in range(3):
        for k in range(2):
            print(i, j, k)
```

```
0 0 0
0 0 1
0 1 0
0 1 1
0 2 0
0 2 1
1 0 0
1 0 1
1 1 0
1 1 1
1 2 0
1 2 1
2 0 0
```

```
2 0 1
2 1 0
2 1 1
2 2 0
2 2 1
```

Podemos parar o *loop* antes de chegar no final da contagem.

Para isso usamos o *break*.

[142]:
```
for i in range(20):
    if i == 7:
        break
    print(i)
```

```
0
1
2
3
4
5
6
```

O *loop* deveria ir de zero a 19. Mas quando i é igual a 7, aparece um *break* para o *loop*.

Então o *loop* acaba ali, e o código continua para as próximas linhas.

Outra função interessante para controlar o *loop* é a *continue*.

[317]:
```
for i in range(12):
    if i == 4 or i == 8:
        continue
    print(i)
```

```
0
1
2
3
5
6
7
9
10
11
```

Quando a *continue* é chamada, ela faz com que o código passe imediatamente para o próximo passo do *loop*.

Em resumo: ***break* finaliza o *loop* e *continue* vai para o próximo passo do *loop*.**

Outra forma de fazer iterações seguidas no Python é com o ***while***.

A tradução de *while* é enquanto.

O *while* funciona assim: enquanto for verdadeiro ele continua, até alguém dizer que é falso, aí ele para.

Muito **cuidado** aqui: se você não informar corretamente o critério de parada o *while* vai continuar para sempre (ou até você interromper manualmente a execução do código).

Vamos começar do mais simples para o mais complexo, por ter esse caminho mais sentido que o outro.

[144]:
```
a = 1    # Iniciando o valor da minha variável.

while a < 5:    # Começa o _loop_. Temos o while e a expressão.
    print(a)    # Operações que queremos fazer no loop.
    a = a + 1   # Atualização do valor de a.
```

```
1
2
3
4
```

O código executa a operação no corpo principal do *loop* e volta para o cabeçalho.

Aí ele pergunta: a expressão ainda é verdadeira?

Se for ele executa mais uma vez; se não for ele finaliza o *loop* e segue para as próximas linhas de código.

Algumas outras maneiras de executar o mesmo procedimento.

[330]:
```
a = 1

b = True

while b:
    print(a)
    a += 1
    if a > 4:
        b = False
```

```
1
2
3
4
```

[145]:
```
a = 1

while True:
    if a > 4:
        break
    print(a)
```

```
    a += 1    # Sem essa linha o código fica printando 1 para sempre.
```

```
1
2
3
4
```

Observe o código abaixo. Ele não printa nada. O que aconteceu?

[147]:
```
a = 10

while a < 5:
    print(a)
    a += 1
```

Vamos achar o Mínimo Múltiplo Comum entre dois números.

[148]:
```
a = 1

while True:

    if a%14 == 0 and a%18 == 0:
        print(a)
        break

    a += 1
```

```
126
```

O que essa operação % faz?

3.6 Tarefa de Casa

1. Imprima todos os números de 1 a 100 que são múltiplos de 3 ou de 5.
2. Desenvolva um jogo da forca.
3. Desenvolva um código que diz se uma determinada palavra é um palíndromo ou não.
4. Dadas as coordenadas de dois pontos no espaço, calcule a distância entre esses pontos.
5. Faça um programa que solicite o nome do usuário e depois o imprima na vertical em formato de escada.

J

JO

JOÃ

JOÃO

((Obs.: para resolver alguns dos exercícios acima você vai precisar de funções que não foram apresentadas na aula.

Para manipular *strings*, dê uma olhada em https://wiki.python.org.br/ManipulandoStringsComPython

Para os outros casos, pesquise no Google. Boa sorte!))

---

4 Curso de Python: Dia 3 (Funções)

4.1 Revisão

4.2 Funções

Funções são uma maneira de criar partes de código que podem ser utilizadas mais tarde no próprio código ou em outros códigos.

São **essenciais** para não ficarmos repetindo linhas de código e também contribuem muito para a organização do programa.

Não é bom copiar e colar partes do código porque se você encontrar um erro ou quiser fazer uma modificação, você vai ter que fazer isso em várias partes do código.

A ideia é dividir uma tarefa grande em várias **tarefas menores**. Isso é possível em quase todas as situações.

Dividir para conquistar.

Essas pequenas tarefas (ou peças) podem ser usadas várias vezes no seu código.

Vamos ver um exemplo.

[150]:
```
def soma2(x, y):     # Definindo a função.
    w = x + y        # Código indentado, dentro da função.
    return w         # Retorna o valor para o código principal.

a = soma2(1, 4)      # Chamando a função.

print(a)

b = soma2(17, -2)    # Chamando a função.

print(b)
```

```
5
15
```

Criamos a **função** *soma* no código acima.

Para definir (criar) uma função nós usamos o comando *def*.

A função é criada apenas uma vez.

O código da função deve estar indentado.

Depois de criar a função, nós a chamamos, demos um *call* na função.

A função pode ser chamada quantas vezes quisermos.

Note que a função deve ser criada primeiro e depois chamada (isso acontece sempre em linguagens interpretadas).

O que essa função faz? Nós entramos com dois valores e ela retorna a soma desses valores.

As entradas de uma função são chamadas de **argumentos** ou **parâmetros**.

Mais um exemplo.

[151]:
```
def soma3(x, y, z):
    w = x + y + z
    return w

a = soma3(7, 9, 11)

print(a)
```

```
27
```

Agora temos 3 argumentos na nossa função.

IMPORTANTE: note o espaçamento inicial nas linhas de código dentro da função.

Esse espaço em branco é chamado de **indentação** (já vimos isso em condicionais e loops).

A indentação é importantíssima no Python.

Usamos **4 espaços em branco** na indentação.

Tudo que está indentado forma um bloco de código separado do bloco principal.

Note que indentação não é apenas para organizar o código visualmente, como em outras linguagens.

Temos erros no código quando a indentação correta não é utilizada.

[152]:
```
def soma3(x, y, z):
w = x + y + z
return w
```

```
  File "<ipython-input-152-51621ada81c2>", line 2
    w = x + y + z
    ^
IndentationError: expected an indented block
```

Na função *soma2*, nós entramos com dois argumentos, e a função retorna (*return*) a variável w, que é a soma dos dois.

Na função *soma3* nós entramos com 3 argumentos, e a função retorna a soma dos 3.

Podemos reescrever os códigos acima sem a variável w.

[154]:
```
def soma2(x, y):
    return x + y

def soma3(x, y, z):
    return x + y + z

soma2(3, 4)

a = soma2(3, 4)

print(soma2(3, 4))

print(soma3(8, 9, 10))
```

```
7
27
```

Como no exemplo acima,
* podemos apenas chamar a função, * podemos associar uma variável ao resultado da função * podemos printar direto na tela o resultado da função

Podemos ter funções sem argumentos.

[155]:
```
def printar_42():
    print(42)

printar_42()
```

```
42
```

A função acima não tem argumento (e não tem return).

Ela apenas printa o número 42 na tela. Bem útil.

Não é obrigatório ter *return*.

Quando tem *return*, a função para no primeiro return que ela encontra.

[156]:
```
def printar_42():
    print('Começando.')
    return print(42)
    print('Terminando.')    # A função não vê o que está nas linhas após o␣
 ↪return.

printar_42()
```

```
Começando.
42
```

Mas atenção! Veja o código abaixo.

[157]:
```
def dois_ou_nao():
    a = 1
    if a == 2:
        return print('a é igual a 2.')
    else:
        return print('a é diferente de 2.')

dois_ou_nao()
```

```
a é diferente de 2.
```

A função acima continua depois do *return* porque o *return* não foi acessado, já que a condição do *if* não é verdadeira.

Podemos criar uma função chamada divisão, para evitar erros causados por divisões por zero.

[158]:
```
def divisao(a, b):
    '''
    Entradas: números a e b.
    Retorna o valor a/b quando b é diferente de zero.
    Retorna uma mensagem quando b é igual a zero.
    '''
    if b == 0 or b == 0.0:
        print('Não dividirás por zero!')
        return None
    else:
        return a/b

print(divisao(4, 2))

print(divisao(4, 0.0))
```

```
2.0
Não dividirás por zero!
None
```

A parte entre 3 aspas simples no início da função serve para explicar o que essa função faz.

É sempre importante comentar o código e explicar o que cada coisa faz.

Isso é fundamental pra quem vai usar o seu código e também para que você não se esqueça de como cada parte funciona.

Olha que legal:

[159]:
```
print(divisao.__doc__)
```

```
    Entradas: números a e b.
    Retorna o valor a/b quando b é diferente de zero.
```

```
    Retorna uma mensagem quando b é igual a zero.
```

Vamos ver como obter mais informações sobre as funções na próxima aula, sobre classes e objetos.

None (nada), que apareceu lá em cima, é um tipo de variável especial no Python para dizer que a variável existe, mas nada foi especificado sobre ela ainda.

[161]:
```
a = None

print(a)

print(type(a))
```

```
None
<class 'NoneType'>
```

[162]:
```
a == None
```

[162]: True

[163]:
```
a == 0
```

[163]: False

Note que *None* não significa zero. Zero é um *int*.

Ainda sobre a função *divisao*, veja o código abaixo.

[164]:
```
divisao(4)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-164-02196844c0eb> in <module>
----> 1 divisao(4)

TypeError: divisao() missing 1 required positional argument: 'b'
```

Aqui temos um erro!!

Na definição, a função *divisao* precisa de dois argumentos de entrada.

Na hora de chamar a função, apenas um argumento foi dado.

Isso gera um erro.

Com três argumentos também temos um erro.

[75]:
```
divisao(1, 2, 3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-75-b59e6ff4675e> in <module>
----> 1 divisao(1, 2, 3)

TypeError: divisao() takes 2 positional arguments but 3 were given
```

Podemos ter também, strings, listas, dicionários, tuples, e qualquer tipo de dado como argumento de uma função.

[166]:
```
def run(nome):
    print('Run, ' + nome + ', run !')

run('Forrest')
```

```
Run, Forrest, run !
```

Vamos calcular as médias dos elementos de uma lista.

[167]:
```
def calcular_media(lista):
    a = 0
    for i in lista:
        a += i

    return a/len(lista)

a = [1, 8.0, 9.0, 25.0]

media = calcular_media(a)

print(media)
```

```
10.75
```

Outro detalhe importante é que podemos retornar mais de um valor em uma função.

[168]:
```
def f(x, y, z):
    return x + 10, y + 20, z + 30

a, b, c = f(0, 0, 0)

print(a, b, c)
```

```
10 20 30
```

Default, args e kwargs

Temos alguns tipos de comportamento para os argumentos em funções no Python.

Vamos ver três deles, que são os *defaults* (predefinidos), ****args*** e *****kwargs***.

Usamos o ***default*** quando um ou mais argumentos já têm um valor comumente utilizado, meio que padrão.

Vamos criar uma função que devolve o número de semanas em um mês, dado o número de dias.

[169]:
```
def semanas_no_mes(dias=30):
    return f'Esse mês tem {dias/7} semanas.'

print(semanas_no_mes())
```

```
Esse mês tem 4.285714285714286 semanas.
```

Note que a função tem um argumento e não foi passado nenhum, na hora de chamar a função.

Isso não dá problema, porque a função já tem um valor que ela assume sem ninguém falar nada.

Agora temos um mês com 31 dias.

[9]:
```
print(semanas_no_mes(31))
```

```
Esse mês tem 4.428571428571429 semanas.
```

Estamos em fevereiro.

[7]:
```
print(semanas_no_mes(28))
```

```
Esse mês tem 4.0 semanas.
```

Vamos ver um exemplo mais completo com *default*.

Estamos finalizando um curso, que teve três provas, e queremos saber qual foi a menção final do aluno.

Vamos criar uma função que retorna o nome do aluno, a média final e a menção.

Como os alunos são ótimos, o *default* é nota 10 nas três provas.

Caso eles tenham mandado mal em alguma prova, eu vou informar à função.

[171]:
```
def calcular_media_final(nota1, nota2, nota3):

    media_final = (nota1 + nota2 + nota3)/3

    return media_final


def calcular_mencao(nome, sobrenome, nota1 = 10, nota2 = 10, nota3 = 10):

    media_final = calcular_media_final(nota1, nota2, nota3)

    if media_final >= 9.0:
        mencao = 'SS'
    elif media_final >= 7.0:
        mencao = 'MS'
```

```
    elif media_final >= 5.0:
        mencao = 'MM'
    elif media_final >= 3.0:
        mencao = 'MI'
    else:
        mencao = 'II'

    return print(f'A média final de {nome} {sobrenome} foi {media_final:.2f}.'
                 f' A menção final é {mencao}.')


calcular_mencao('Joseph', 'Klimber', 8.0, 8.0, 1.0)

calcular_mencao('Douglas', 'Adams')

calcular_mencao('Ford', 'Prefect', nota2 = 5.0, nota3 = 3.0)
```

```
A média final de Joseph Klimber foi 5.67. A menção final é MM.
A média final de Douglas Adams foi 10.00. A menção final é SS.
A média final de Ford Prefect foi 6.00. A menção final é MM.
```

Tente entender todos os detalhes do código acima. Note que estamos chamando uma função dentro de outra função.

Comentário: no código acima, temos o conceito aluno e temos três exemplares desse conceito.

Na próxima aula veremos uma maneira um pouquinho mais organizada de fazer esse mesmo procedimento, usando classes e objetos.

Aqui é só pra dar um spoiler. Não tente entender o código abaixo, apenas dê uma olhada.

[172]:
```
class Aluno:

    def __init__(self, nome, sobrenome, nota1 = 10, nota2 = 10, nota3 = 10):
        self.nome = nome
        self.sobrenome = sobrenome
        self.notas = [nota1, nota2, nota3]
        self.media = self.calcular_media()
        self.mencao = self.calcular_mencao()
        print(f'A média final de {self.nome} {self.sobrenome} foi {self.media:.
 ↪2f}.'
              f' A menção final é {self.mencao}.')

    def calcular_media(self):
        return sum(self.notas)/3

    def calcular_mencao(self):
        media = self.media
        if media >= 9.0:
```

```
            mencao = 'SS'
        elif media >= 7.0:
            mencao = 'MS'
        elif media >= 5.0:
            mencao = 'MM'
        elif media >= 3.0:
            mencao = 'MI'
        else:
            mencao = 'II'

        return mencao


joseph = Aluno('Joseph', 'Klimber', 8.0, 8.0, 1.0)
douglas = Aluno('Douglas', 'Adams')
ford = Aluno('Ford', 'Prefect', 3.0, 9.0, 5.0)

douglas.notas
```

```
A média final de Joseph Klimber foi 5.67. A menção final é MM.
A média final de Douglas Adams foi 10.00. A menção final é SS.
A média final de Ford Prefect foi 5.67. A menção final é MM.
```

[172]:
```
[10, 10, 10]
```

Na próxima aula vamos entender tudo que está aí.

Voltando para **funções**, vimos como funcionam os argumentos *default*.

Agora, e se eu não souber quantos argumentos eu quero na minha função?

E se eu quiser uma função com um número variável de argumentos?

Impossível? Não para o nosso querido Python.

Primeiro, vamos ver um exemplo simples.

[173]:
```
def printar_args(*args):
    print(args)

printar_args()

printar_args(1, 2)

printar_args(1, 2, 3)

printar_args('Universidade', 'de', 'Brasília', 1962, 'abril', 21)
```

```
()
(1, 2)
```

```
(1, 2, 3)
('Universidade', 'de', 'Brasília', 1962, 'abril', 21)
```

Viu? Eu posso entrar com quantos argumentos eu quiser.

Esses argumentos são transformados em uma *tuple* cujo nome é *args*.

Note o asterisco antes do nome *args*, na definição da função.

Atenção, eles se transformam em uma *tuple*, e não em uma *list*.

Ou seja, não podemos modificá-lo.

[174]:
```
def printar_args(*args):
    args[0] = 7
    print(args[0])

printar_args(1, 2, 3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-174-ca8a5b03db77> in <module>
      3     print(args[0])
      4
----> 5 printar_args(1, 2, 3)

<ipython-input-174-ca8a5b03db77> in printar_args(*args)
      1 def printar_args(*args):
----> 2     args[0] = 7
      3     print(args[0])
      4
      5 printar_args(1, 2, 3)

TypeError: 'tuple' object does not support item assignment
```

E se eu quiser modificar um elemento da entrada?

Podemos transformar a *tuple* de entrada em uma *list*, com o comando *list*.

[175]:
```
def printar_args(*args):
    args_lista = list(args)
    args_lista[0] = 7
    print(args)
    print(args_lista)

printar_args(1, 2, 3)
```

```
(1, 2, 3)
[7, 2, 3]
```

Vamos criar uma função que retorna o valor máximo entre os valores de entrada.

A quantidade de valores na entrada é desconhecida.

[176]:
```
def maximo(*args):
    m = args[0]
    for x in args[1:]:
        if x > m:
            m = x
    print(f'O valor máximo entre {[i for i in args]} é {m}.')
    return None

maximo(1,2,3,200)

maximo(-23, 144, 12, 19, 2)
```

```
O valor máximo entre [1, 2, 3, 200] é 200.
O valor máximo entre [-23, 144, 12, 19, 2] é 144.
```

A função acima está funcionando bem.

Mas e se nenhum número é dado como argumento de entrada?

Teremos um erro.

[177]:
```
def maximo(*args):
    m = args[0]
    for x in args[1:]:
        if x > m:
            m = x
    print(f'O valor máximo entre {[i for i in args]} é {m}.')
    return None

maximo()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-177-df8eeeae70dc> in <module>
      7     return None
      8
----> 9 maximo()

<ipython-input-177-df8eeeae70dc> in maximo(*args)
      1 def maximo(*args):
----> 2     m = args[0]
      3     for x in args[1:]:
      4         if x > m:
      5             m = x

IndexError: tuple index out of range
```

Vamos melhorar nossa função para evitar esse erro.

[179]:
```
def maximo(*args):
    if len(args) == 0:
        print('Nenhum número foi dado.')
        return None
    else:
        m = args[0]
        for x in args[1:]:
            if x > m:
                m = x
        print(f'O valor máximo entre {[i for i in args]} é {m}.')
        return None

maximo()
```

```
Nenhum número foi dado.
```

Abaixo temos outra maneira de fazer isso, um pouco mais elegante, usando *try* e *except*.

[182]:
```
def maximo(*args):

    try:
        m = args[0]
        for x in args[1:]:
            if x > m:
                m = x
        print(f'O valor máximo entre {[i for i in args]} é {m}.')
        return None

    except:
        print('Nenhum número foi dado.')
        return None

maximo()
```

```
Nenhum número foi dado.
```

E os **kwargs**? O que são?

Os *kwargs* são *KeyWord Arguments*, ou seja, argumentos que são identificados por seus nomes, suas chaves.

No caso de *kwargs*, o que importa para identificar um argumento é o seu nome, e não sua posição.

Quando usamos *kwargs*, a entrada se transforma em um dicionário.

Veja abaixo.

[183]:
```
def densidade_populacional(**kwargs):
    print(kwargs)
```

```
    print(type(kwargs))

densidade_populacional(cidade = 'Brasilia', populacao = 3000000, area = 5802)
```

```
{'cidade': 'Brasilia', 'populacao': 3000000, 'area': 5802}
<class 'dict'>
```

Quando usamos *kwargs* nós passamos os argumentos com o seu significado.

A ordem não interessa, já que temos um dicionário.

Agora vamos calcular a densidade populacional.

[185]:
```
def densidade_populacional(**kwargs):
    densidade = kwargs['populacao']/kwargs['area']
    print(f"A densidade populacional de {kwargs['cidade']} é {int(densidade)}␣
 ↪hab/km2.")
    return densidade


brasilia_densidade = densidade_populacional(cidade = 'Brasília',
                                            populacao = 3000000,
                                            area = 5802)

goiania_densidade = densidade_populacional(area = 728,
                                           cidade = 'Goiânia',
                                           populacao = 1090000)
```

```
A densidade populacional de Brasília é 517 hab/km2.
A densidade populacional de Goiânia é 1497 hab/km2.
```

Note que os argumentos no segundo *call* estão em um ordem diferente.

Mas isso não importa, porque os valores são identificados pelo nome da chave, e não pela posição na entrada.

Vamos colocar um argumento *default* nessa função.

[186]:
```
def densidade_populacional(pais = 'Brasil', **kwargs):
    densidade = kwargs['populacao']/kwargs['area']
    print(f"A densidade populacional de {kwargs['cidade']} ({pais}) é␣
 ↪{int(densidade)} hab/km2.")
    return densidade

brasilia_densidade = densidade_populacional(cidade = 'Brasília',
                                            populacao = 3000000,
                                            area = 5802)

goiania_densidade = densidade_populacional(area = 728,
                                           cidade = 'Goiânia',
```

```
                                            populacao = 1090000)

paris_densidade = densidade_populacional(pais = 'França',
                                         cidade = 'Paris',
                                         area = 105,
                                         populacao = 2000000)
```

```
A densidade populacional de Brasília (Brasil) é 517 hab/km2.
A densidade populacional de Goiânia (Brasil) é 1497 hab/km2.
A densidade populacional de Paris (França) é 19047 hab/km2.
```

Agora um exemplo com um argumento normal (posicional) e *kwargs*.

Dado o valor inicial de um produto que estamos comprando pela internet, queremos o valor final depois de aplicarmos o desconto e a taxa de entrega.

[188]:
```
def calcular_preco(valor, **kwargs):
    desconto = kwargs.get('desconto')
    if desconto:
        valor = valor*(1 - desconto/100)
    entrega = kwargs.get('entrega')
    if entrega:
        valor = valor + entrega
    return valor

preco_final = calcular_preco(200, entrega = 20, desconto = 30)

#preco_final = calcular_preco(entrega = 20, desconto = 30, 200)    # Por que␣
↪isso dá erro?

print(f'O preço final é R$ {preco_final:.2f}.')

