Curso de SymPy

Eduardo Adame Salles

23 de julho de 2021

Sumário

1 Conhecendo e Preparando o Ambiente

1.1 Introdução

Esse curso busca capacitar seus estudantes no tocante à biblioteca Sympy, do Python. Portanto, há a necessidade do conhecimento do próprio Python base. Com isso, o curso foi dividido em uma ordem lógica de aprendizagem para que o aluno consiga aprender continuamente, sem grandes "degraus" entre os assuntos. O capítulo 1 é uma introdução ao Python base, o que é suficiente para esse curso.

Neste capítulo, temos como objetivo fazer a instalação e configuração do nosso ambiente de desenvolvimento. Se você já é desenvolvedor Python, não acredito que seja necessária sua mudança ao ambiente que será indicado a esse curso. Mas, caso seja iniciante, recomendo que siga as instruções.

1.2 Escolhendo um Ambiente

Há duas opções para seguir esse curso:

1. Na forma de Notebooks
2. Na forma de Scripts

No curso, serão utilizado notebooks. Mais especificamente, Jupyter Notebooks, através do ambiente JupyterHub. Mas, não se preocupe, ambas as opções serão eficientes. Embora a utilização do Sympy seja regularmente associada ao desenvolvimento em Jupyter.

Na primeira opção, você só terá que instalar um pacote que virá com tudo incluso (IDE, Pacotes, etc.), e sua programação será mais parecida como um caderno. Onde você pode escrever, colocar imagens, matemática, e separar o código em blocos. Contudo, ele é mais pesado e menos flexível.

Na segunda opção, você terá um ambiente mais semelhante à programação tradicional. Linha após linha e comentários. Você terá que instalar o Python puro, versionar/atualizar seus módulos manualmente (o que é mais trabalhoso, mas dá mais autonomia) e utilizar um Editor de Código/IDE.

Caso não tenha entendido o que fora explicado acima, acredito que, nesse momento inicial, a primeira opção é a melhor para você. Contudo, ela não te ajudará a migrar para outra linguagem de programação no futuro, caso queira ou seja necessário. (Isso não é um problema, ao meu ver).

1.3 Instalando o Ambiente (1ª Opção)

1.3.1 Windows

No Windows, a instalação é bem simples, utilizaremos o Anaconda. Para isso, baixe o instalador do Anaconda.

Prossiga a instalação normalmente. Contudo, preste atenção às duas caixas de marcação (checkboxes) que perguntam sobre adicionar o Anaconda ao PATH e torná-lo padrão. Ambas as opções devem estar marcadas.

Se ele oferecer a instalação de algum outro IDE (como o PyCharm), você pode negar.

1.3.2 Linux

A instalação no Linux é um pouco mais complicada. Mas, se você utiliza Linux, possivelmente consegue instalar com certa facilidade.

Verifique as dependências da sua distribuição nesse link: https://docs.anaconda.com/anaconda/install/linux/

E depois siga a instalação com o instalador para Linux. Ele é um instalador em Bash, então recomendo que utilize os comandos prescritos no site do Anaconda.

1.4 Instalando o Ambiente (2ª Opção)

Como dito na seção anterior, para essa opção temos que instalar 2 programas, além dos pacotes separadamente.

1.4.1 Windows

No Windows, você terá que baixar e instalar o Python que está disponível nesse link: https://www.python.org/downloads/

A instalação é bem simples, você só deve se certificar que ele será adicionado ao PATH e que o `pip` será instalado. (São duas caixinhas de marcar que devem aparecer)

Para testar se a instalação deu certo, abra um prompt de comando (basta inserir ou cmd ou powershell no menu de pesquisa) e digite:

```
py --version
```

Deve aparecer a versão do Python que você instalou. Caso isso não ocorra, repita o procedimento e veja se fez tudo corretamente.

Também confirme se o pip foi instalado corretamente:

```
pip --version
```

Após isso, você deve escolher um editor. A minha recomendação é o Visual Studio Code.

Mas você pode utilizar outros como o Atom ou PyCharm.

A instalação de todos é bem simples. Ao abri-los, certifique-se se é necessário instalar uma extensão para suporte ao Python. Caso for, instale-a.

1.4.2 Linux

No Linux, a imensa maioria das distribuições já vêm com o Python instalado e disponível. A única diferença é que algumas adotam o nome `python3` e outras somente `python`.

Para isso, teste no terminal:

```
python --version
python3 --version
```

E comece a usar o que tem a versão mais atual.

Também certifique-se se você tem o `pip` instalado

```
pip --version
pip3 --version
```

Caso não tenha, veja como instalá-lo em sua distribuição.

Assim como no Windows, você deve instalar um editor. As minhas recomendações são as mesmas para o Windows. Ou seja, primeiramente o Visual Studio Code. E depois ou o Atom ou o PyCharm.

Todos têm versão para Linux e a instalação deve ser até mais simples que para windows. No caso do Visual Studio Code, recomendo que utilize a versão flatpak ou snap.

1.5 Como acompanhar as explicações do curso

O nosso curso será baseado em Jupyter Notebooks, uma forma de programar em blocos. Esses blocos podem ser tanto de código como de texto.

No caso, a primeira opção de ambiente lhe entregará um ambiente muito parecido com o que eu utilizei para escrever os meus Notebooks. Para a segunda, acredito que será mais difícil. Principalmente, pela ausência de alguns pacotes e softwares que vêm com o Anaconda. Mas, decidi incluí-la, pois há pessoas que precisam somente de uma consulta rápida, e possivelmente já têm seu ambiente de desenvolvimento, sendo em Anaconda ou não.

Então, para a primeira opção, você deverá abrir o "Anaconda Navigator". Esse software servirá como gerenciador de pacotes e dos serviços.

Como dito anteriormente, utilizaremos Jupyter Notebooks. Então basta