#print(f'O preço final é R$ {valor:.2f}.')    # Isso dá erro. Por quê?
```

```
O preço final é R$ 160.00.
```

Pronto. Acho que é esse o básico sobre funções.

Pra resumir, abaixo temos uma função com argumentos posicionais, *defaults*, args e kwargs.

[190]:
```
def exemplo(x, y, z = 10, *args, **kwargs):
    print(x, y, z)
    print(args)
    print(kwargs)

exemplo(1, 2, 3, 4, 5, 6, 7, oito = 8, nove = 9, dez = 10)
```

```
1 2 3
(4, 5, 6, 7)
{'oito': 8, 'nove': 9, 'dez': 10}
```

4.3 Comentários Gerais

Aqui vamos ver algumas curiosidades sobre o Python, para que possamos entender melhor como ele funciona.

O Python tem muitas funções que são dele mesmo, de origem.

São as chamadas funções embutidas, ou funções *builtins*.

Para conhecer essas funções, usamos o comando abaixo.

[393]:
```
dir(__builtins__)
```

[393]:
```
['ArithmeticError',
 'AssertionError',
 'AttributeError',
 'BaseException',
 'BlockingIOError',
 'BrokenPipeError',
 'BufferError',
 'BytesWarning',
 'ChildProcessError',
 'ConnectionAbortedError',
 'ConnectionError',
 'ConnectionRefusedError',
 'ConnectionResetError',
 'DeprecationWarning',
 'EOFError',
 'Ellipsis',
 'EnvironmentError',
 'Exception',
 'False',
 'FileExistsError',
 'FileNotFoundError',
 'FloatingPointError',
 'FutureWarning',
 'GeneratorExit',
 'IOError',
 'ImportError',
 'ImportWarning',
 'IndentationError',
 'IndexError',
 'InterruptedError',
 'IsADirectoryError',
 'KeyError',
 'KeyboardInterrupt',
 'LookupError',
 'MemoryError',
 'ModuleNotFoundError',
```

```
'NameError',
'None',
'NotADirectoryError',
'NotImplemented',
'NotImplementedError',
'OSError',
'OverflowError',
'PendingDeprecationWarning',
'PermissionError',
'ProcessLookupError',
'RecursionError',
'ReferenceError',
'ResourceWarning',
'RuntimeError',
'RuntimeWarning',
'StopAsyncIteration',
'StopIteration',
'SyntaxError',
'SyntaxWarning',
'SystemError',
'SystemExit',
'TabError',
'TimeoutError',
'True',
'TypeError',
'UnboundLocalError',
'UnicodeDecodeError',
'UnicodeEncodeError',
'UnicodeError',
'UnicodeTranslateError',
'UnicodeWarning',
'UserWarning',
'ValueError',
'Warning',
'ZeroDivisionError',
'__IPYTHON__',
'__build_class__',
'__debug__',
'__doc__',
'__import__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'abs',
'all',
'any',
```

```
'ascii',
'bin',
'bool',
'breakpoint',
'bytearray',
'bytes',
'callable',
'chr',
'classmethod',
'compile',
'complex',
'copyright',
'credits',
'delattr',
'dict',
'dir',
'display',
'divmod',
'enumerate',
'eval',
'exec',
'filter',
'float',
'format',
'frozenset',
'get_ipython',
'getattr',
'globals',
'hasattr',
'hash',
'help',
'hex',
'id',
'input',
'int',
'isinstance',
'issubclass',
'iter',
'len',
'license',
'list',
'locals',
'map',
'max',
'memoryview',
'min',
'next',
```

```
 'object',
 'oct',
 'open',
 'ord',
 'pow',
 'print',
 'property',
 'range',
 'repr',
 'reversed',
 'round',
 'set',
 'setattr',
 'slice',
 'sorted',
 'staticmethod',
 'str',
 'sum',
 'super',
 'tuple',
 'type',
 'vars',
 'zip']
```

Essas funções (ou métodos) que aparecem primeiro, com inicial maiúscula, são avisos de erro.

Para saber mais sobre uma função, usamos o help.

Vamos ver o que esse método *ZeroDivisionError*.

[405]:
```
help(ZeroDivisionError)
```

```
Help on class ZeroDivisionError in module builtins:

class ZeroDivisionError(ArithmeticError)
 |  Second argument to a division or modulo operation was zero.
 |
 |  Method resolution order:
 |      ZeroDivisionError
 |      ArithmeticError
 |      Exception
 |      BaseException
 |      object
 |
 |  Methods defined here:
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
```

```
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Methods inherited from BaseException:
|
|  __delattr__(self, name, /)
|      Implement delattr(self, name).
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __reduce__(…)
|      Helper for pickle.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __setattr__(self, name, value, /)
|      Implement setattr(self, name, value).
|
|  __setstate__(…)
|
|  __str__(self, /)
|      Return str(self).
|
|  with_traceback(…)
|      Exception.with_traceback(tb) --
|      set self.__traceback__ to tb and return self.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from BaseException:
|
|  __cause__
|      exception cause
|
|  __context__
|      exception context
|
|  __dict__
|
|  __suppress_context__
|
|  __traceback__
|
```

```
|  args
```

Depois, temos as funções entre dois *underscores* (ou underlines).

Essas funções são chamadas de mágicas ou *dunders* (*double underscores*). Elas são internas ao Python, e geralmente não usamos essas funções.

Vamos ver a função *name*.

[404]:
```
__name__
```

[404]:
```
'__main__'
```

Essa função devolve o nome do módulo em que estamos e, nesse caso, mostra que estamos no arquivo principal.

Podemos dividir nosso código em vários arquivos diferentes, ou módulos.

Essa função *name* nos ajuda a identificar o arquivo principal, que está sendo executado.

Depois dos *dunders* vêm as funções que mais usamos na prática.

Para saber mais sobre uma função, é só dar um *help*.

[406]:
```
help(abs)
```

```
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.
```

Essa função retorna o valor absoluto do argumento.

[408]:
```
abs(-5)
```

[408]:
```
5
```

[409]:
```
abs(10.3)
```

[409]:
```
10.3
```

[410]:
```
abs(-19.2)
```

[410]:
```
19.2
```

Note que já usamos algumas dessas funções, como a *print* e a *input*.

Vamos ver mais uma função, a *min*.

[415]:
```
help(min)
```

```
Help on built-in function min in module builtins:

min(…)
    min(iterable, *[, default=obj, key=func]) -> value
    min(arg1, arg2, *args, *[, key=func]) -> value

    With a single iterable argument, return its smallest item. The
    default keyword-only argument specifies an object to return if
    the provided iterable is empty.
    With two or more arguments, return the smallest argument.
```

Ela retorna o menor dentre os argumentos ou o menor valor de uma lista, por exemplo.

[412]:
```
min(1, 3, 5, -2, -7, 19)
```

[412]:
```
-7
```

[414]:
```
a = [1, 3, -1, 0, 0, -2, -21]

min(a)
```

[414]:
```
-21
```

Uma outra função interessante é a *id*.

[191]:
```
help(id)
```

```
Help on built-in function id in module builtins:

id(obj, /)
    Return the identity of an object.

    This is guaranteed to be unique among simultaneously existing objects.
    (CPython uses the object's memory address.)
```

Ela nos dá a identidade do objeto (ou variável) em questão.

É a posição na memória onde o objeto está armazenado.

[454]:
```
a = 1

print(id(a))
```

```
94778814127424
```

A questão de alocação de memória no Python, ou seja, como o Python armazena os dados na memória do computador, é bem complexa e vai além dos objetivos desse curso.

Mas vamos usar o *id* pra tentar entender o que acontece com as variáveis dentro e fora de uma função.

Veja o exemplo abaixo.

[11]:
```
a = 10

def f():    # Nunca dê um nome tão curto e sem significado para funções, nem␣
 ↪variáveis!
    b = 5
    print(a, b)

f()
print(a)
```

```
10 5
10
```

A variável definida no código principal é global.

A variável definida na função é local.

Podemos usar uma variável global dentro da função (não é boa prática, mas dá pra usar).

Mas não podemos usar uma variável local fora de sua função de origem. A variável local é finalizada e apagada quando a função termina.

Veja o erro abaixo.

[196]:
```
a = 10

def f():
    d = 5
    print(a, d)

f()
print(a, d)    # Aqui está o erro.
```

```
10 5
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-196-552f77622182> in <module>
      6
      7 f()
----> 8 print(a, d)    # Aqui está o erro.

NameError: name 'd' is not defined
```

Quando eu modifico o valor de uma variável global dentro de uma função, essa modificação não volta para o código principal.

[197]:
```
a = 10

def f():
    a = 2
    b = 3
    print(a, b)

f()
print(a)
```

```
2 3
10
```

Vamos acompanhar a identidade das variáveis para ver o que aconteceu.

[198]:
```
a = 10

print('Antes da função: ', id(a))

def f():
    a = 2
    b = 3
    print('Dentro da função: ', id(a))

f()

print('Depois da função: ', id(a))
```

```
Antes da função:  94785693687392
Dentro da função:  94785693687136
Depois da função:  94785693687392
```

O Python não permite que você modifique, assim diretamente, o valor de uma variável global de dentro de uma função.

Isso é para sua própria segurança. Muitas vezes podemos ter variáveis dentro de funções com o mesmo nome de variáveis de fora.

E isso pode causar muita dor de cabeça.

Assim, o Python cria um novo objeto para a variável dentro da função, e protege a sua variável global.

((Obrigado, Python.))

Mas e se eu realmente quiser mudar o valor da minha variável global dentro da função?

Para fazer isso devemos declarar que a variável é global, dentro da função.

[204]:
```
a = 5

print('Antes da função: ', id(a))

def f():
    global a
    print('Dentro da função: ', id(a))

f()

print('Depois da função: ', id(a))

print(a)
```

```
Antes da função:  94785693687232
Dentro da função:  94785693687232
Depois da função:  94785693687232
5
```

Pra fechar a aula de hoje, o Python tem várias funções associadas com os tipos de dados também.

É só rodar o código abaixo para ver quais funções são essas.

Explore algumas dessas funções: veja para que servem e como utilizá-las.

É impossível (e meio inútil também) decorar todas as funções.

Vamos usando de acordo com a necessidade em nossos projetos.

[205]:
```
#dir(int)
#dir(float)
#dir(list)
#dir(dict)
dir(str)
```

[205]:
```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
```

```
'__init_subclass__',
'__iter__',
'__le__',
'__len__',
'__lt__',
'__mod__',
'__mul__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rmod__',
'__rmul__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'capitalize',
'casefold',
'center',
'count',
'encode',
'endswith',
'expandtabs',
'find',
'format',
'format_map',
'index',
'isalnum',
'isalpha',
'isascii',
'isdecimal',
'isdigit',
'isidentifier',
'islower',
'isnumeric',
'isprintable',
'isspace',
'istitle',
'isupper',
'join',
'ljust',
'lower',
'lstrip',
'maketrans',
'partition',
```

```
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

[210]:
```
nome = 'PYTHON'
nome.lower()
```

[210]:
```
'python'
```

[211]:
```
#dir(5)
#dir(5.0)
#dir([1,2])
dir('palavra')
```

[211]:
```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
```

```
'__mod__',
'__mul__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rmod__',
'__rmul__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'capitalize',
'casefold',
'center',
'count',
'encode',
'endswith',
'expandtabs',
'find',
'format',
'format_map',
'index',
'isalnum',
'isalpha',
'isascii',
'isdecimal',
'isdigit',
'isidentifier',
'islower',
'isnumeric',
'isprintable',
'isspace',
'istitle',
'isupper',
'join',
'ljust',
'lower',
'lstrip',
'maketrans',
'partition',
'replace',
'rfind',
'rindex',
'rjust',
'rpartition',
```

```
'rsplit',
'rstrip',
'split',
'splitlines',
'startswith',
'strip',
'swapcase',
'title',
'translate',
'upper',
'zfill']
```

4.4 Tarefa de Casa

1. Calcule o MMC entre dois números usando uma função. Modifique essa função para que ela aceite mais números como entrada, usando *args.
2. Horóscopo. Devolva o signo de uma pessoa, dada a sua data de nascimento. Aproveite para incluir na saída uma frase motivacional.
3. Faça uma função que devolve o reverso do inteiro que foi dado como entrada. Por exemplo, se a entrada é 921, a saída deve ser 129.
4. Crie uma função que resolve uma equação do segundo grau do tipo $ax^2 + bx + c = 0$. As entradas são a, b, c e a saída são as duas raízes. Pesquise sobre números complexos em Python.
5. (Difícil) Pesquise sobre *try* e *except*. Faça um código usando esse conceito.
6. (Difícil) Crie uma função que devolve a soma de dois números. Use a forma *lambda*. Pesquise sobre isso.
7. (Difícil) Abaixo temos um dicionário com a representação das letras usando código Morse. Crie uma função que tenha como entrada uma palavra e saída a traduação dessa palavra em código Morse.

[181]:
```
letras_para_morse = {
       'a':'.-', 'b':'-...', 'c':'-.-.', 'd':'-..', 'e':'.', 'f':'..-.',
       'g':'--.', 'h':'....', 'i':'..', 'j':'.---', 'k':'-.-', 'l':'.-..', 'm':
↪'--',
       'n':'-.', 'o':'---', 'p':'.--.', 'q':'--.-', 'r':'.-.', 's':'...', 't':
↪'-',
       'u':'..-', 'v':'...-', 'w':'.--', 'x':'-..-', 'y':'-.--', 'z':'--..',
       '0':'-----', '1':'.----', '2':'..---', '3':'...--', '4':'....-',
       '5':'.....', '6':'-....', '7':'--...', '8':'---..', '9':'----.', ' ':'/'
       }
```

5 Curso de Python: Dia 4 (Classes e Objetos)

5.1 Revisão

Resolvendo o último exercício da aula anterior.

[212]:
```
def traduzir_para_morse(mensagem):

    letras_para_morse = {
            'a':'.-', 'b':'-...', 'c':'-.-.', 'd':'-..', 'e':'.', 'f':'..-.',
            'g':'--.', 'h':'....', 'i':'..', 'j':'.---', 'k':'-.-', 'l':'.-..',␣
 ↪'m':'--',
            'n':'-.', 'o':'---', 'p':'.--.', 'q':'--.-', 'r':'.-.', 's':'...',␣
 ↪'t':'-',
            'u':'..-', 'v':'...-', 'w':'.--', 'x':'-..-', 'y':'-.--', 'z':'--..
 ↪',
            '0':'-----', '1':'.----', '2':'..---', '3':'...--', '4':'....-',
            '5':'.....', '6':'-....', '7':'--...', '8':'---..', '9':'----.', '␣
 ↪':'/'
            }

    morse = []

    for letra in mensagem:
        morse.append(letras_para_morse[letra.lower()])

    mensagem_morse = "  ".join(morse)

    print(f"Palavra original: {mensagem}")
    print(f"Em código Morse: {mensagem_morse}")


traduzir_para_morse("SOS mande ajuda")
```

```
Palavra original: SOS mande ajuda
Em código Morse: …  ---  …  /  --  .-  -.  -..  .  /  .-  .---  ..-  -..  .-
```

5.2 Classes e Objetos

Com classes nós podemos combinar dados e funções em um único **objeto**.

É possível organizar esses dados e funções de maneira clara, com uma hierarquia que faz sentido para o programador e para o programa.

Isso pode deixar o código muito mais organizado e facilitar o uso de partes comuns.

Na verdade dá pra fazer tudo sem usar classes e objetos. Mas eles podem facilitar muito a nossa vida.

É importante aprender sobre **Classes e Objetos** também para entender melhor como o Python funciona.

Tudo no python é objeto.

Temos que conhecer como **Classes e Objetos** funcionam para podermos usar as bibliotecas e os frameworks do Python também.

Todos utilizam classes e objetos.

Em resumo, a Programação Orientada a Objeto

> facilita a representação do mundo à nossa volta, com classes de objetos que possuem atributos e métodos e relações hierárquicas.

Vamos com calma nesse início de Classes e Objetos, pois temos muitas informações novas aqui.

Antes de entrar no assunto, vamos começar com um exemplo usando apenas dicionário e funções.

Apresento-lhes o cachorro Bob.

[217]:
```
# Criando o dicionário bob, com informações sobre o Bob.
bob = {'nome': 'Bob', 'cor': 'preta', 'idade': 10}

# Essa função printa na tela as informações do cachorro.
def descrever_cachorro(cachorro):
    print(f"O cachorro se chama {cachorro['nome']},"
          f" tem cor {cachorro['cor']} e tem {cachorro['idade']} anos de idade.
 ↪")

# Essa função printa na tela se o cachorro é jovem ou idoso.
def jovem_ou_idoso(cachorro):
    if cachorro['idade'] < 8:
        print(f"{cachorro['nome]']} é um cachorro jovem.")
    else:
        print(f"{cachorro['nome']} já está ficando idoso.")

# Chamando a primeira função.
descrever_cachorro(bob)

# Chamando a segunda função.
jovem_ou_idoso(bob)
```

```
O cachorro se chama Bob, tem cor preta e tem 10 anos de idade.
Bob já está ficando idoso.
```

Temos o dicionário *Bob*, com todas as informações sobre o *Bob*, e temos duas funções que trabalham com esse dicionário.

Funciona, o código faz tudo que queremos. Mas tem um jeito de deixá-lo mais **organizado**.

Vamos juntar tudo, **dados e funções**, numa única **classe**, chamada Cachorro.

Também melhora a nossa vida no caso de termos outros cachorros, além do *Bob*.

Então, bora reescrever esse código usando **Classe e Objeto**.

A convenção no Python é de dar nome a classes começando com letra maiúscula.

Para definir uma classe usamos a palavra *class*.

[219]:
```
class Cachorro:

    nome = 'Cachorro'
    # Outros dados e funções aqui!!

bob = Cachorro()
```

No código acima nós criamos uma classe chamada Cachorro.

Depois, criamos um exemplar dessa classe, que é o *bob*.

No linguajar do Python, nós criamos uma instância da classe Cachorro, que é o **objeto** *Bob*.

Classe (Cachorro) é um negócio abstrato, uma categoria.

Objeto (bob) é a materialização da classe, é um exemplar real dessa classe.

No vocabulário do Python, um **objeto** é uma **instância** de uma classe.

((Aqui eu confesso que às vezes eu me perco nesses nomes, mas acho que é isso mesmo.))

Essa classe tem um **atributo**, que é a variável nome.

Atributos são dados de uma classe ou instância.

Para acessar um atributo, usamos **nome do objeto** ponto **nome do atributo** .

[220]:
```
bob.nome
```

[220]:
```
'Cachorro'
```

Podemos ter **funções** definidas dentro da classe também.

Funções definidas dentro de classes são chamadas de **métodos** dessa classe.

[222]:
```
class Cachorro:    # Criando a classe.

    nome = 'Cachorro'    # Variável que pertence à classe.

    def printar_hello(self):    # Função (método) da classe. O objeto␣
↪referenciado é
                                # representado por self.
        print('Hello')

bob = Cachorro()    # Criando o objeto Bob.

print(bob.nome)    # Printando a variável nome.

bob.printar_hello()    # Executando o método printar_hello.
```

```
Cachorro
Hello
```

Temos a classe *Cachorro*. *Bob* é um objeto dessa classe.

Dentro da classe Cachorro temos a variável *nome* e a função *printar_hello*.

Para acessar esses atributos do objeto *Bob*, temos os comandos Bob.nome e Bob.printar_hello() .

O que é esse ***self*** ???

Self é usado para referenciar o objeto dentro da classe.

Você vai criar vários objetos de uma mesma classe, então não pode ficar passando sempre (manualmente) o nome desses objetos dentro da classe.

Por isso usamos o *self*.

Veja esse exemplo.

[223]:
```
class Cachorro:

    def printar_hello(self):

        self.nome = "Cachorro"    # Variável associada ao objeto.

        print('Hello')

bob = Cachorro()

bob.printar_hello()

print(bob.nome)    # Se tentarmos printar Bob.nome antes de chamar Bob.
 ↪printar_hello()
                   # vai dar erro. Por quê?
```

```
Hello
Cachorro
```

Uma **função especial** usada para inicializar uma classe é a ***init***, com dois underscores antes e dois depois.

Esse método ***init*** é chamado sempre que um novo objeto da classe é criado. Por isso ele recebe o nome de **construtor**.

Ela aparece (quase) sempre como primeiro método de uma classe.

O primeiro argumento da função ***init*** é sempre o ***self***.

O *self* representa o objeto atual que está sendo tratado na classe.

Uma classe pode ter vários objetos.

((Obs.: a ideia do *self* pode ser bem complicada pra quem nunca trabalhou com isso. Então se você não entender de primeira, não tem problema, aos poucos você vai entender.))

Vamos ver o exemplo abaixo.

[225]:
```
class Cachorro:

    def __init__(self, nome):    # Temos o self e o nome agora como argumentos.
        self.nome = nome

bob = Cachorro('Bob')
```

Quando o objeto *bob* é criado, o método ***init*** é executado **automaticamente**, uma única vez.

Estamos passando o nome *'Bob'* como argumento, na criação do objeto *bob*, que é da classe Cachorro.

O método *init* pega essa nome e cria o atributo *bob.nome* e associa a esse atributo a variável *nome*.

Note que dentro da classe, ***bob* foi substituído por *self***.

Por quê? Porque podemos ter vários objetos.

[227]:
```
class Cachorro:

    def __init__(self, nome):
        self.nome = nome
        print(f'O cachorro se chama {self.nome}.')

bob = Cachorro('Bob')    # Criando o objeto bob.

marley = Cachorro('Marley')   # Criando o objeto marley.

bidu = Cachorro('Bidu')    # Criando o objeto bidu.

print('\n')

print(bob.nome)    # Printando o atributo nome do objeto bob.

print(marley.nome)    # Printando o atributo nome do objeto marley.

print(bidu.nome)    # Printando o atributo nome do objeto bidu.
```

```
O cachorro se chama Bob.
O cachorro se chama Marley.
O cachorro se chama Bidu.


Bob
Marley
Bidu
```

No exemplo acima, *self* substitui, dentro da classe, o nome do objeto com o qual estamos trabalhando.

Primeiro *self* substitui *bob*, depois *marley* e, por fim, *bidu*.

Vamos agora criar um método específico para printar o nome do cachorro na tela.

Note que a função é definida **dentro** da classe.

Ela é uma função da classe, ou um **método** dessa classe.

[228]:
```
class Cachorro:

    def __init__(self, nome):
        self.nome = nome

    def printar_nome(self):
        print(f'O cachorro se chama {self.nome}.')

bob = Cachorro('Bob')

bob.printar_nome()

marley = Cachorro('Marley')

marley.printar_nome()
```

```
O cachorro se chama Bob.
O cachorro se chama Marley.
```

Podemos colocar para printar automaticamente, no momento em que o objeto é criado.

[19]:
```
class Cachorro:

    def __init__(self, nome):    # Método chamada automaticamente quando o␣
↪objeto é criado.
        self.nome = nome
        self.printar_nome()    # Vamos chamar o método printar_nome␣
↪automaticamente.

    def printar_nome(self):
        print(f'O cachorro se chama {self.nome}.')

bob = Cachorro('Bob')

marley = Cachorro('Marley')
```

```
O cachorro se chama Bob.
O cachorro se chama Marley.
```

No código acima, no *init* nós já estamos executando o método *printar_nome*.

Vamos adicionar mais um argumento na função *init*, que é a cor do cachorro.

[229]:
```
class Cachorro:

    def __init__(self, nome, cor):
        self.nome = nome
        self.cor = cor
        self.descrever_cachorro()

    def descrever_cachorro(self):
        print(f'O cachorro se chama {self.nome} e tem cor {self.cor}.')

bob = Cachorro('Bob', 'preta')

marley = Cachorro('Marley', 'marrom')
```

```
O cachorro se chama Bob e tem cor preta.
O cachorro se chama Marley e tem cor marrom.
```

Vamos adicionar também a idade do cachorro.

[230]:
```
class Cachorro:

    def __init__(self, nome, cor, idade):
        self.nome = nome
        self.cor = cor
        self.idade = idade
        self.descrever_cachorro()

    def descrever_cachorro(self):
        print(f'\nO cachorro se chama {self.nome}, tem cor {self.cor}'
              f' e tem {self.idade} anos de idade.')

bob = Cachorro('Bob', 'preta', 10)

marley = Cachorro('Marley', 'marrom', 3)
```

```
O cachorro se chama Bob, tem cor preta e tem 10 anos de idade.

O cachorro se chama Marley, tem cor marrom e tem 3 anos de idade.
```

Por fim, vamos adicionar o método que diz se o cachorro é jovem ou idoso.

[236]:
```
class Cachorro:

    def __init__(self, nome, cor, idade):
        self.nome = nome
        self.cor = cor
        self.idade = idade
        self.descrever_cachorro()
```

```
        self.jovem_ou_idoso()

    def descrever_cachorro(self):
        print(f'\nO cachorro se chama {self.nome}, tem cor {self.cor}'
              f' e tem {self.idade} anos de idade.')

    def jovem_ou_idoso(self):
        if self.idade < 8:
            print(f'{self.nome} é um cachorro jovem.')
        else:
            print(f'{self.nome} está ficando idoso.')


bob = Cachorro('Bob', 'preta', 10)

marley = Cachorro('Marley', 'marrom', 4)
```

```
O cachorro se chama Bob, tem cor preta e tem 10 anos de idade.
Bob está ficando idoso.

O cachorro se chama Marley, tem cor marrom e tem 4 anos de idade.
Marley é um cachorro jovem.
```

Esse código acima faz a mesma coisa que o código do início dessa aula, mas agora está escrito usando classe e objeto (Orientação a Objeto).

Veja abaixo o código original, sem o uso de Classe e Objeto.

[235]:
```
bob = {'nome': 'Bob', 'cor': 'preta', 'idade': 10}

def descrever_cachorro(cachorro):
    print(f"\nO cachorro se chama {cachorro['nome']},"
          f" tem cor {cachorro['cor']} e tem {cachorro['idade']} anos de idade.
 ↪")

def jovem_ou_idoso(cachorro):
    if cachorro['idade'] < 8:
        print(f"{cachorro['nome']} é um cachorro jovem.")
    else:
        print(f"{cachorro['nome']} já está ficando idoso.")


descrever_cachorro(bob)

jovem_ou_idoso(bob)
```

```
O cachorro se chama Bob, tem cor preta e tem 10 anos de idade.
```

```
Bob já está ficando idoso.
```

E aí, **qual forma você prefere?**

Os dois códigos fazem exatamente a mesma coisa.

Mas na forma orientada a objeto parece que temos um código mais organizado: todas as informações estão agrupadas dentro da classe.

E não tem jeito, temos que aprender a trabalhar com classes e objetos porque tudo no Python gira em torno disso.

Comentários Gerais

Podemos ver qual é o tipo de *marley* e podemos conferir também se ele pertence à classe Cachorro.

[240]:
```
type(marley)
```

[240]:
```
__main__.Cachorro
```

[241]:
```
isinstance(marley, Cachorro)
```

[241]:
```
True
```

Podemos ver todos os atributos e métodos associados a um objeto usando a função *dir*.

[242]:
```
dir(marley)
```

[242]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
```

```
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'cor',
 'descrever_cachorro',
 'idade',
 'jovem_ou_idoso',
 'nome']
```

Aparecem vários **métodos especiais** (ou mágicos), que estão entre dois *underscores*.

Eles são criados automaticamente pelo Python. Todos têm uma utilidade interna para o Python. ((Não sei quase nada sobre muitos desses métodos.))

Também aparecem os atributos e métodos que nós criamos.

Note que sempre associamos os atributos do objeto criado aos argumento de entrada, no método *init*.

O primeiro argumento do *init* é sempre o *self*.

Os outros métodos também tem o *self* como argumento.

Quando chamamos um método, estamos passando o *self* implicitamente. O *self* representa o objeto.

Os dois comandos abaixo são equivalentes.

[244]:
```
marley.descrever_cachorro()

Cachorro.descrever_cachorro(marley)
```

```
O cachorro se chama Marley, tem cor marrom e tem 4 anos de idade.

O cachorro se chama Marley, tem cor marrom e tem 4 anos de idade.
```

Podemos executar a função *init* manualmente também.

As duas linhas abaixo são equivalentes.

[55]:
```
marley.__init__('Marley', 'marrom', 4)

Cachorro.__init__(marley, 'Marley', 'marrom', 4)
```

```
O cachorro se chama Marley, tem cor marrom e tem 4 anos de idade.
Marley é um cachorro jovem.

O cachorro se chama Marley, tem cor marrom e tem 4 anos de idade.
Marley é um cachorro jovem.
```

Só para deixar nossa classe mais completa, vamos inserir alguns comentários que explicam o seu funcionamento.

[246]:
```
class Cachorro:
    '''
    Classe que descreve as características de um cachorro.
    Atributos:
     - nome: nome do cachorro
     - cor: cor do cachorro
     - idade: idade do cachorro
    Métodos:
     - descrever_cachorro: printa na tela as características do cachorro
     - jovem_ou_idoso: printa na tela se o cachorro é jovem ou idoso
    '''

    def __init__(self, nome, cor, idade):
        self.nome = nome
        self.cor = cor
        self.idade = idade
        self.descrever_cachorro()
        self.jovem_ou_idoso()

    def descrever_cachorro(self):
        '''
        Printa na tela o nome, a cor e a idade do cachorro.
        '''
        print(f'\nO cachorro se chama {self.nome}, tem cor {self.cor}'
              f' e tem {self.idade} anos de idade.')

    def jovem_ou_idoso(self):
        '''
        Printa na tela se o cachorro é jovem ou idoso.
        '''
        if self.idade < 8:
            print(f'{self.nome} é um cachorro jovem.')
        else:
            print(f'{self.nome} está ficando idoso.')


bob = Cachorro('Bob', 'preta', 10)
```

```
O cachorro se chama Bob, tem cor preta e tem 10 anos de idade.
Bob está ficando idoso.
```

Quando usamos o *help* o que aparece são essas informações que estão entre três aspas.

Veja abaixo.

[61]:
```
help(Cachorro)
#help(Cachorro.descrever_cachorro)
```

```
#help(Cachorro.jovem_ou_idoso)
```

```
Help on class Cachorro in module __main__:

class Cachorro(builtins.object)
 |  Cachorro(nome, cor, idade)
 |
 |  Classe que descreve as características de um cachorro.
 |  Atributos:
 |   - nome: nome do cachorro
 |   - cor: cor do cachorro
 |   - idade: idade do cachorro
 |  Métodos:
 |   - descrever_cachorro: printa na tela as características do cachorro
 |   - jovem_ou_idoso: printa na tela se o cachorro é jovem ou idoso
 |
 |  Methods defined here:
 |
 |  __init__(self, nome, cor, idade)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  descrever_cachorro(self)
 |      Printa na tela o nome, a cor e a idade do cachorro.
 |
 |  jovem_ou_idoso(self)
 |      Printa na tela se o cachorro é jovem ou idoso.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Essas informações são armazenadas pelo método especial *doc*.

[247]:
```
Cachorro.__doc__
```

[247]:
```
'\n    Classe que descreve as características de um cachorro.\n    Atributos:\n
- nome: nome do cachorro\n     - cor: cor do cachorro\n     - idade: idade do
cachorro\n    Métodos:\n     - descrever_cachorro: printa na tela as
características do cachorro\n     - jovem_ou_idoso: printa na tela se o cachorro
é jovem ou idoso\n    '
```

Vamos ver outro exemplo.

Tente entender o código abaixo.

[251]:
```
class ContaCorrente:

    conta_tipo = 'Corrente'

    def __init__(self, nome, numero, saldo = 0):
        self.nome = nome
        self.numero = numero
        self.saldo = saldo
        self.printar_saldo()

    def deposito(self, valor_deposito):
        self.saldo += valor_deposito
        self.printar_saldo()

    def printar_saldo(self):
        print(f'\nO saldo da conta {self.numero} é de R$ {self.saldo:.2f}.')


conta1 = ContaCorrente('José', 1111)

conta1.deposito(50)

conta1.deposito(150)

conta2 = ContaCorrente('João', 2224, 35)

conta2.deposito(35)
```

```
O saldo da conta 1111 é de R$ 0.00.

O saldo da conta 1111 é de R$ 50.00.

O saldo da conta 1111 é de R$ 200.00.

O saldo da conta 2224 é de R$ 35.00.

O saldo da conta 2224 é de R$ 70.00.
```

No código acima criamos a classe ContaCorrente.

Os argumentos de entrada na criação do objeto são: nome, número da conta e saldo.

Note que saldo é *default*, e por isso não precisa ser necessariamente declarado.

A novidade aqui é a variável *conta_tipo*.

Note que essa é uma **variável da classe**, e não de um objeto em particular.

Essa variável é comum à classe e a todos os objetos dessa classe.

As variáveis nome, conta e saldo são variáveis particulares dos objetos criados.

Veja o que acontece nos códigos abaixo.

[252]:
```
ContaCorrente.conta_tipo
```

[252]:
```
'Corrente'
```

[253]:
```
conta1.conta_tipo
```

[253]:
```
'Corrente'
```

[254]:
```
conta1.saldo
```

[254]:
```
200
```

[255]:
```
conta2.conta_tipo
```

[255]:
```
'Corrente'
```

[256]:
```
ContaCorrente.nome
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-256-6e09516ac2c1> in <module>
----> 1 ContaCorrente.nome

AttributeError: type object 'ContaCorrente' has no attribute 'nome'
```

Deu erro porque a classe ContaCorrente não tem atributo *nome*, os objetos dessa classe têm esse atributo.

Muito confuso???

Reveja os códigos acima porque agora vai piorar bastante.

Em **resumo**, temos o seguinte.

As 5 ideias da Orientação a Objeto:

1. Classes e Objetos combinam funções e dados para facilitar o uso de ambos
2. Uma Classe define o comportamento de uma nova categoria de coisas. O Objeto é essa coisa
3. Classes tem construtores que descrevem como criar um novo objeto
4. Os atributos definem as características de uma classe; os métodos definem o seu comportamento
5. Uma Clase pode herdar tudo de outra, e mudar só aquilo que interessa

3 principais recursos de Orientação:

1. Encapsulamento: capacidade de possuir dados

2. Herança: capacidade de pegar os métodos de outras classes
3. Polimorfismo: capacidade de escrever métodos de acordo com a necessidade

5.3 Herança

Uma das principais características da Orientação a Objeto é a **herança**.

Esse artifício é muito usado no Python e está presente em diversas aplicações.

A **herança** permite criar uma nova classe usando tudo que já existe em uma outra classe.

Além disso, podemos adicionar novos **atributos** e novos **métodos** nessa nova classe.

Ou seja, a **classe filha** vai herdar tudo da **classe mãe** e modificar ou acrescentar aquilo que achar necessário.

Com isso você pode usar o que já existe, sem precisar ficar reinventando a roda toda vez.

Vamos ver um exemplo.

Essa é a nossa classe Cachorro.

[257]:
```
class Cachorro:

    def __init__(self, nome, cor, idade):
        self.nome = nome
        self.cor = cor
        self.idade = idade
        self.descrever_cachorro()
        self.jovem_ou_idoso()

    def descrever_cachorro(self):
        print(f'\nO cachorro se chama {self.nome}, tem cor {self.cor}'
              f' e tem {self.idade} anos de idade.')

    def jovem_ou_idoso(self):
        if self.idade < 8:
            print(f'{self.nome} é um cachorro jovem.')
        else:
            print(f'{self.nome} está ficando idoso.')
```

Ela tem os atributos nome, cor e idade, e os métodos descrever_cachorro e jovem_ou_idoso.

Agora eu quero trabalhar com uma raça de cachorro: Pinscher.

Todo Pinscher é cachorro, mas nem todo cachorro é Pinscher (ainda bem né).

Então, na minha nova classe Pinscher, eu vou ter os **mesmos atributos** que eu tenho para um cachorro qualquer: nome, cor e idade.

Ou seja, eu posso **aproveitar** tudo que está na classe Cachorro.

Além desses atributos, eu quero criar o atributo tamanho e nervosismo.

Então nós vamos criar uma classe Pinscher que vai **herdar** a classe Cachorro.

A classe mãe é Cachorro e a classe filha é Pinscher. A classe mãe aparece na definição da classe filha.

((A classe mãe também pode ser chamada de base e a filha de derivada.))

[261]:
```
class Cachorro:

    def __init__(self, nome, cor, idade):
        self.nome = nome
        self.cor = cor
        self.idade = idade
        self.descrever_cachorro()
        self.jovem_ou_idoso()

    def descrever_cachorro(self):
        print(f'\nO cachorro se chama {self.nome}, tem cor {self.cor}'
              f' e tem {self.idade} anos de idade.')

    def jovem_ou_idoso(self):
        if self.idade < 8:
            print(f'{self.nome} é um cachorro jovem.')
        else:
            print(f'{self.nome} está ficando idoso(a).')


class Pinscher(Cachorro):    # Passamos a classe mãe entre parênteses na␣
 ↪definição.

    def __init__(self, nome, cor, idade, tamanho, nervosismo):
        self.tamanho = tamanho
        self.nervosismo = nervosismo

        # As duas linhas abaixo fazem a mesma coisa.
        # Ambas chamam o método __init__ da classe mãe.

        #Cachorro.__init__(self, nome, cor, idade)
        super().__init__(nome, cor, idade)

bob = Cachorro('Bob', 'preta', 7)    # bob é da classe Cachorro.

pretinha = Pinscher('Pretinha', 'preta', 10, 0, 10)    # pretinha é da classe␣
 ↪Pinscher.
```

```
O cachorro se chama Bob, tem cor preta e tem 7 anos de idade.
Bob é um cachorro jovem.

O cachorro se chama Pretinha, tem cor preta e tem 10 anos de idade.
```

```
Pretinha está ficando idoso(a).
```

Quando criamos o objeto *pretinha* estamos executando automaticamente o *init* da classe Pinscher.

Mas precisamos executar também o *init* da classe Cachorro, para que os atributos sejam definidos corretamente.

Para executar o *init* da classe Cachorro temos duas opções: ou usamos o nome da classe mãe explicitamente ou usamos o *super()* do Python.

O mais recomendado é usar o *super()*, pois deixa o código mais geral.

Temos três atributos que já existiam em Cachorro: nome, cor e idade.

Também adicionamos dois atributos: tamanho e nervosismo.

Podemos ver que esses atributos existem usando o *dir*.

[147]:
```
dir(pretinha)
```

[147]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'cor',
 'descrever_cachorro',
 'idade',
 'jovem_ou_idoso',
 'nervosismo',
```

```
 'nome',
 'tamanho']
```

Agora vamos ver uma propriedade muito útil quando estamos trabalhando com classes e objetos.

Na classe Cachorro nós temos o método *descrever_cachorro*.

No entanto, agora no caso da classe Pinscher, eu quero **mudar esse método**.

Eu quero mudar, na classe filha, um método que existe na classe mãe.

Eu quero acrescentar na descrição as informações sobre tamanho e grau de nervosismo.

Como fazer isso???

Temos que entrar na classe mãe e modificar o método????

E se eu não tiver acesso à classe mãe?? ((Isso é muito comum.))

É aí que entra o **polimorfismo**.

Podemos escrever um método, na classe filha, com **exatamente o mesmo nome** do método que aparece na classe mãe.

O que o Python faz???

Na hora de chamar um método, ele procura **primeiro** na classe filha.

Se o método não existe na classe filha, aí ele procura na classe mãe.

Vamos incluir um método *descrever_cachorro* na classe Pinscher.

[268]:
```
class Cachorro:

    def __init__(self, nome, cor, idade):
        self.nome = nome
        self.cor = cor
        self.idade = idade
        self.descrever_cachorro()
        self.jovem_ou_idoso()

    def descrever_cachorro(self):
        print(f'\nO cachorro se chama {self.nome}, tem cor {self.cor}'
              f' e tem {self.idade} anos de idade.')

    def jovem_ou_idoso(self):
        if self.idade < 8:
            print(f'{self.nome} é um cachorro jovem.')
        else:
            print(f'{self.nome} está ficando idoso(a).')


class Pinscher(Cachorro):
```

```
    def __init__(self, nome, cor, idade, tamanho, nervosismo):
        self.tamanho = tamanho
        self.nervosismo = nervosismo
        super().__init__(nome, cor, idade)

    def descrever_cachorro(self):
        print(f'\nO Pinscher se chama {self.nome}, tem cor {self.cor} '
              f'e tem {self.idade} anos de idade.'
              f'\nO tamanho é {self.tamanho} e o grau de '
              f'nervosismo é {self.nervosismo}.')

    def bravo_ou_manso(self):
        if self.nervosismo <= 5:
            print(f'Pode ficar tranquilo, {self.nome} é manso(a).')
        else:
            print(f'Cuidado!!! {self.nome} é nervoso(a).')


bob = Cachorro('Bob', 'preta', 7)

pretinha = Pinscher('Pretinha', 'preta', 10, 0, 10)
```

```
O cachorro se chama Bob, tem cor preta e tem 7 anos de idade.
Bob é um cachorro jovem.

O Pinscher se chama Pretinha, tem cor preta e tem 10 anos de idade.
O tamanho é 0 e o grau de nervosismo é 10.
Pretinha está ficando idoso(a).
```

Note que a descrição mudou: bob é da classe Cachorro e pretinha é da classe Pinscher.

Note também que estamos chamando a função *descrever_cachorro* de dentro da classe Cachorro.

Mesmo assim, no caso da pretinha, o método chamado foi o método da classe Pinscher.

Podemos chamar o método *bravo_ou_manso* a partir do objeto pretinha.

[264]:
```
pretinha.bravo_ou_manso()
```

```
Cuidado!!! Pretinha é nervoso(a).
```

Mas não podemos chamar esse método para o Bob.

[265]:
```
bob.bravo_ou_manso()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-265-2564ada33a9c> in <module>
----> 1 bob.bravo_ou_manso()
```

```
AttributeError: 'Cachorro' object has no attribute 'bravo_ou_manso'
```

Podemos usar o *dir* para ver que bob não possui o método _bravo_ou_manso.

[267]:
```
#dir(pretinha)
dir(bob)
```

[267]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'cor',
 'descrever_cachorro',
 'idade',
 'jovem_ou_idoso',
 'nome']
```

Pra fechar essa parte, vamos fazer uma pequena modificação no código.

Vamos chamar o método bravo_ou_manso automaticamente quando um objeto de Pinscher é criado.

[270]:
```
class Cachorro:

    def __init__(self, nome, cor, idade):
```

```
        self.nome = nome
        self.cor = cor
        self.idade = idade
        self.descrever_cachorro()
        self.jovem_ou_idoso()

    def descrever_cachorro(self):
        print(f'\nO cachorro se chama {self.nome}, tem cor {self.cor}'
              f' e tem {self.idade} anos de idade.')

    def jovem_ou_idoso(self):
        if self.idade < 8:
            print(f'{self.nome} é um cachorro jovem.')
        else:
            print(f'{self.nome} está ficando idoso(a).')


class Pinscher(Cachorro):

    def __init__(self, nome, cor, idade, tamanho, nervosismo):
        self.tamanho = tamanho
        self.nervosismo = nervosismo
        super().__init__(nome, cor, idade)
        self.bravo_ou_manso()

    def descrever_cachorro(self):
        print(f'\nO Pinscher se chama {self.nome}, tem cor {self.cor} '
              f'e tem {self.idade} anos de idade.'
              f'\nO tamanho é {self.tamanho} e o grau de '
              f'nervosismo é {self.nervosismo}.')

    def bravo_ou_manso(self):
        if self.nervosismo <= 5:
            print(f'Pode ficar tranquilo, {self.nome} é manso(a).')
        else:
            print(f'Cuidado!!! {self.nome} é nervoso(a).')


bob = Cachorro('Bob', 'preta', 7)

pretinha = Pinscher('Pretinha', 'preta', 10, 0, 10)
```

```
O cachorro se chama Bob, tem cor preta e tem 7 anos de idade.
Bob é um cachorro jovem.

O Pinscher se chama Pretinha, tem cor preta e tem 10 anos de idade.
```

```
O tamanho é 0 e o grau de nervosismo é 10.
Pretinha está ficando idoso(a).
Cuidado!!! Pretinha é nervoso(a).
```

Agora, pra mostrar o poder que o Python nos dá, vamos agora mudar o significado do sinal de mais +.

((“Com grandes poderes vêm grandes responsabilidades.” Tio Ben.))

Temos os tipos de dados.

int, *float*, *list*, *string*, ...

Para saber se um determinado objeto pertence a uma classe podemos usar a função *isinstance*.

[273]:
```
a = 1

isinstance(a, int)
```

[273]:
```
True
```

[274]:
```
a = 1.0

isinstance(a, int)
```

[274]:
```
False
```

[275]:
```
a = [1, 2, 3]

isinstance(a, list)
```

[275]:
```
True
```

Nem precisaríamos definir as variáveis.

[276]:
```
isinstance(1.0, float)
```

[276]:
```
True
```

O sinal de + é definido na classe pelo método especial *add*.

Na verdade, os dois comandos abaixo são análogos.

[277]:
```
1.0 + 1.0
```

[277]:
```
2.0
```

[278]:
```
1.0.__add__(1.0)
```

[278]:
```
2.0
```

Vamos usar a classe *float* como classe mãe. Vamos herdar todos os seus métodos.

Mas vamos modificar o método especial *add*, que define o sinal de +.

[280]:
```
class NewFloat(float):

    def __add__(self, y):    # Para quando o NewFloat está à esquerda do sinal␣
↪de +.
        print('Esse é o valor do float:', self)
        return self - y

a = NewFloat(4)

b = 10

print('Resposta:', a + b)
```

```
Esse é o valor do float: 4.0
Resposta: -6.0
```

Esse exemplo acima mostra o controle que temos sobre tudo no Python.

((Não é muito útil, mas é divertido. Se você entende o que está acontecendo aí, você já está entendendo muita coisa de Python.))

Vamos ver agora um exemplo de **várias gerações de classes**.

Temos a classe inicial (avó, nesse caso). Ela é a mais geral de todas. É a classe Pessoa.

Depois temos a classe intermediária (mãe). É a classe Jovem. Já é uma classe mais restrita. Nem todas as pessoas são jovens (temos crianças, adultos, idosos), mas todos os jovens são pessoas.

Por fim temos a classe final (filha). É a classe Estudante. Entre os jovens, temos aqueles que são estudantes.

Funciona assim:

- Todo Jovem é Pessoa, então queremos 'herdar' todos os atributos e métodos que já existem na classe Pessoa e queremos adicionar mais alguns, específicos aos jovens. Essa é a primeira etapa.
- Depois, todo Estudante é Jovem. Então queremos 'herdar' tudo que já tem na classe Jovem e adicionar mais algumas informações ou funções específicas aos estudantes.

Isso é muito útil, pois podemos aproveitar códigos que já estão prontos, e apenas acrescentar o que queremos.

Vamos ver o código.

[281]:
```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade
        self.printar_info()

    def printar_info(self):
```

```
        print('\nEste é ', self.nome, '.')

class Jovem(Pessoa):
    def __init__(self, nome, idade, musica):
        self.musica = musica
        super().__init__(nome, idade)

    def printar_info(self):
        print('\nEste é ', self.nome, '. Ele é jovem.')
        print('O estilo de música favorito é:', self.musica)

class Estudante(Jovem):
    def __init__(self, nome, idade, musica, curso):
        self.curso = curso
        super().__init__(nome, idade, musica)

    def printar_info(self):
        print('\nEste é ', self.nome, '. Ele é jovem.')
        print('O estilo de música favorito é:', self.musica)
        print('Este jovem cursa', self.curso, '.')


john = Pessoa('John', 35)
ringo = Jovem('Bob', 23, 'Rock')
paul = Estudante('Paul', 22, 'módão', 'agronomia')
```

```
Este é  John .

Este é  Bob . Ele é jovem.
O estilo de música favorito é: Rock

Este é  Paul . Ele é jovem.
O estilo de música favorito é: módão
Este jovem cursa agronomia .
```

Pra finalizar essa parte de **Classes e Objetos**, vamos dar uma olhada no exemplo abaixo, que usa o conceito de **heranças múltiplas**.

Temos as classes Corrida, Natação e Ciclismo, cada uma com seus atributos e métodos.

Queremos criar a modalidade Triathlon, que vai usar as propriedades dessas três classes.

[289]:
```
class Corrida:
    def __init__(self, distancia_corrida):
        self.distancia_corrida =  distancia_corrida
        self.tempo_corrida = 20.0

    def calcular_velocidade_media_corrida(self):
```

```
        self.velocidade_media = self.distancia_corrida/self.tempo_corrida

class Natacao:
    def __init__(self, distancia_natacao):
        self.distancia_natacao =  distancia_natacao

class Ciclismo:
    def __init__(self, distancia_ciclismo):
        self.distancia_ciclismo =  distancia_ciclismo

# Atenção: a ordem das classes entre parênteses aqui é importante,
# pois na hora de chamar os métodos, o Python vai começar a procurar
# na própria classe, e depois na classe mãe à esquerda, e depois nas próximas,
# seguindo a ordem.
class Triathlon(Corrida, Natacao, Ciclismo):

    def __init__(self, distancia_corrida, distancia_natacao,␣
 ↪distancia_ciclismo):
        Corrida.__init__(self, distancia_corrida)
        Natacao.__init__(self, distancia_natacao)
        Ciclismo.__init__(self, distancia_ciclismo)


olimpico = Triathlon(10.0, 1.5, 40.0)

sprint = Triathlon(5.0, 0.75, 20.0)

dir(olimpico)
```

[289]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
```

```
'__reduce__',
'__reduce_ex__',
'__repr__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'__weakref__',
'calcular_velocidade_media_corrida',
'distancia_ciclismo',
'distancia_corrida',
'distancia_natacao',
'tempo_corrida']
```

Nesse exemplo temos **heranças múltiplas**.

Temos uma classe filha com três mães.

A Classe Triathlon (filha) herda as propriedades das Classes mãe Corrida, Natacao e Ciclismo.

Note que não é possível usar *super()* nesse caso. ((Talvez seja, mas eu não descobri como.))

Por isso usamos o próprio nome da Classe mãe na hora de chamar o *init* de cada Classe.

O comando *dir* acima mostra que o objeto Triathlon tem os atributos e métodos das três classes mãe.

Tem muito mais coisa relacionada a herança e a herança múltipla, mas acho que já está bom por aqui.

O objetivo principal era mostrar que isso também é possível.

Vamos fechar essa aula por aqui.

5.4 Tarefa de Casa

1. Escreva uma classe chamada Circulo. O raio do círculo é dado como entrada no momento de criar um novo objeto. A classe tem dois métodos: um que calcula a área e outro que calcula a circunferência.
2. Crie a classe Complexo, que lida com números complexos. Na criação do objeto devem ser passados a parte real e a parte imaginária do número complexo. Adicione, na classe, o métodos conjugado e modulo, que retornam o conjugado do número original e o seu módulo, respectivamente.
3. Crie a classe ContaCorrente. Os dados de entrada, no momento de criação do objeto, devem ser número da conta, nome do correntista e saldo. Os métodos devem ser alterar_nome, depósito e saque. Depois de cada operação de depósito e saque o saldo atual deve ser impresso na tela. O saldo na criação da conta deve ser *default*, com valor igual a zero. Você não deve permitir que o saldo da conta fique negativo. Caso haja tentativa de saque com valor maior que o saldo, você deve printar a mensagem de saldo insuficiente, e a operação não será realizada.
4. Agora você vai usar herança entre classes. Crie uma classe mãe chamada ContaBancaria. Os objetos dessa classe possuem alguns atributos gerais, como nome, número da conta e data de nascimento. Depois, crie a classe ContaCorrente e a classe ContaPoupanca, ambas filhas da

classe ContaBancaria. Com a conta corrente, podemos fazer saques, depósitos, aplicação (e resgate) na poupança. Com a conta poupança podemos ter aumento do saldo, com aplicações feitas a partir da conta corrente, e diminuição do saldo, com os resgates, além de termos um rendimento mensal. Printe na tela, a cada operação, o saldo da conta corrente e o saldo da poupança. Inclua um simulador de saldo da poupança após determinado número de meses.

---

6 Curso de Python: Dia 5 (Bibliotecas, Numpy, Plot, Próximos Passos)

6.1 Revisão

Resolvendo um dos exercícios da tarefa de casa da aula anterior.

[308]:
```
class ContaCorrente:

    def __init__(self, nome, conta, saldo = 0):
        self.nome = nome
        self.conta = conta
        self.saldo = saldo
        self.printar_saldo()

    def alterar_nome(self, novo_nome):
        print(f'O nome do dono da conta foi alterado de {self.nome} para␣
↪{novo_nome}.')
        self.nome = novo_nome

    def printar_saldo(self):
        print(f'O saldo da conta de {self.nome} é de R$ {self.saldo:.2f}.')

    def deposito(self, valor_deposito):
        self.saldo += valor_deposito
        self.printar_saldo()

    def saque(self, valor_saque):
        saldo_novo = self.saldo - valor_saque
        if saldo_novo < 0:
            print("Operação não permitida. Saldo insuficiente.")
            self.printar_saldo()
        else:
            self.saldo = saldo_novo
            self.printar_saldo()

conta1 = ContaCorrente('João', 1234)
```

```
conta1.deposito(20)

conta1.saque(10)

conta1.saque(20)

conta1.alterar_nome('José')

conta1.deposito(30)
```

```
O saldo da conta de João é de R$ 0.00.
O saldo da conta de João é de R$ 20.00.
O saldo da conta de João é de R$ 10.00.
Operação não permitida. Saldo insuficiente.
O saldo da conta de João é de R$ 10.00.
O nome do dono da conta foi alterado de João para José.
O saldo da conta de José é de R$ 40.00.
```

6.2 Bibliotecas Externas

O Python possui algumas funções internas, como vimos nas aulas anteriores.

Essas funções podem ser usadas diretamente no seu código.

A grande vantagem do Python, porém, é o vasto repertório de **bibliotecas externas** que podem ser utilizadas.

((Obs.: temos que saber usar primeiro o que já tem no Python, para entender o seu funcionamento. Depois fica muito mais fácil entender essas bibliotecas externas.))

Bibliotecas são conjuntos de variáveis, classes e funções que podemos usar em nossos códigos.

Para usar funções dessas bibliotecas em nossos programas, usamos *import BIBLIOTECA* no início do código.

O Python possui muitas (milhares) bibliotecas externas.

Essas bibliotecas são desenvolvidas por terceiros e por empresas.

Algumas bibliotecas muito utilizadas são:

1. sys
2. os
3. Numpy
4. Matplotlib
5. Plotly
6. Pandas
7. OpenCV
8. TensorFlow
9. Scipy
10. PyGame

Algumas bibliotecas já vêm instaladas no Python, então basta dar o *import*.

Essas são as chamadas **Bibliotecas Padrão** do Python.

Dê uma olhada nesse site:

> https://docs.python.org/pt-br/3/tutorial/stdlib.html

A maioria das bibliotecas, no entanto, precisa ser instalada. Mas isso é tranquilo.

((Pesquise no Google "Python instalar biblioteca NOME DA BIBLIOTECA" e divirta-se.))

Você desenvolve o seu código e usa as funções dessas bibliotecas.

Também temos vários frameworks em Python, sendo o Django o mais famoso, principalmente para desenvolvimento web.

Diferença entre **biblioteca** e **framework**.

- A biblioteca você usa no seu código, ou seja, você faz o seu código e usa métodos dessa biblioteca dentro do seu código.
- No caso do framework, você constroi o seu código em cima do framework. O framework já tem uma estrutura pronta, você usa essa estrutura e monta o que você quer em cima.

Vamos ver aqui alguns exemplos com duas bibliotecas muito usadas na programação científica: a ***Numpy*** e a ***Matplotlib***.

6.3 Numpy

[1]:
```
import numpy as np
```

No código acima estamos importando a biblioteca *numpy* e estamos dando a ela o apelido *np*.

Ou seja, aqui no nosso código ela será chamada por *np*.

Pra conhecer as funções do *numpy*, podemos dar um *dir*.

((Dê uma olhada com calma nas funções abaixo.))

[3]:
```
dir(np)
```

[3]:
```
['ALLOW_THREADS',
 'AxisError',
 'BUFSIZE',
 'CLIP',
 'ComplexWarning',
 'DataSource',
 'ERR_CALL',
 'ERR_DEFAULT',
 'ERR_IGNORE',
 'ERR_LOG',
 'ERR_PRINT',
 'ERR_RAISE',
 'ERR_WARN',
```

```
'FLOATING_POINT_SUPPORT',
'FPE_DIVIDEBYZERO',
'FPE_INVALID',
'FPE_OVERFLOW',
'FPE_UNDERFLOW',
'False_',
'Inf',
'Infinity',
'MAXDIMS',
'MAY_SHARE_BOUNDS',
'MAY_SHARE_EXACT',
'MachAr',
'ModuleDeprecationWarning',
'NAN',
'NINF',
'NZERO',
'NaN',
'PINF',
'PZERO',
'RAISE',
'RankWarning',
'SHIFT_DIVIDEBYZERO',
'SHIFT_INVALID',
'SHIFT_OVERFLOW',
'SHIFT_UNDERFLOW',
'ScalarType',
'Tester',
'TooHardError',
'True_',
'UFUNC_BUFSIZE_DEFAULT',
'UFUNC_PYVALS_NAME',
'VisibleDeprecationWarning',
'WRAP',
'_NoValue',
'_UFUNC_API',
'__NUMPY_SETUP__',
'__all__',
'__builtins__',
'__cached__',
'__config__',
'__dir__',
'__doc__',
'__file__',
'__getattr__',
'__git_revision__',
'__loader__',
'__mkl_version__',
```

```
'__name__',
'__package__',
'__path__',
'__spec__',
'__version__',
'_add_newdoc_ufunc',
'_distributor_init',
'_globals',
'_mat',
'_pytesttester',
'abs',
'absolute',
'add',
'add_docstring',
'add_newdoc',
'add_newdoc_ufunc',
'alen',
'all',
'allclose',
'alltrue',
'amax',
'amin',
'angle',
'any',
'append',
'apply_along_axis',
'apply_over_axes',
'arange',
'arccos',
'arccosh',
'arcsin',
'arcsinh',
'arctan',
'arctan2',
'arctanh',
'argmax',
'argmin',
'argpartition',
'argsort',
'argwhere',
'around',
'array',
'array2string',
'array_equal',
'array_equiv',
'array_repr',
'array_split',
```

```
'array_str',
'asanyarray',
'asarray',
'asarray_chkfinite',
'ascontiguousarray',
'asfarray',
'asfortranarray',
'asmatrix',
'asscalar',
'atleast_1d',
'atleast_2d',
'atleast_3d',
'average',
'bartlett',
'base_repr',
'binary_repr',
'bincount',
'bitwise_and',
'bitwise_not',
'bitwise_or',
'bitwise_xor',
'blackman',
'block',
'bmat',
'bool',
'bool8',
'bool_',
'broadcast',
'broadcast_arrays',
'broadcast_to',
'busday_count',
'busday_offset',
'busdaycalendar',
'byte',
'byte_bounds',
'bytes0',
'bytes_',
'c_',
'can_cast',
'cast',
'cbrt',
'cdouble',
'ceil',
'cfloat',
'char',
'character',
'chararray',
```

```
'choose',
'clip',
'clongdouble',
'clongfloat',
'column_stack',
'common_type',
'compare_chararrays',
'compat',
'complex',
'complex128',
'complex256',
'complex64',
'complex_',
'complexfloating',
'compress',
'concatenate',
'conj',
'conjugate',
'convolve',
'copy',
'copysign',
'copyto',
'core',
'corrcoef',
'correlate',
'cos',
'cosh',
'count_nonzero',
'cov',
'cross',
'csingle',
'ctypeslib',
'cumprod',
'cumproduct',
'cumsum',
'datetime64',
'datetime_as_string',
'datetime_data',
'deg2rad',
'degrees',
'delete',
'deprecate',
'deprecate_with_doc',
'diag',
'diag_indices',
'diag_indices_from',
'diagflat',
```

```
'diagonal',
'diff',
'digitize',
'disp',
'divide',
'divmod',
'dot',
'double',
'dsplit',
'dstack',
'dtype',
'e',
'ediff1d',
'einsum',
'einsum_path',
'emath',
'empty',
'empty_like',
'equal',
'errstate',
'euler_gamma',
'exp',
'exp2',
'expand_dims',
'expm1',
'extract',
'eye',
'fabs',
'fastCopyAndTranspose',
'fft',
'fill_diagonal',
'find_common_type',
'finfo',
'fix',
'flatiter',
'flatnonzero',
'flexible',
'flip',
'fliplr',
'flipud',
'float',
'float128',
'float16',
'float32',
'float64',
'float_',
'float_power',
```

```
'floating',
'floor',
'floor_divide',
'fmax',
'fmin',
'fmod',
'format_float_positional',
'format_float_scientific',
'format_parser',
'frexp',
'frombuffer',
'fromfile',
'fromfunction',
'fromiter',
'frompyfunc',
'fromregex',
'fromstring',
'full',
'full_like',
'fv',
'gcd',
'generic',
'genfromtxt',
'geomspace',
'get_array_wrap',
'get_include',
'get_printoptions',
'getbufsize',
'geterr',
'geterrcall',
'geterrobj',
'gradient',
'greater',
'greater_equal',
'half',
'hamming',
'hanning',
'heaviside',
'histogram',
'histogram2d',
'histogram_bin_edges',
'histogramdd',
'hsplit',
'hstack',
'hypot',
'i0',
'identity',
```

```
'iinfo',
'imag',
'in1d',
'index_exp',
'indices',
'inexact',
'inf',
'info',
'infty',
'inner',
'insert',
'int',
'int0',
'int16',
'int32',
'int64',
'int8',
'int_',
'intc',
'integer',
'interp',
'intersect1d',
'intp',
'invert',
'ipmt',
'irr',
'is_busday',
'isclose',
'iscomplex',
'iscomplexobj',
'isfinite',
'isfortran',
'isin',
'isinf',
'isnan',
'isnat',
'isneginf',
'isposinf',
'isreal',
'isrealobj',
'isscalar',
'issctype',
'issubclass_',
'issubdtype',
'issubsctype',
'iterable',
'ix_',
```

```
'kaiser',
'kernel_version',
'kron',
'lcm',
'ldexp',
'left_shift',
'less',
'less_equal',
'lexsort',
'lib',
'linalg',
'linspace',
'little_endian',
'load',
'loads',
'loadtxt',
'log',
'log10',
'log1p',
'log2',
'logaddexp',
'logaddexp2',
'logical_and',
'logical_not',
'logical_or',
'logical_xor',
'logspace',
'long',
'longcomplex',
'longdouble',
'longfloat',
'longlong',
'lookfor',
'ma',
'mafromtxt',
'mask_indices',
'mat',
'math',
'matmul',
'matrix',
'matrixlib',
'max',
'maximum',
'maximum_sctype',
'may_share_memory',
'mean',
'median',
```

```
'memmap',
'meshgrid',
'mgrid',
'min',
'min_scalar_type',
'minimum',
'mintypecode',
'mirr',
'mkl',
'mod',
'modf',
'moveaxis',
'msort',
'multiply',
'nan',
'nan_to_num',
'nanargmax',
'nanargmin',
'nancumprod',
'nancumsum',
'nanmax',
'nanmean',
'nanmedian',
'nanmin',
'nanpercentile',
'nanprod',
'nanquantile',
'nanstd',
'nansum',
'nanvar',
'nbytes',
'ndarray',
'ndenumerate',
'ndfromtxt',
'ndim',
'ndindex',
'nditer',
'negative',
'nested_iters',
'newaxis',
'nextafter',
'nonzero',
'not_equal',
'nper',
'npv',
'numarray',
'number',
```

```
'obj2sctype',
'object',
'object0',
'object_',
'ogrid',
'oldnumeric',
'ones',
'ones_like',
'os',
'outer',
'packbits',
'pad',
'partition',
'percentile',
'pi',
'piecewise',
'place',
'pmt',
'poly',
'poly1d',
'polyadd',
'polyder',
'polydiv',
'polyfit',
'polyint',
'polymul',
'polynomial',
'polysub',
'polyval',
'positive',
'power',
'ppmt',
'printoptions',
'prod',
'product',
'promote_types',
'ptp',
'put',
'put_along_axis',
'putmask',
'pv',
'quantile',
'r_',
'rad2deg',
'radians',
'random',
'rate',
```

```
'ravel',
'ravel_multi_index',
'real',
'real_if_close',
'rec',
'recarray',
'recfromcsv',
'recfromtxt',
'reciprocal',
'record',
'remainder',
'repeat',
'require',
'reshape',
'resize',
'result_type',
'right_shift',
'rint',
'roll',
'rollaxis',
'roots',
'rot90',
'round',
'round_',
'row_stack',
's_',
'safe_eval',
'save',
'savetxt',
'savez',
'savez_compressed',
'sctype2char',
'sctypeDict',
'sctypeNA',
'sctypes',
'searchsorted',
'select',
'set_numeric_ops',
'set_printoptions',
'set_string_function',
'setbufsize',
'setdiff1d',
'seterr',
'seterrcall',
'seterrobj',
'setxor1d',
'shape',
```

```
'shares_memory',
'short',
'show_config',
'sign',
'signbit',
'signedinteger',
'sin',
'sinc',
'single',
'singlecomplex',
'sinh',
'size',
'sometrue',
'sort',
'sort_complex',
'source',
'spacing',
'split',
'sqrt',
'square',
'squeeze',
'stack',
'std',
'str',
'str0',
'str_',
'string_',
'subtract',
'sum',
'swapaxes',
'sys',
'take',
'take_along_axis',
'tan',
'tanh',
'tensordot',
'test',
'testing',
'tile',
'timedelta64',
'trace',
'tracemalloc_domain',
'transpose',
'trapz',
'tri',
'tril',
'tril_indices',
```

```
'tril_indices_from',
'trim_zeros',
'triu',
'triu_indices',
'triu_indices_from',
'true_divide',
'trunc',
'typeDict',
'typeNA',
'typecodes',
'typename',
'ubyte',
'ufunc',
'uint',
'uint0',
'uint16',
'uint32',
'uint64',
'uint8',
'uintc',
'uintp',
'ulonglong',
'unicode',
'unicode_',
'union1d',
'unique',
'unpackbits',
'unravel_index',
'unsignedinteger',
'unwrap',
'use_hugepage',
'ushort',
'vander',
'var',
'vdot',
'vectorize',
'version',
'void',
'void0',
'vsplit',
'vstack',
'warnings',
'where',
'who',
'zeros',
'zeros_like']
```

Acima, primeiro temos os métodos com inicial maiúscula. Esses são avisos de erro. ((Ou algo assim.))

Depois temos os métodos especiais, iniciados por um ou dois underscores.

Muito raramente nós usamos esses métodos.

Nós vamos usar mesmo as funções *abs* e todas que estão abaixo.

Note quantas funções temos.

Essa biblioteca é **muito poderosa** e muito usada na programação científica.

Vou mostrar aqui algumas funções apenas. ((Para saber mais, https://numpy.org/ .))

Para usar essas funções devemos escrever *np.NOME_DA_FUNÇÃO* .

Pra saber um pouco mais sobre uma determinada função, podemos dar um *help*.

[311]:
```
#help(np.abs)
#help(np.zeros)
#help(np.linalg)
help(np.array)
```

```
Help on built-in function array in module numpy:

array(…)
    array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0)

    Create an array.

    Parameters
    ----------
    object : array_like
        An array, any object exposing the array interface, an object whose
        __array__ method returns an array, or any (nested) sequence.
    dtype : data-type, optional
        The desired data-type for the array.  If not given, then the type will
        be determined as the minimum type required to hold the objects in the
        sequence.
    copy : bool, optional
        If true (default), then the object is copied.  Otherwise, a copy will
        only be made if __array__ returns a copy, if obj is a nested sequence,
        or if a copy is needed to satisfy any of the other requirements
        (`dtype`, `order`, etc.).
    order : {'K', 'A', 'C', 'F'}, optional
        Specify the memory layout of the array. If object is not an array, the
        newly created array will be in C order (row major) unless 'F' is
        specified, in which case it will be in Fortran order (column major).
        If object is an array the following holds.

        ===== ========= ===================================================
```

```
    order  no copy                     copy=True
    ===== ========= ===================================================
    'K'   unchanged F & C order preserved, otherwise most similar order
    'A'   unchanged F order if input is F and not C, otherwise C order
    'C'   C order   C order
    'F'   F order   F order
    ===== ========= ===================================================

    When ``copy=False`` and a copy is made for other reasons, the result is
    the same as if ``copy=True``, with some exceptions for `A`, see the
    Notes section. The default order is 'K'.
subok : bool, optional
    If True, then sub-classes will be passed-through, otherwise
    the returned array will be forced to be a base-class array (default).
ndmin : int, optional
    Specifies the minimum number of dimensions that the resulting
    array should have.  Ones will be pre-pended to the shape as
    needed to meet this requirement.

Returns
-------
out : ndarray
    An array object satisfying the specified requirements.

See Also
--------
empty_like : Return an empty array with shape and type of input.
ones_like : Return an array of ones with shape and type of input.
zeros_like : Return an array of zeros with shape and type of input.
full_like : Return a new array with shape of input filled with value.
empty : Return a new uninitialized array.
ones : Return a new array setting values to one.
zeros : Return a new array setting values to zero.
full : Return a new array of given shape filled with value.


Notes
-----
When order is 'A' and `object` is an array in neither 'C' nor 'F' order,
and a copy is forced by a change in dtype, then the order of the result is
not necessarily 'C' as expected. This is likely a bug.

Examples
--------
>>> np.array([1, 2, 3])
array([1, 2, 3])

Upcasting:
```

```
>>> np.array([1, 2, 3.0])
array([ 1.,  2.,  3.])

More than one dimension:

>>> np.array([[1, 2], [3, 4]])
array([[1, 2],
       [3, 4]])

Minimum dimensions 2:

>>> np.array([1, 2, 3], ndmin=2)
array([[1, 2, 3]])

Type provided:

>>> np.array([1, 2, 3], dtype=complex)
array([ 1.+0.j,  2.+0.j,  3.+0.j])

Data-type consisting of more than one element:

>>> x = np.array([(1,2),(3,4)],dtype=[('a','<i4'),('b','<i4')])
>>> x['a']
array([1, 3])

Creating an array from sub-classes:

>>> np.array(np.mat('1 2; 3 4'))
array([[1, 2],
       [3, 4]])

>>> np.array(np.mat('1 2; 3 4'), subok=True)
matrix([[1, 2],
        [3, 4]])
```

O *help* da *numpy* é muito detalhado, tem até alguns exemplos.

Com o *numpy* podemos criar vetores e matrizes.

Vamos usar essa função *array*.

[312]:
```
a = np.array([7.0, -3.0, 29.0])

print(a)

print('\n')
```

```
print(a[0])
print(a[1])
print(a[2])

print('\n')

print(type(a))
```

```
[ 7. -3. 29.]


7.0
-3.0
29.0


<class 'numpy.ndarray'>
```

[314]:
```
b = np.array([[5.0, 1.0, 0.0], [1.0, 7.0, 2.0], [0.0, 2.0, 10.0]])

print(b)

print('\n')

print(b[1,2])
print(b[2,2])
print(b[0,0])

print('\n')

for i in range(3):
    for j in range(3):
        print(b[i,j])
```

```
[[ 5.  1.  0.]
 [ 1.  7.  2.]
 [ 0.  2. 10.]]


2.0
10.0
5.0


5.0
1.0
0.0
1.0
```

```
7.0
2.0
0.0
2.0
10.0
```

A vantagem de usar uma matriz do *numpy* em vez de uma lista do Python é podemos fazer muitas coisas com essa matriz do *numpy*.

[315]:
```
dir(b)
```

[315]:
```
['T',
 '__abs__',
 '__add__',
 '__and__',
 '__array__',
 '__array_finalize__',
 '__array_function__',
 '__array_interface__',
 '__array_prepare__',
 '__array_priority__',
 '__array_struct__',
 '__array_ufunc__',
 '__array_wrap__',
 '__bool__',
 '__class__',
 '__complex__',
 '__contains__',
 '__copy__',
 '__deepcopy__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__divmod__',
 '__doc__',
 '__eq__',
 '__float__',
 '__floordiv__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__iand__',
 '__ifloordiv__',
 '__ilshift__',
```

```
'__imatmul__',
'__imod__',
'__imul__',
'__index__',
'__init__',
'__init_subclass__',
'__int__',
'__invert__',
'__ior__',
'__ipow__',
'__irshift__',
'__isub__',
'__iter__',
'__itruediv__',
'__ixor__',
'__le__',
'__len__',
'__lshift__',
'__lt__',
'__matmul__',
'__mod__',
'__mul__',
'__ne__',
'__neg__',
'__new__',
'__or__',
'__pos__',
'__pow__',
'__radd__',
'__rand__',
'__rdivmod__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rfloordiv__',
'__rlshift__',
'__rmatmul__',
'__rmod__',
'__rmul__',
'__ror__',
'__rpow__',
'__rrshift__',
'__rshift__',
'__rsub__',
'__rtruediv__',
'__rxor__',
'__setattr__',
```

```
'__setitem__',
'__setstate__',
'__sizeof__',
'__str__',
'__sub__',
'__subclasshook__',
'__truediv__',
'__xor__',
'all',
'any',
'argmax',
'argmin',
'argpartition',
'argsort',
'astype',
'base',
'byteswap',
'choose',
'clip',
'compress',
'conj',
'conjugate',
'copy',
'ctypes',
'cumprod',
'cumsum',
'data',
'diagonal',
'dot',
'dtype',
'dump',
'dumps',
'fill',
'flags',
'flat',
'flatten',
'getfield',
'imag',
'item',
'itemset',
'itemsize',
'max',
'mean',
'min',
'nbytes',
'ndim',
'newbyteorder',
```

```
 'nonzero',
 'partition',
 'prod',
 'ptp',
 'put',
 'ravel',
 'real',
 'repeat',
 'reshape',
 'resize',
 'round',
 'searchsorted',
 'setfield',
 'setflags',
 'shape',
 'size',
 'sort',
 'squeeze',
 'std',
 'strides',
 'sum',
 'swapaxes',
 'take',
 'tobytes',
 'tofile',
 'tolist',
 'tostring',
 'trace',
 'transpose',
 'var',
 'view']
```

Além disso, as operações com as matrizes do *numpy* são muito mais rápidas do que as operações com as listas do Python.

Podemos calcular o produto escalar entre dois vetores usando *np.dot* .

[320]:
```
x = np.array([1.0, 1.0, 0.0])

y = np.array([0.0, 2.0, 0.0])

# Os dois comandos abaixo fazem a mesma coisa.
z = np.dot(x, y)
zz = x.dot(y)

print(z)
print(zz)
```

```
2.0
2.0
```

Podemos calcular o produto vetorial entre esses vetores usando o *np.cross* .

[30]:
```
w = np.cross(x, y)

print(w)
```

```
[0. 0. 2.]
```

Podemos pegar o valor máximo (ou o mínimo) de um vetor ou matriz.

[321]:
```
a = np.array([7.0, -3.0, 29.0])

maximo = np.max(a)

minimo = np.min(a)

print(f'O valor máximo de a é {maximo} .')

print(f'O valor mínimo de a é {minimo} .')
```

```
O valor máximo de a é 29.0 .
O valor mínimo de a é -3.0 .
```

E podemos também colocar os elementos do vetor em ordem crescente.

[323]:
```
b = np.sort(a)

print(b)
```

```
[-3.  7. 29.]
```

Dentro do *numpy* temos a biblioteca *linalg*, de álgebra linear.

Essa biblioteca tem muitas funções importantes, vamos dar uma olhada.

[324]:
```
dir(np.linalg)
```

[324]:
```
['LinAlgError',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 '_umath_linalg',
```

```
 'cholesky',
 'cond',
 'det',
 'eig',
 'eigh',
 'eigvals',
 'eigvalsh',
 'inv',
 'lapack_lite',
 'linalg',
 'lstsq',
 'matrix_power',
 'matrix_rank',
 'multi_dot',
 'norm',
 'pinv',
 'qr',
 'slogdet',
 'solve',
 'svd',
 'tensorinv',
 'tensorsolve',
 'test']
```

Considere a matriz A 4x4 abaixo.

[325]:
```
A = np.array([[5.0, 1.0, 0.0, 2.0],
              [1.0, 7.0, 2.0, 1.0],
              [0.0, 2.0, 10.0, 3.0],
              [1.0, -1.0, 3.0, 8.0]])

print(A)
```

```
[[ 5.  1.  0.  2.]
 [ 1.  7.  2.  1.]
 [ 0.  2. 10.  3.]
 [ 1. -1.  3.  8.]]
```

Podemos encontar o seu determinante.

[326]:
```
determinante = np.linalg.det(A)

print(determinante)
```

```
2143.999999999998
```

Note que pra usar o determinante nós escrevemos *np* PONTO *linalg* PONTO *det* .

Isso porque a *det* é uma função da biblioteca *linalg* que por sua vez é uma biblioteca da *numpy*.

Podemos calcular os autovalores e autovetores da matriz A.

[327]:
```
autovalores, autovetores = np.linalg.eig(A)

print('Autovalores\n')
for autovalor in autovalores:
    print(autovalor, '\n')

print('\nAutovetores\n')
for autovetor in autovetores:
    print(autovetor, '\n')
```

```
Autovalores

(12.712074908877456+0j)

(3.56409320416069+0j)

(6.861915943480913+0.4854841334440346j)

(6.861915943480913-0.4854841334440346j)


Autovetores

[0.16766911+0.j         0.80409104+0.j         0.46902609-0.26509053j
 0.46902609+0.26509053j]

[ 0.38412234+0.j         -0.27512405+0.j         -0.21654879-0.26587157j
 -0.21654879+0.26587157j]

[ 0.78599463+0.j          0.29047371+0.j         -0.45965716+0.09833608j
 -0.45965716-0.09833608j]

[ 0.45447721+0.j -0.43973787+0.j  0.60926659+0.j  0.60926659-0.j]
```

Podemos calcular a inversa da matriz A.

[328]:
```
A_inversa = np.linalg.inv(A)

print(A_inversa)
```

```
[[ 0.22154851 -0.04804104  0.02751866 -0.05970149]
 [-0.03125     0.15625    -0.03125     0.        ]
 [ 0.01772388 -0.04384328  0.12220149 -0.04477612]
 [-0.03824627  0.04197761 -0.05317164  0.14925373]]
```

Para verificarmos que essa é realmente a inversa, podemos fazer o produto entre as matrizes A e

A_inversa. O resultado tem que ser a matriz identidade.

[329]:
```
produto = np.matmul(A, A_inversa)

print(produto)
```

```
[[ 1.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-5.55111512e-17  1.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  1.00000000e+00  0.00000000e+00]
 [-5.55111512e-17  2.77555756e-17 -1.11022302e-16  1.00000000e+00]]
```

E agora? Deu certo? E esses números que aparecem com e-17?

Isso para o Python é zero. O Python tem uma precisão de 16 casas decimais.

Então pra ele esses valores pequenos (estamos falando de 0.00000000000000001) são iguais a zero.

Para deixar o print mais elegante, podemos usar a opção abaixo da *numpy*.

((Os valores continuam os mesmos, estamos apenas arredondando na hora de printar.))

[330]:
```
np.set_printoptions(suppress=True)

print(produto)
```

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

Vamos agora resolver o seguinte sistema de equações lineares:

$$\begin{aligned} 5x + y + 0z + 2w &= 9 \\ 1x + 7y + 2z + 1w &= 20 \\ 0x + 2y + 10z + 3w &= 27 \\ 1x - 1y + 3z + 8w &= 13 \end{aligned} \tag{1}$$

Queremos encontrar os valores de x, y, z e w que satisfazem essas 4 equações simultaneamente.

Esse sistema pode ser representado na forma matricial

$$Ax = b \ , \tag{2}$$

com

$$A = \begin{bmatrix} 5.0 & 1.0 & 0.0 & 2.0 \\ 1.0 & 7.0 & 2.0 & 1.0 \\ 0.0 & 2.0 & 10.0 & 3.0 \\ 1.0 & -1.0 & 3.0 & 8.0 \end{bmatrix} \tag{3}$$

$$x = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \tag{4}$$

e

$$b = \begin{bmatrix} 9 \\ 20 \\ 27 \\ 13 \end{bmatrix} \tag{5}$$

Podemos resolver esse sistema usando a *numpy*.

[333]:
```
A = np.array([[5.0, 1.0, 0.0, 2.0],
              [1.0, 7.0, 2.0, 1.0],
              [0.0, 2.0, 10.0, 3.0],
              [1.0, -1.0, 3.0, 8.0]])

b = np.array([9.0, 20.0, 27.0, 13.0])

x = np.linalg.solve(A,b)

print(x)
```

```
[1. 2. 2. 1.]
```

A solução desse sistema é

$$x = \begin{bmatrix} 1 \\ 2 \\ 2 \\ 1 \end{bmatrix} \tag{6}$$

Com o *numpy* podemos criar matrizes grandes, com um formato predefinido.

Por exemplo, se quisermos uma matriz 10x10 de zeros, podemos usar a função *zeros*.

[334]:
```
A = np.zeros((10, 10))

print(A)
```

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

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

Agora uma matriz preenchida com o número um.

[335]:
```
A = np.ones((10, 10))

print(A)
```

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

A matriz identidade.

[336]:
```
I = np.eye(10)

print(A)
```

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

Vamos somar a matriz identidade com a matriz de uns.

[338]:
```
A = np.ones((10, 10))

I = np.eye(10)

B = A + I

print(B)
```

```
[[2. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

```
[1. 2. 1. 1. 1. 1. 1. 1. 1. 1.]
[1. 1. 2. 1. 1. 1. 1. 1. 1. 1.]
[1. 1. 1. 2. 1. 1. 1. 1. 1. 1.]
[1. 1. 1. 1. 2. 1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 2. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 2. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 2. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1. 2. 1.]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 2.]]
```

Deu pra perceber como a *numpy* é poderosa?

Uma ferramenta legal é a biblioteca *numpy* chamada *random*, de **geração de números aleatórios**.

[339]:
```
dir(np.random)
```

[339]:
```
['BitGenerator',
 'Generator',
 'MT19937',
 'PCG64',
 'Philox',
 'RandomState',
 'SFC64',
 'SeedSequence',
 '__RandomState_ctor',
 '__all__',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 '_bounded_integers',
 '_common',
 '_generator',
 '_mt19937',
 '_pcg64',
 '_philox',
 '_pickle',
 '_sfc64',
 'beta',
 'binomial',
 'bit_generator',
 'bytes',
 'chisquare',
```

```
'choice',
'default_rng',
'dirichlet',
'exponential',
'f',
'gamma',
'geometric',
'get_state',
'gumbel',
'hypergeometric',
'laplace',
'logistic',
'lognormal',
'logseries',
'mtrand',
'multinomial',
'multivariate_normal',
'negative_binomial',
'noncentral_chisquare',
'noncentral_f',
'normal',
'pareto',
'permutation',
'poisson',
'power',
'rand',
'randint',
'randn',
'random',
'random_integers',
'random_sample',
'ranf',
'rayleigh',
'sample',
'seed',
'set_state',
'shuffle',
'standard_cauchy',
'standard_exponential',
'standard_gamma',
'standard_normal',
'standard_t',
'test',
'triangular',
'uniform',
'vonmises',
'wald',
```

```
 'weibull',
 'zipf']
```

Com ela podemos gerar números aleatórios com diferentes distribuições.

Queremos 5 números aleatórios entre 0 e 1.

[346]:
```
np.random.random(5)
```

[346]:
```
array([0.27323435, 0.15670734, 0.56869924, 0.2717823 , 0.43340903])
```

((Cada vez que executamos o código são gerados números diferentes. Esses números estão entre 0 e 1 e têm uma distribuição uniforme.))

Agora queremos 5 números inteiros aleatórios entre 1 e 60 (inclusive).

[351]:
```
np.random.randint(1, 61, 6)
```

[351]:
```
array([31, 31, 10, 32, 34, 24])
```

Agora em ordem crescente.

[355]:
```
np.sort(np.random.randint(1, 61, 6))
```

[355]:
```
array([15, 18, 20, 21, 33, 47])
```

Também podemos gerar números aleatórios com distribuição normal.

[356]:
```
np.random.normal()
```

[356]:
```
0.18389683490809083
```

Pra fechar aqui essa parte, vou apresentar formar de criar vetores com um intervalo igual usando o ***arange*** e o ***linspace*** .

((Isso vai ser importante na próxima seção, de plotar gráficos.))

Queremos um vetor com valores entre zero e dez e com intervalo de 0.1 entre os números.

[357]:
```
x = np.arange(0.0, 10.0, 0.1)

print(x)
```

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7
 1.8 1.9 2.  2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.  3.1 3.2 3.3 3.4 3.5
 3.6 3.7 3.8 3.9 4.  4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9 5.  5.1 5.2 5.3
 5.4 5.5 5.6 5.7 5.8 5.9 6.  6.1 6.2 6.3 6.4 6.5 6.6 6.7 6.8 6.9 7.  7.1
 7.2 7.3 7.4 7.5 7.6 7.7 7.8 7.9 8.  8.1 8.2 8.3 8.4 8.5 8.6 8.7 8.8 8.9
 9.  9.1 9.2 9.3 9.4 9.5 9.6 9.7 9.8 9.9]
```

Veja outros exemplos.

[125]:
```
x = np.arange(5.0, 10.0, 0.1)

print(x)
```

```
[5.  5.1 5.2 5.3 5.4 5.5 5.6 5.7 5.8 5.9 6.  6.1 6.2 6.3 6.4 6.5 6.6 6.7
 6.8 6.9 7.  7.1 7.2 7.3 7.4 7.5 7.6 7.7 7.8 7.9 8.  8.1 8.2 8.3 8.4 8.5
 8.6 8.7 8.8 8.9 9.  9.1 9.2 9.3 9.4 9.5 9.6 9.7 9.8 9.9]
```

[126]:
```
x = np.arange(-3.0, 20.0, 0.3)

print(x)
```

```
[-3.  -2.7 -2.4 -2.1 -1.8 -1.5 -1.2 -0.9 -0.6 -0.3 -0.   0.3  0.6  0.9
  1.2  1.5  1.8  2.1  2.4  2.7  3.   3.3  3.6  3.9  4.2  4.5  4.8  5.1
  5.4  5.7  6.   6.3  6.6  6.9  7.2  7.5  7.8  8.1  8.4  8.7  9.   9.3
  9.6  9.9 10.2 10.5 10.8 11.1 11.4 11.7 12.  12.3 12.6 12.9 13.2 13.5
 13.8 14.1 14.4 14.7 15.  15.3 15.6 15.9 16.2 16.5 16.8 17.1 17.4 17.7
 18.  18.3 18.6 18.9 19.2 19.5 19.8]
```

O *linspace* é muito parecido. A diferença é que não falamos o intervalo, e sim o número de pontos.

Por exemplo, quero um vetor com 11 números entre 0 e 10, igualmente espaçados.

[359]:
```
x = np.linspace(0.0, 10.0, 11)

print(x)
```

```
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

Veja mais alguns exemplos.

[360]:
```
x = np.linspace(0.0, 0.1, 41)

print(x)
```

```
[0.     0.0025 0.005  0.0075 0.01   0.0125 0.015  0.0175 0.02   0.0225
 0.025  0.0275 0.03   0.0325 0.035  0.0375 0.04   0.0425 0.045  0.0475
 0.05   0.0525 0.055  0.0575 0.06   0.0625 0.065  0.0675 0.07   0.0725
 0.075  0.0775 0.08   0.0825 0.085  0.0875 0.09   0.0925 0.095  0.0975
 0.1   ]
```

[361]:
```
x = np.linspace(-5.0, -3.0, 101)

print(x)
```

```
[-5.   -4.98 -4.96 -4.94 -4.92 -4.9  -4.88 -4.86 -4.84 -4.82 -4.8  -4.78
 -4.76 -4.74 -4.72 -4.7  -4.68 -4.66 -4.64 -4.62 -4.6  -4.58 -4.56 -4.54
 -4.52 -4.5  -4.48 -4.46 -4.44 -4.42 -4.4  -4.38 -4.36 -4.34 -4.32 -4.3
 -4.28 -4.26 -4.24 -4.22 -4.2  -4.18 -4.16 -4.14 -4.12 -4.1  -4.08 -4.06
 -4.04 -4.02 -4.   -3.98 -3.96 -3.94 -3.92 -3.9  -3.88 -3.86 -3.84 -3.82
```

```
-3.8  -3.78 -3.76 -3.74 -3.72 -3.7  -3.68 -3.66 -3.64 -3.62 -3.6  -3.58
-3.56 -3.54 -3.52 -3.5  -3.48 -3.46 -3.44 -3.42 -3.4  -3.38 -3.36 -3.34
-3.32 -3.3  -3.28 -3.26 -3.24 -3.22 -3.2  -3.18 -3.16 -3.14 -3.12 -3.1
-3.08 -3.06 -3.04 -3.02 -3.  ]
```

[363]:
```
x = np.linspace(1.0, 15.0, 15)

print(x)
```

```
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15.]
```

A *numpy* permite fazer operações diretamente com esses vetores.

Podemos criar um vetor com os elementos sendo o quadrado dos elementos de um outro vetor.

[364]:
```
y = x**2.0

print(y)
```

```
[  1.   4.   9.  16.  25.  36.  49.  64.  81. 100. 121. 144. 169. 196.
 225.]
```

Ou o seno.

[366]:
```
y = np.sin(x)

print(y)
```

```
[ 0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427 -0.2794155
  0.6569866   0.98935825  0.41211849 -0.54402111 -0.99999021 -0.53657292
  0.42016704  0.99060736  0.65028784]
```

Ou o logaritmo.

[139]:
```
y = np.log(x)

print(y)
```

```
[0.         0.69314718 1.09861229 1.38629436 1.60943791 1.79175947
 1.94591015 2.07944154 2.19722458 2.30258509 2.39789527 2.48490665
 2.56494936 2.63905733 2.7080502 ]
```

Vou fechar essa parte da *numpy* por aqui.

Como eu falei, a *numpy* tem muito mais ferramentas do que essas poucas que foram mostradas aqui.

Com certeza uma delas vai resolver o seu problema. É só procurar com calma.

Vamos plotar gráficos agora.

6.4 Plotando Gráficos com o Matplotlib

Primeiro vamos importar a bilioteca *matplotlib*.

[2]:
```
import matplotlib.pyplot as plt
```

Note que o nome dessa biblioteca é muito grande.

Por isso demos o apelido *plt* a ela.

Para podermos conhecer essa bilioteca, vamos ver as funções que ela possui com o *dir* e vamos dar um *help* também.

[368]:
```
#dir(plt)
#help(plt)
help(plt.plot)
#help(plt.subplots)
```

```
Help on function plot in module matplotlib.pyplot:

plot(*args, scalex=True, scaley=True, data=None, **kwargs)
    Plot y versus x as lines and/or markers.

    Call signatures::

        plot([x], y, [fmt], *, data=None, **kwargs)
        plot([x], y, [fmt], [x2], y2, [fmt2], …, **kwargs)

    The coordinates of the points or line nodes are given by *x*, *y*.

    The optional parameter *fmt* is a convenient way for defining basic
    formatting like color, marker and linestyle. It's a shortcut string
    notation described in the *Notes* section below.

    >>> plot(x, y)        # plot x and y using default line style and color
    >>> plot(x, y, 'bo')  # plot x and y using blue circle markers
    >>> plot(y)           # plot y using x as index array 0..N-1
    >>> plot(y, 'r+')     # ditto, but with red plusses

    You can use `.Line2D` properties as keyword arguments for more
    control on the appearance. Line properties and *fmt* can be mixed.
    The following two calls yield identical results:

    >>> plot(x, y, 'go--', linewidth=2, markersize=12)
    >>> plot(x, y, color='green', marker='o', linestyle='dashed',
    …      linewidth=2, markersize=12)

    When conflicting with *fmt*, keyword arguments take precedence.
```

Plotting labelled data

There's a convenient way for plotting objects with labelled data (i.e.
data that can be accessed by index ``obj['y']``). Instead of giving
the data in *x* and *y*, you can provide the object in the *data*
parameter and just give the labels for *x* and *y*::

>>> plot('xlabel', 'ylabel', data=obj)

All indexable objects are supported. This could e.g. be a `dict`, a
`pandas.DataFrame` or a structured numpy array.

Plotting multiple sets of data

There are various ways to plot multiple sets of data.

- The most straight forward way is just to call `plot` multiple times.
  Example:

  >>> plot(x1, y1, 'bo')
  >>> plot(x2, y2, 'go')

- Alternatively, if your data is already a 2d array, you can pass it
  directly to *x*, *y*. A separate data set will be drawn for every
  column.

  Example: an array ``a`` where the first column represents the *x*
  values and the other columns are the *y* columns::

  >>> plot(a[0], a[1:])

- The third way is to specify multiple sets of *[x]*, *y*, *[fmt]*
  groups::

  >>> plot(x1, y1, 'g^', x2, y2, 'g-')

  In this case, any additional keyword argument applies to all
  datasets. Also this syntax cannot be combined with the *data*
  parameter.

By default, each line is assigned a different style specified by a
'style cycle'. The *fmt* and line property parameters are only
necessary if you want explicit deviations from these defaults.
Alternatively, you can also change the style cycle using
:rc:`axes.prop_cycle`.

```
Parameters
----------
x, y : array-like or scalar
    The horizontal / vertical coordinates of the data points.
    *x* values are optional and default to ``range(len(y))``.

    Commonly, these parameters are 1D arrays.

    They can also be scalars, or two-dimensional (in that case, the
    columns represent separate data sets).

    These arguments cannot be passed as keywords.

fmt : str, optional
    A format string, e.g. 'ro' for red circles. See the *Notes*
    section for a full description of the format strings.

    Format strings are just an abbreviation for quickly setting
    basic line properties. All of these and more can also be
    controlled by keyword arguments.

    This argument cannot be passed as keyword.

data : indexable object, optional
    An object with labelled data. If given, provide the label names to
    plot in *x* and *y*.

    .. note::
        Technically there's a slight ambiguity in calls where the
        second label is a valid *fmt*. ``plot('n', 'o', data=obj)``
        could be ``plt(x, y)`` or ``plt(y, fmt)``. In such cases,
        the former interpretation is chosen, but a warning is issued.
        You may suppress the warning by adding an empty format string
        ``plot('n', 'o', '', data=obj)``.

Returns
-------
list of `.Line2D`
    A list of lines representing the plotted data.

Other Parameters
----------------
scalex, scaley : bool, default: True
    These parameters determine if the view limits are adapted to the
    data limits. The values are passed on to `autoscale_view`.

**kwargs : `.Line2D` properties, optional
    *kwargs* are used to specify properties like a line label (for
```

```
        auto legends), linewidth, antialiasing, marker face color.
        Example::

        >>> plot([1, 2, 3], [1, 2, 3], 'go-', label='line 1', linewidth=2)
        >>> plot([1, 2, 3], [1, 4, 9], 'rs', label='line 2')

        If you make multiple lines with one plot call, the kwargs
        apply to all those lines.

        Here is a list of available `.Line2D` properties:

        Properties:
        agg_filter: a filter function, which takes a (m, n, 3) float array and a
dpi value, and returns a (m, n, 3) array
        alpha: float or None
        animated: bool
        antialiased or aa: bool
        clip_box: `.Bbox`
        clip_on: bool
        clip_path: Patch or (Path, Transform) or None
        color or c: color
        contains: unknown
        dash_capstyle: {'butt', 'round', 'projecting'}
        dash_joinstyle: {'miter', 'round', 'bevel'}
        dashes: sequence of floats (on/off ink in points) or (None, None)
        data: (2, N) array or two 1D arrays
        drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-
post'}, default: 'default'
        figure: `.Figure`
        fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
        gid: str
        in_layout: bool
        label: object
        linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), …}
        linewidth or lw: float
        marker: marker style string, `~.path.Path` or `~.markers.MarkerStyle`
        markeredgecolor or mec: color
        markeredgewidth or mew: float
        markerfacecolor or mfc: color
        markerfacecoloralt or mfcalt: color
        markersize or ms: float
        markevery: None or int or (int, int) or slice or List[int] or float or
(float, float) or List[bool]
        path_effects: `.AbstractPathEffect`
        picker: unknown
        pickradius: float
        rasterized: bool or None
        sketch_params: (scale: float, length: float, randomness: float)
```

```
    snap: bool or None
    solid_capstyle: {'butt', 'round', 'projecting'}
    solid_joinstyle: {'miter', 'round', 'bevel'}
    transform: `matplotlib.transforms.Transform`
    url: str
    visible: bool
    xdata: 1D array
    ydata: 1D array
    zorder: float

See Also
--------
scatter : XY scatter plot with markers of varying size and/or color (
    sometimes also called bubble chart).

Notes
-----
**Format Strings**

A format string consists of a part for color, marker and line::

    fmt = '[marker][line][color]'

Each of them is optional. If not provided, the value from the style
cycle is used. Exception: If ``line`` is given, but no ``marker``,
the data will be a line without markers.

Other combinations such as ``[color][marker][line]`` are also
supported, but note that their parsing may be ambiguous.

**Markers**

=============    ===============================
character        description
=============    ===============================
``'.'``          point marker
``','``          pixel marker
``'o'``          circle marker
``'v'``          triangle_down marker
``'^'``          triangle_up marker
``'<'``          triangle_left marker
``'>'``          triangle_right marker
``'1'``          tri_down marker
``'2'``          tri_up marker
``'3'``          tri_left marker
``'4'``          tri_right marker
``'s'``          square marker
``'p'``          pentagon marker
```

```
``'*'``          star marker
``'h'``          hexagon1 marker
``'H'``          hexagon2 marker
``'+'``          plus marker
``'x'``          x marker
``'D'``          diamond marker
``'d'``          thin_diamond marker
``'|'``          vline marker
``'_'``          hline marker
=============    ==============================

**Line Styles**

=============    ==============================
character        description
=============    ==============================
``'-'``          solid line style
``'--'``         dashed line style
``'-.'``         dash-dot line style
``':'``          dotted line style
=============    ==============================

Example format strings::

    'b'    # blue markers with default shape
    'or'   # red circles
    '-g'   # green solid line
    '--'   # dashed line with default color
    '^k:'  # black triangle_up markers connected by a dotted line

**Colors**

The supported color abbreviations are the single letter codes

=============    ==============================
character        color
=============    ==============================
``'b'``          blue
``'g'``          green
``'r'``          red
``'c'``          cyan
``'m'``          magenta
``'y'``          yellow
``'k'``          black
``'w'``          white
=============    ==============================

and the ``'CN'`` colors that index into the default property cycle.
```

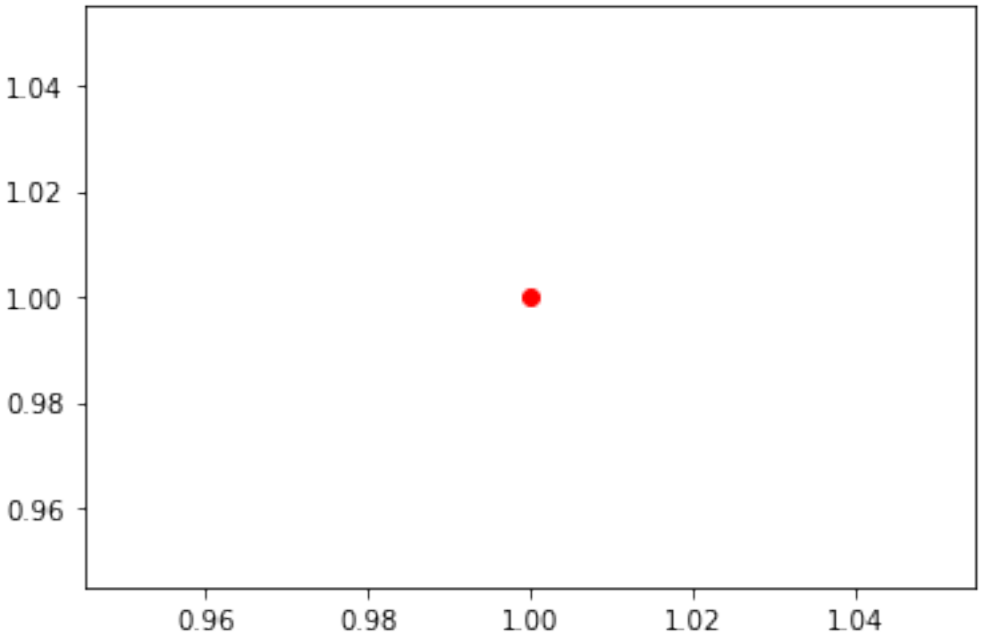
```
If the color is the only part of the format string, you can
additionally use any  `matplotlib.colors` spec, e.g. full names
(``'green'``) or hex strings (``'#008000'``).
```

Vamos começar com um gráfico simples.

Primeiro nós vamos criar dois objetos: uma figura (*fig*), que representa a figura como um todo, e um objeto plot (*ax*), representado pelos eixos, que é onde serão acrescentados os dados.

[369]:
```
fig, ax = plt.subplots()    # Criando os objetos.

ax.plot(1.0, 1.0, 'ro')    # Plotando o ponto 1.0, 1.0 como um círculo vermelho.
```

[369]:
```
[<matplotlib.lines.Line2D at 0x7fa3437b8d30>]
```



No código acima estamos criando os objetos *fig* e *ax* e estamos adicionando os pontos que queremos plotar ao objeto *ax*.

Estamos plotando o ponto (1,1) usando um círculo vermelho.

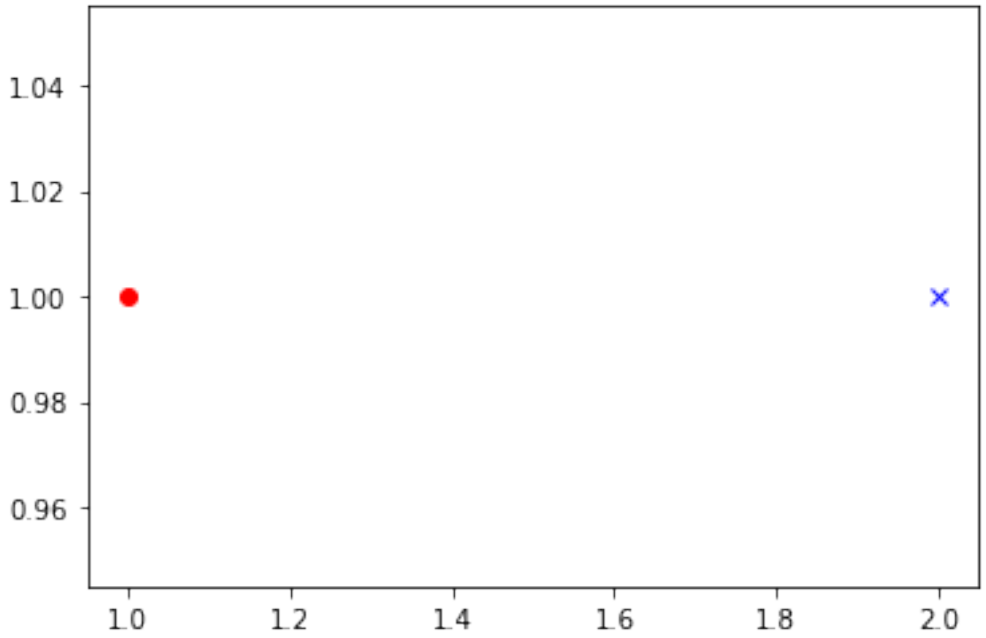
((Ver o final do help(plt.plot) para mais opções de cores e formas.))

Vamos adicionar mais um ponto.

[370]:
```
fig, ax = plt.subplots()

ax.plot(1.0, 1.0, 'ro')
```

```
ax.plot(2.0, 1.0, 'xb')
```

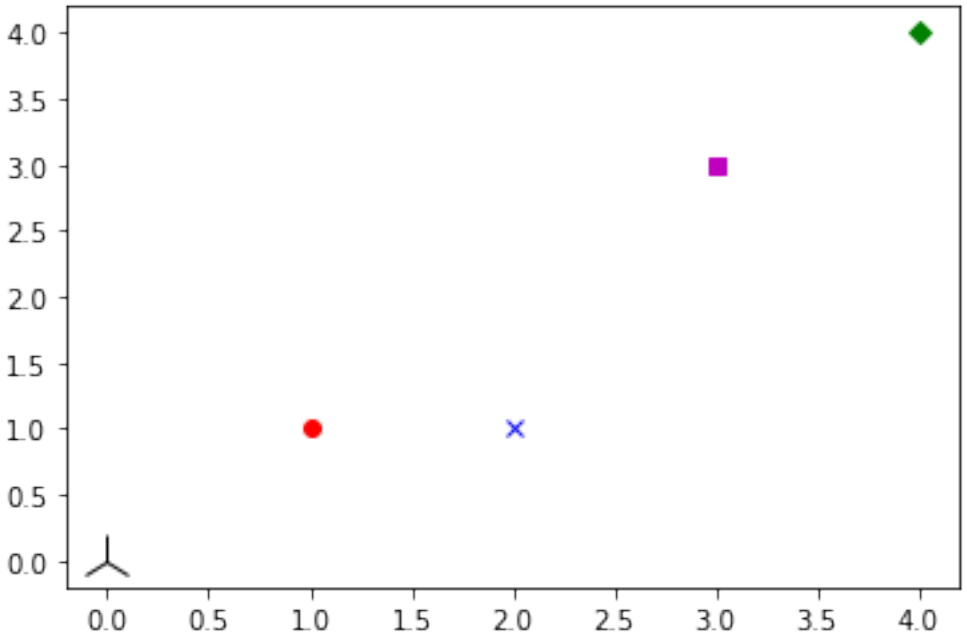
[370]: [<matplotlib.lines.Line2D at 0x7fa34369a280>]



Mais pontos.

[371]:
```
fig, ax = plt.subplots()

ax.plot(1.0, 1.0, 'ro')

ax.plot(2.0, 1.0, 'xb')

ax.plot(0.0, 0.0, 'k2', markersize = 20)

ax.plot(4.0, 4.0, 'Dg')

ax.plot(3.0, 3.0, 'ms')
```

[371]: [<matplotlib.lines.Line2D at 0x7fa3436870d0>]



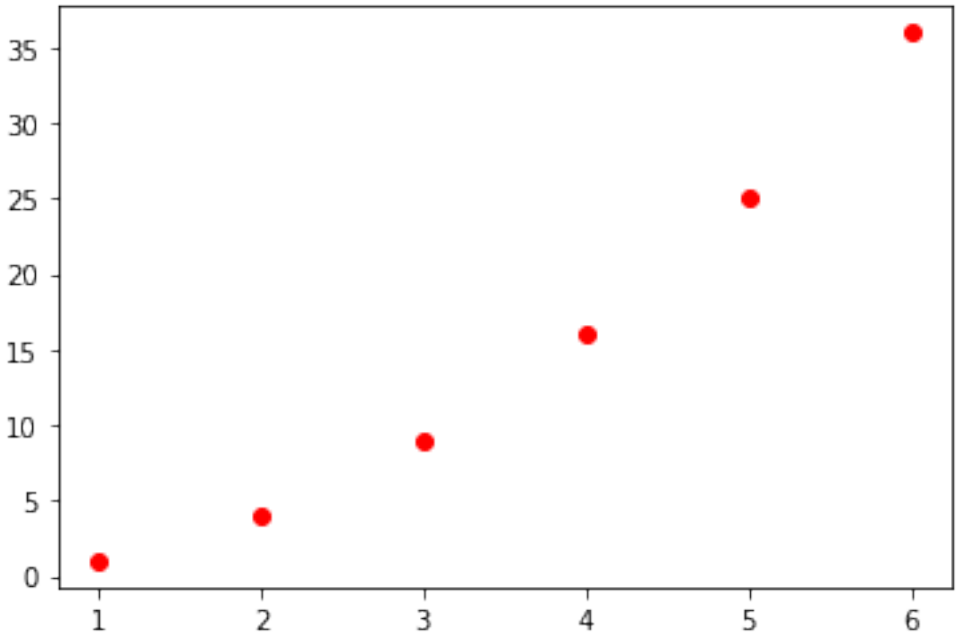
E se quisermos plotar 100 pontos? Ou 1000?

A função plot aceita listas como entrada.

((Atenção: as duas listas devem ter o mesmo tamanho.))

[372]:
```
fig, ax = plt.subplots()

x = [1, 2, 3, 4, 5, 6]

y = [1, 4, 9, 16, 25, 36]

ax.plot(x, y, 'ro')
```

[372]:
```
[<matplotlib.lines.Line2D at 0x7fa3435e3c40>]
```

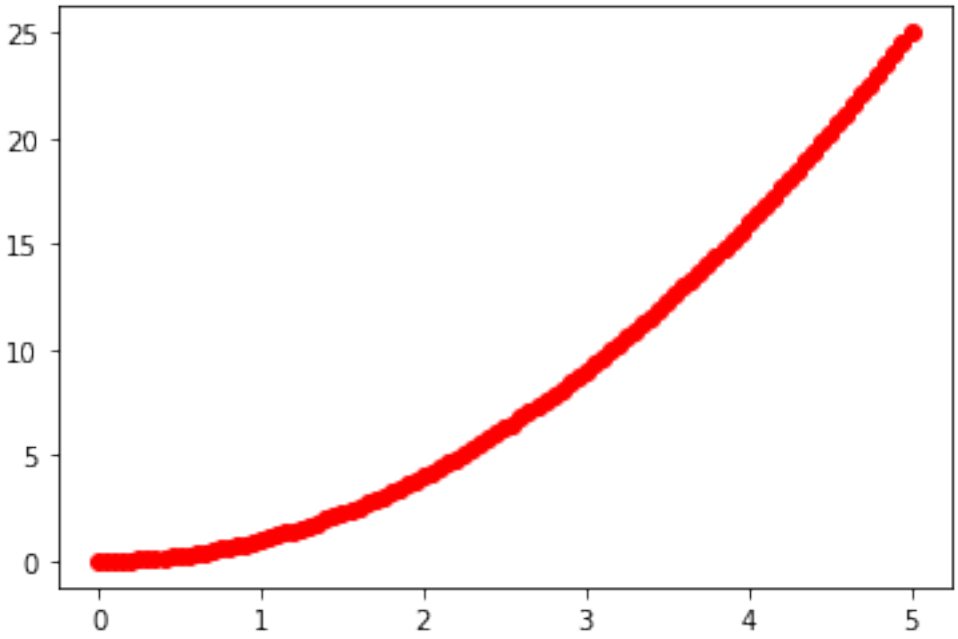


E é aí que podemos usar a *numpy*.

Vamos plotar o quadrado de x.

[373]:
```
fig, ax = plt.subplots()

x = np.linspace(0.0, 5.0, 101)

ax.plot(x, x**2.0, 'ro')
```

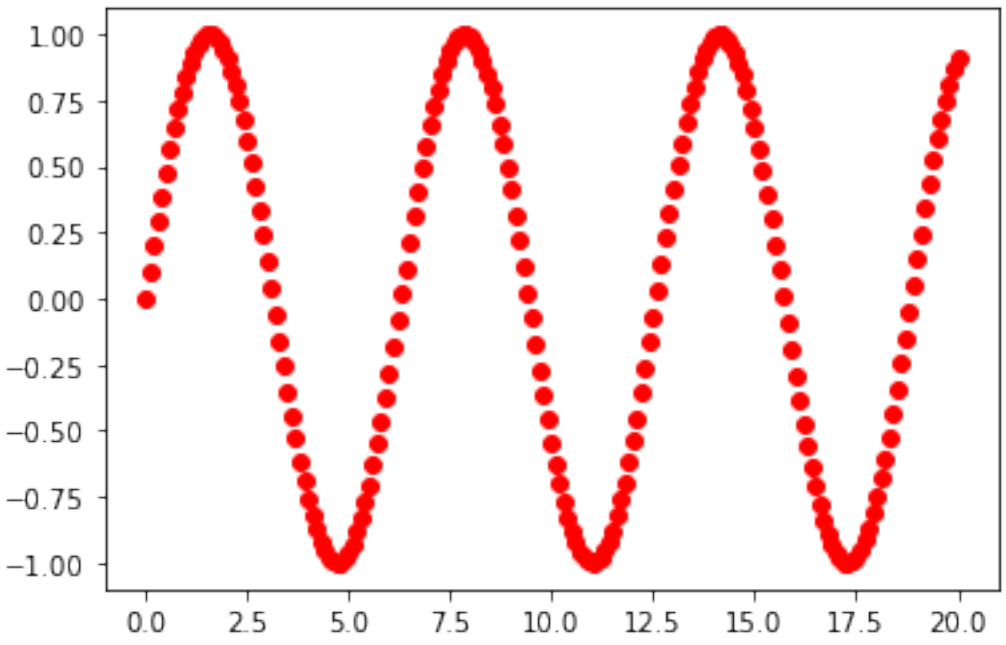
[373]:
```
[<matplotlib.lines.Line2D at 0x7fa3435beca0>]
```



Agora o seno.

[374]:
```
fig, ax = plt.subplots()

x = np.linspace(0.0, 20.0, 201)

ax.plot(x, np.sin(x), 'ro')
```

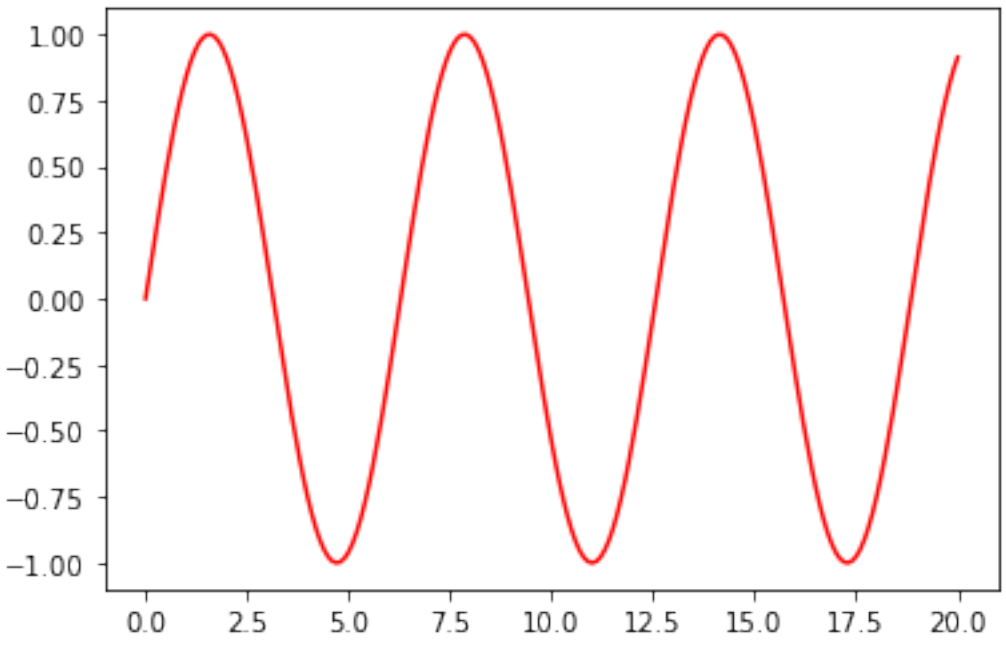
[374]:
```
[<matplotlib.lines.Line2D at 0x7fa34351c400>]
```



E podemos usar linha em vez de ponto.

[375]:
```
fig, ax = plt.subplots()

x = np.linspace(0.0, 20.0, 201)

ax.plot(x, np.sin(x), 'r-')
```

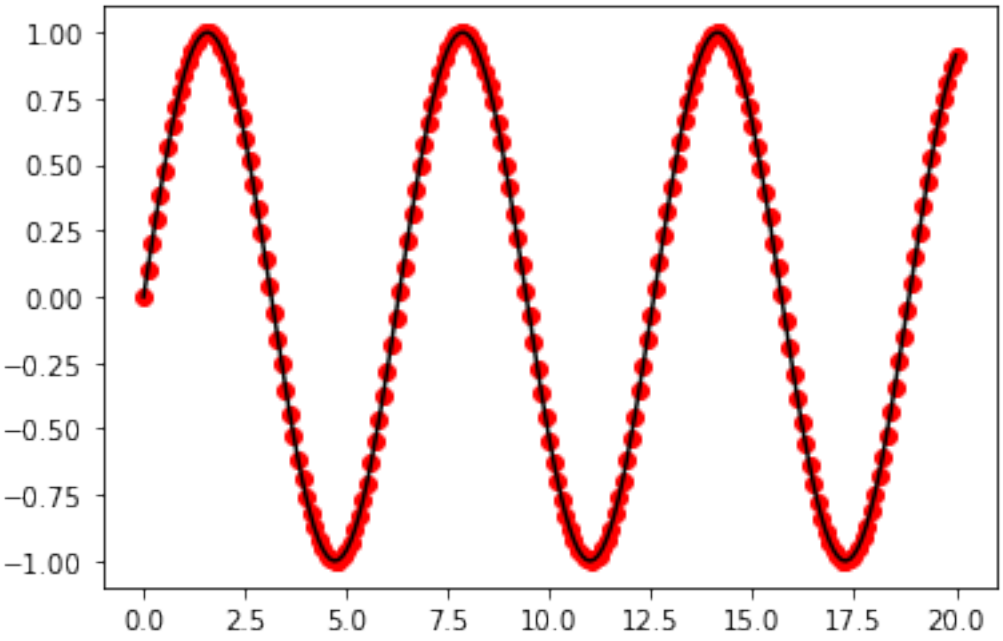
[375]:
```
[<matplotlib.lines.Line2D at 0x7fa3434fab20>]
```



Ou os dois.

[376]:
```
fig, ax = plt.subplots()

x = np.linspace(0.0, 20.0, 201)

ax.plot(x, np.sin(x), 'ro')

ax.plot(x, np.sin(x), 'k-')
```

[376]:
```
[<matplotlib.lines.Line2D at 0x7fa343464880>]
```



Agora vamos plotar vários funções de x e vamos adicionar várias opções.

No final, vamos salvar a figura externamente. Isso é importante quando você quer usar a figura em um artigo ou relatório.

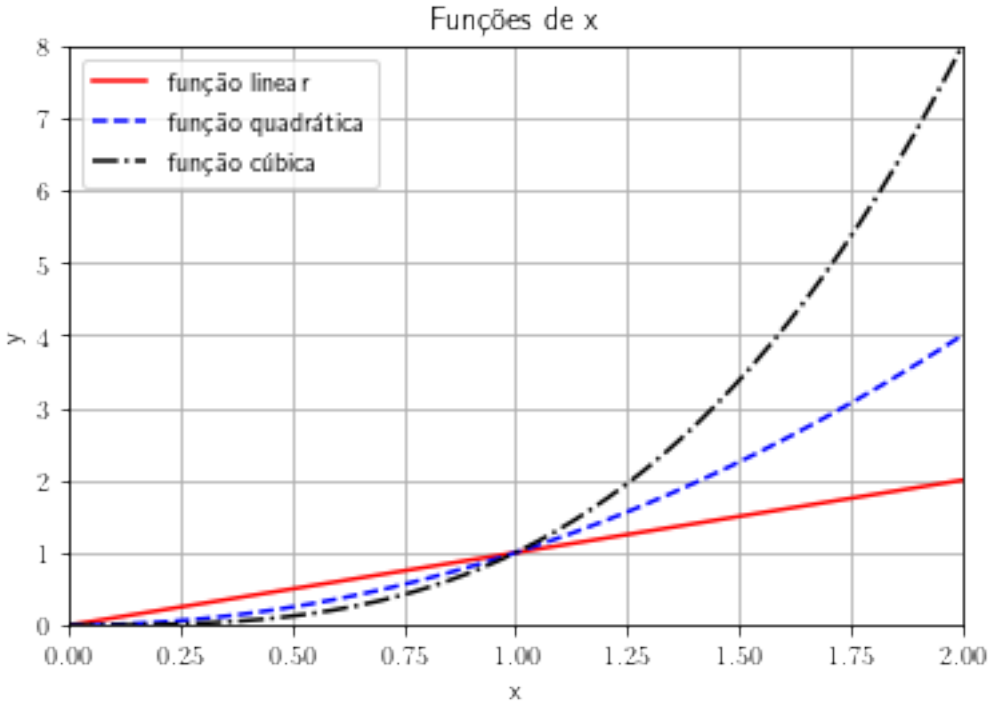
Podemos exportar como pdf, png, ps, eps e jpg (e outros formatos).

A figura vai aparecer na pasta onde está o seu código.

((Para exportar com o Google Colab eu ainda não sei muito bem como fazer, pois a figura vai para o seu Drive.))

[12]:
```
fig, ax = plt.subplots()

x = np.linspace(0.0, 2.0, 201)

ax.plot(x, x, 'r-', label='função linear')    # Label dá o nome que vai␣
↪aparecer na legenda.
ax.plot(x, x**2.0, 'b--', label='função quadrática')
ax.plot(x, x**3.0, 'k-.', label='função cúbica')

ax.set_title("Funções de x")    # Título do gráfico.
ax.set_xlabel("x")    # Nome do eixo x.
ax.set_ylabel("y")    # Nome do eixo y.
ax.set_xlim(0.0, 2.0)    # Intervalo no eixo x.
ax.set_ylim(0.0, 8.0)    # Intervalo no eixo y.
```

```
ax.grid("on")     # Mostrando o grid, que são esses quadrados que aparecem no␣
 ↪gráfico.

ax.legend()     # Criando a legenda.

# Exportando a figura em pdf. O bbox_inches tira todo o espaço branco em volta
# da figura e o pad_inches ajusta esse recorte.
fig.savefig("figura1.pdf", format = "pdf", bbox_inches = 'tight', pad_inches =␣
 ↪0.2)
fig.savefig("figura1.png", format = "png", bbox_inches = 'tight', pad_inches =␣
 ↪0.2)

# Se você está usando o Google Colab, tem que rodar os comandos
# abaixo para fazer o download da figura.
# from google.colab import files
# files.download('figura1.pdf')
# files.download('figura1.png')
```

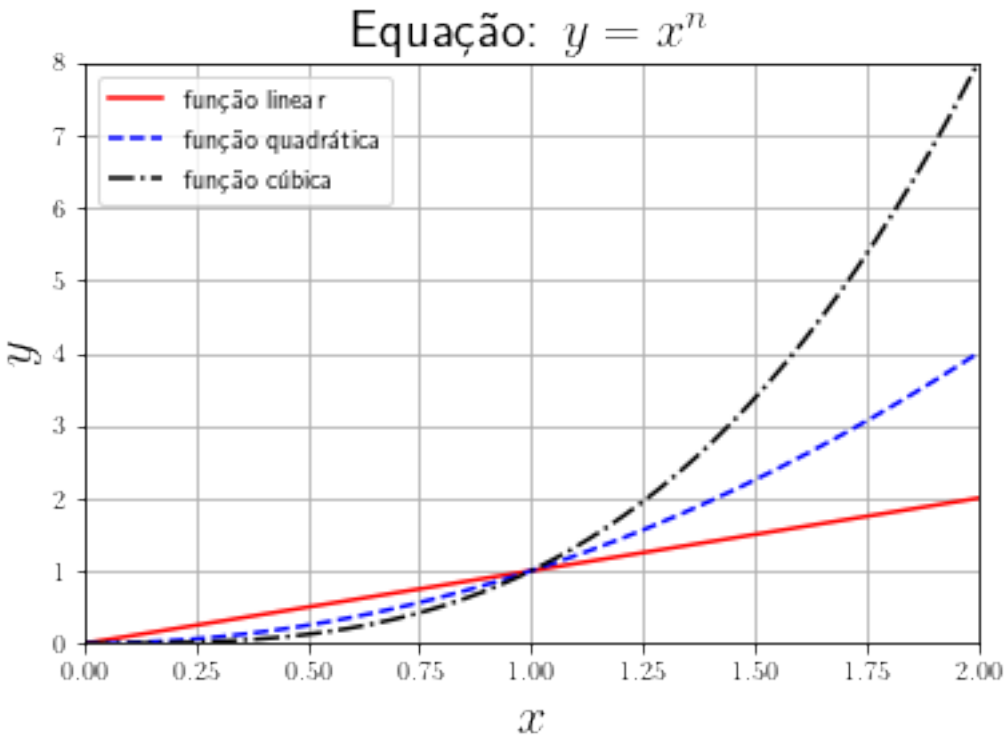


Pra quem usar *Latex*, podemos deixar todas as fontes de acordo com o *Latex* também.

Além disso, podemos escrever equações na figura.

[4]:
```
# Atenção: para dar certo no Google Colab, precisei rodar
# a linha abaixo (leva um tempinho).
```

```
#!apt install texlive-fonts-recommended texlive-fonts-extra cm-super dvipng

plt.rcParams['text.usetex'] = True    # Esse comando deixa as fontes no padrão␣
↪Latex.

fig, ax = plt.subplots()

x = np.linspace(0.0, 2.0, 201)

ax.plot(x, x, 'r-', label='função linear')
ax.plot(x, x**2.0, 'b--', label='função quadrática')
ax.plot(x, x**3.0, 'k-.', label='função cúbica')

ax.set_title(r"Equação: $y = x^n$", fontsize=20)
ax.set_xlabel("$x$", fontsize=20)
ax.set_ylabel("$y$", fontsize=20)
ax.set_xlim(0.0, 2.0)
ax.set_ylim(0.0, 8.0)

ax.grid("on")

ax.legend()

fig.savefig("figura1.pdf", format = "pdf", bbox_inches = 'tight', pad_inches =␣
↪0.2)
fig.savefig("figura1.png", format = "png", bbox_inches = 'tight', pad_inches =␣
↪0.2)
```



Essa é a configuração básica, temos várias outras opções de configuração e de tipos de gráficos.

Para mais informações sobre configuração, dê uma olhada nos *helps* abaixo.

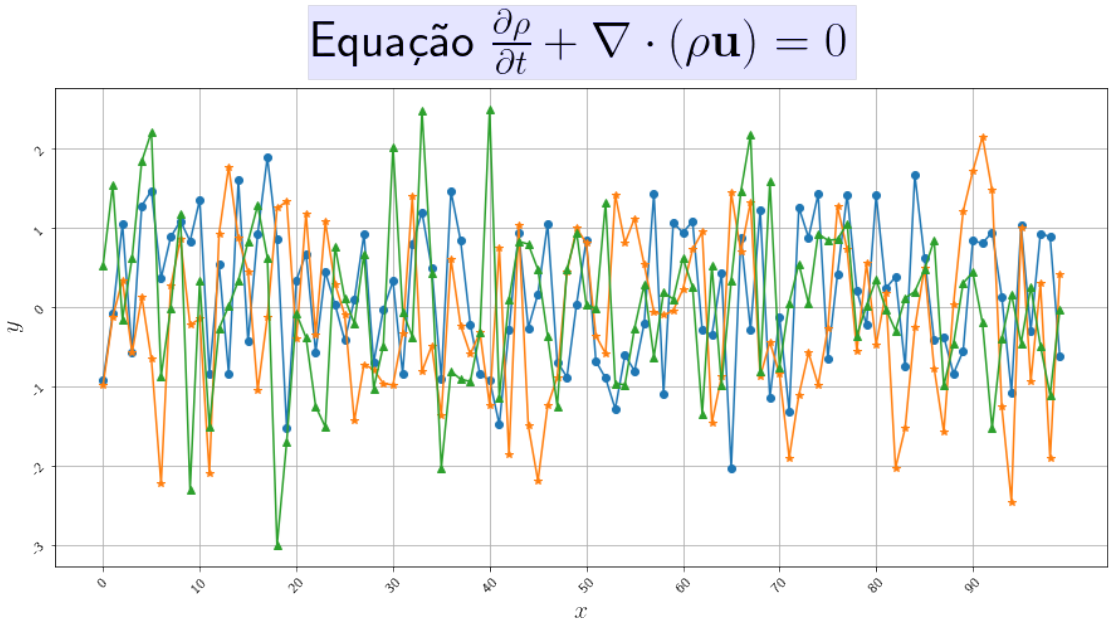
[383]:
```
#help(axs.text)
#help(axs.set_xlabel)
#help(axs.set_title)
```

Veja os exemplos abaixo. Faça experimentos com eles. Mude os parâmetros para entender como eles funcionam.

Exemplo 1:

[5]:
```
plt.rcParams['text.usetex'] = True

fig, axs = plt.subplots(figsize = (15, 7))

data1 = np.random.normal(0, 1, 100)
data2 = np.random.normal(0, 1, 100)
data3 = np.random.normal(0, 1, 100)

x_ax = np.arange(0, 100, 10)
y_ax = np.arange(-3, 3, 1)
```

```
axs.plot(data1, marker = "o")
axs.plot(data2, marker = "*")
axs.plot(data3, marker = "^")

axs.set_xticks(x_ax)
axs.set_xticklabels(labels = x_ax, rotation = 45, fontsize = 12)
axs.set_yticks(y_ax)
axs.set_yticklabels(labels = y_ax, rotation = 45, fontsize = 12)

axs.set_xlabel(r"$x$", fontsize = 20)
axs.set_ylabel(r"$y$", fontsize = 20)

axs.set_title(r"Equação $\frac{\partial \rho}{\partial t} + "+
              r"\nabla \cdot (\rho\mathbf{u}) = 0$",
              fontsize = 40, pad = 25,
              bbox = dict(facecolor = 'blue', alpha = 0.1),
              animated = True)

axs.grid("on")

fig.savefig("figura1.pdf", format = "pdf", bbox_inches = 'tight', pad_inches =␣
↪0.2)
```

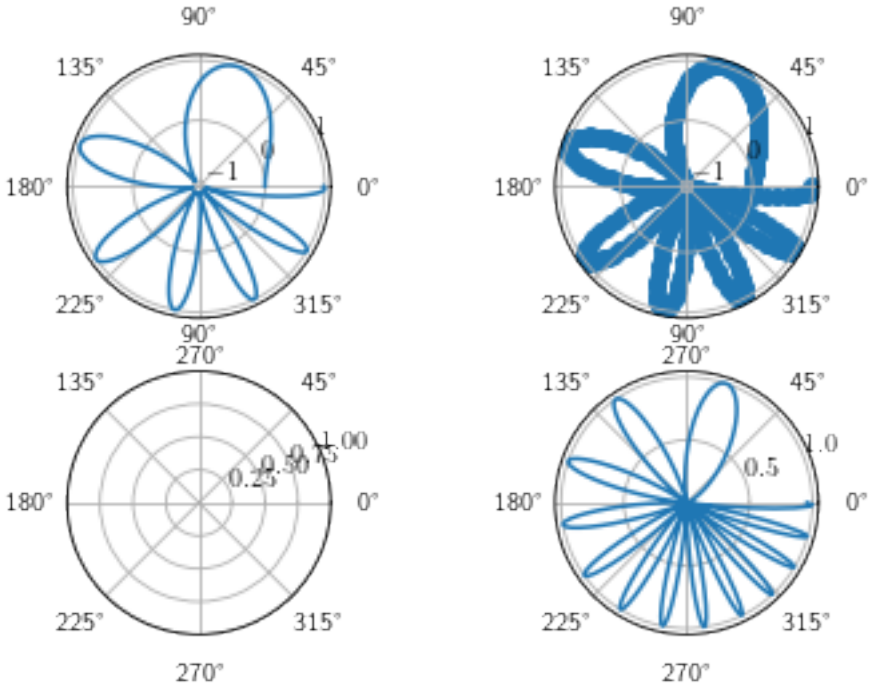


Exemplo 2: coordenadas polares.

[386]:
```
x = np.linspace(0.0, 2.0*np.pi, 400)
y = np.sin(x**2)
```

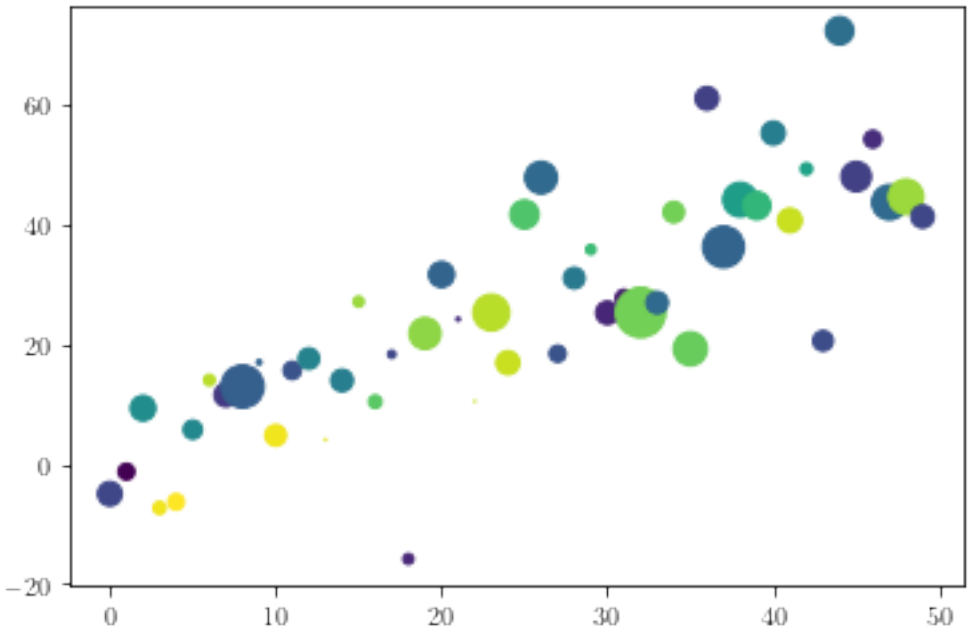
```
# Criando quatro subplots.
f, axs = plt.subplots(2, 2, subplot_kw = dict(polar = True))

axs[0,0].plot(x, y)
axs[0,1].scatter(x, y)
axs[1,1].plot(x, y**2.0)
```

[386]: [<matplotlib.lines.Line2D at 0x7fa343435c70>]



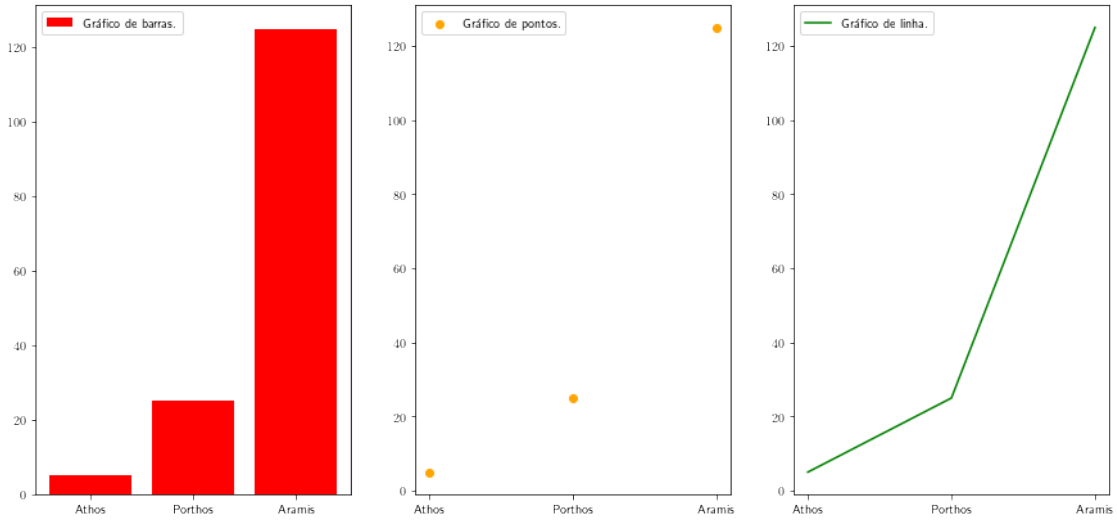
Exemplo 3: usando números aleatórios para criar os pontos, as cores e os tamanhos dos pontos.

[387]:
```
data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}

data['b'] = data['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

fig, ax = plt.subplots()

ax = plt.scatter('a', 'b', c = 'c', s = 'd', data = data)
```



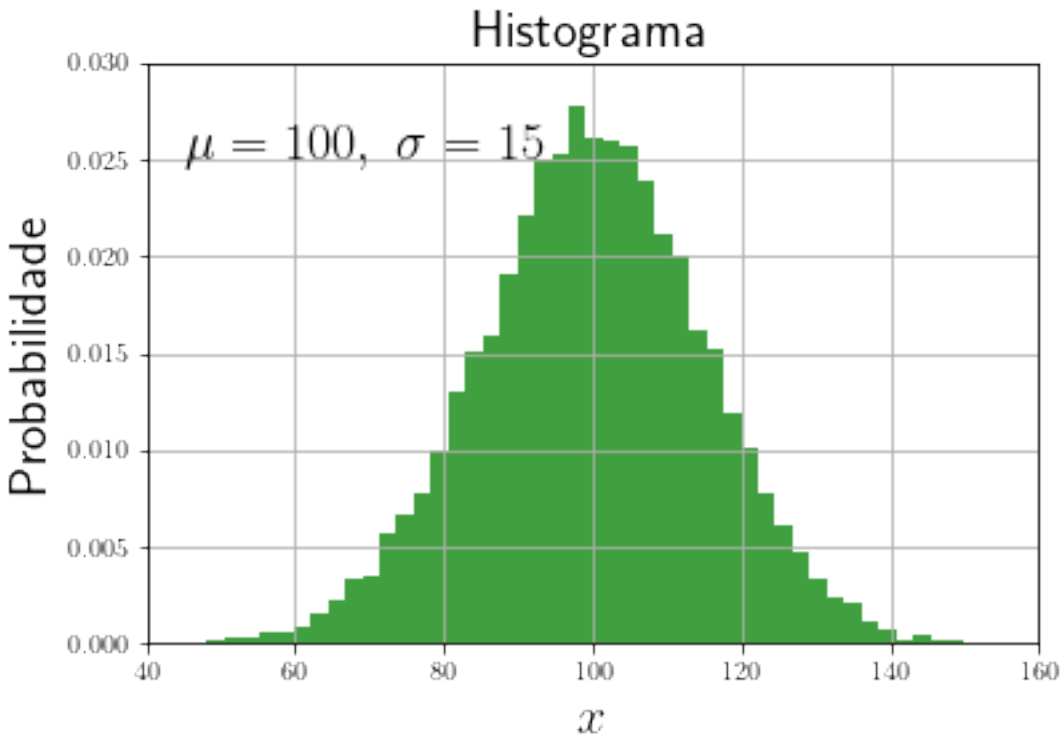
Exemplo 4:

[388]:
```
names = ['Athos', 'Porthos', 'Aramis']
values = [5, 25, 125]

fig, ax = plt.subplots(1, 3, figsize=(15, 7))

ax[0].bar(names, values, color="red", label="Gráfico de barras.")
ax[0].legend()

ax[1].scatter(names, values, color="orange", label="Gráfico de pontos.")
ax[1].legend()

ax[2].plot(names, values, color="green", label="Gráfico de linha.")
ax[2].legend()

fig.savefig("figura1.pdf", format = "pdf", bbox_inches = 'tight', pad_inches =␣
↪0.2)
```



Exemplo 5:

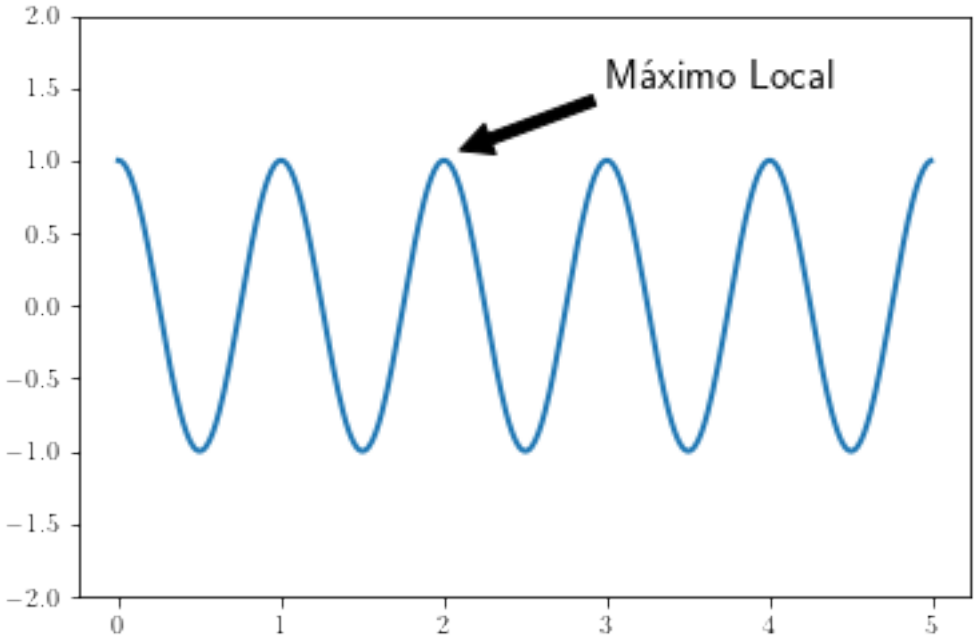
[389]:
```
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

fig, ax = plt.subplots()

ax.hist(x, 50, density=1, facecolor='g', alpha=0.75)

#n, bins, patches = plt.

ax.set_xlabel(r'$x$', fontsize=20)
ax.set_ylabel(r'Probabilidade', fontsize=20)

ax.set_title(r'Histograma', fontsize=20)

ax.text(45, .025, r'$\mu=100,\ \sigma=15$', size = 20)

ax.axis([40, 160, 0, 0.03])

ax.grid(True)
```



Exemplo 6: podemos colocar desenhos no gráfico, e texto também.

[9]:
```
fig, ax = plt.subplots()

x = np.arange(0.0, 5.0, 0.01)

y = np.cos(2*np.pi*x)

ax.plot(x, y, lw=2)

ax.annotate(r'Máximo Local', xy=(2.05, 1.05), xytext=(3, 1.5),
            arrowprops=dict(facecolor='black', shrink=0.05),
            fontsize = 15)

ax.set_ylim(-2.0, 2.0)
```

[9]:
```
(-2.0, 2.0)
```



Para mais informações e exemplos, dê uma olhada no site oficial da *matplotlib* (https://matplotlib.org/) e também em tutoriais na internet.

6.5 Tarefa de Casa

1. Faça um gráfico com o preço do bitcoin nos últimos 20 dias.
2. Crie uma função que plote o gráfico $y = ax^3 + bx^2 + cx + d$, dados a, b, c e d e o intervalo de x para plotar.
3. Cria uma matriz 5x5 de números aleatórios e calcule a média desses números.
4. Vamos desenvolver o jogo Pedra, Papel e Tesoura. Você vai pedir para o usuário entrar a escolha dele. Em seguida você vai gerar a escolha do computador. Use números aleatórios do Python para gerar essa escolha. Primeiro faça um jogo justo, no qual o usuário e o computador têm a mesma chance de ganhar. Depois faça um jogo no qual o computador tem uma chance maior de ganhar. Atenção: nesse segundo jogo o computador não vai ganhar todos os jogos, mas no final ele vai ganhar mais do que o usuário. (Dica: dê um *dir* na biblioteca *numpy* e procure por números aleatórios, ou *random numbers*.)

6.6 Comentário Final

Não abordei aqui no curso um assunto muito importante, que é **manipulação de arquivos externos**.

Com o Python podemos **criar** arquivos externos e **escrever** dados nesses arquivos.

Também podemos **ler** dados existentes em arquivos externos.

Uma biblioteca muito usada para isso é a *pandas*.

Deixo aqui um tutorial e um vídeo explicando como fazer essas manipulações.

http://devfuria.com.br/python/manipulando-arquivos-de-texto/

https://youtu.be/UCOJWSnKCT0

((Não sei como fazer isso no Google Colab, ainda estou aprendendo.))

6.7 Próximos Passos

O mais **importante** agora é: **escolha um projeto.** Algo que te motive.

Pode ter alguma utilidade para você, ou não. Execute esse projeto usando Python.

É assim, e somente assim, que você vai aprender a programar: **programando**, resolvendo problemas.

Seu primeiro código vai ficar feio? Vai. Vai ser o mais eficiente do mundo? Com certeza não.

Mas vai funcionar e você vai aprender muito criando esse código.

Depois de colocar esse projeto pra funcionar, aí você começa a pensar em maneiras para deixá-lo mais organizado e mais eficiente.

Ideias para projetos em Python (escolha um e só passe para o próximo depois de conluir o primeiro):

1. Exporte uma animação (gif ou vídeo), usando matplotlib, da colisão de duas partículas (easy: 2D; hard: 3D).
2. Programe o seu jogo da velha. Primeiro sem interface gráfica, na própria janela de comando. Depois com interface gráfica (ver tkinter).
3. Faça um jogo usando o pygame ou o turtle (hard: transforme em um app).
4. Desenvolva um relógio de desktop usando o tkinter.
5. Crie uma agenda usando o Django.

Github. Faça uma conta AGORA! Entre nas contas dos usuários e veja como eles programam (estilo do código, arquivos, comentários…). Assista a esses dois vídeos:

1. https://www.youtube.com/watch?v=DqTITcMq68k
2. https://www.youtube.com/watch?v=UBAX-13g8OM

Canais no youtube. Comece a ver conteúdos sobre Python para ir se ambientando com os termos utilizados e com a atmosfera de programação. Dê uma olhada nesses canais:

1. https://www.youtube.com/c/FilipeDeschamps
2. https://www.youtube.com/c/rafaellaballerini
3. https://www.youtube.com/c/PythonProBr
4. https://www.youtube.com/c/DevAprender
5. https://www.youtube.com/c/realpython

Tutorial oficial do Python, em português:

- https://docs.python.org/pt-br/3/tutorial/

Guia de estilo para programar em Python (PEP 8):

- https://wiki.python.org.br/GuiaDeEstilo

Livros:

1. Introdução à Programação com Python. Nilo Ney Coutinho Menezes. (Programação geral)
2. Aprenda Python 3 do Jeito Certo. Zed Shaw. (Python geral, baseado em exercícios)
3. Use a Cabeça! Python. Paul Barry. (Livro bem prático)
4. Python Crash Course. Eric Matthes. (Python geral, com muitos projetos)
5. Matplotlib Plotting Cookbook. Alexandre Devert. (Fazer gráficos com o matplotlib)
6. Effective Computation in Physics. Anthony Scopatz e Kathryn D. Huff. (Programação científica)
7. Numerical Methods in Engineering with Python 3. Jaan Kiusalaas. (Programação científica, voltado para engenharia)

Existem também muitas apostilas excelentes na internet.

Aqui acaba o nosso curso.

Espero que vocês tenham gostado. Eu aprendi muito aqui!!

Muito Obrigado!!!

Adriano Possebon Rosa (possebon.adriano@gmail.com)
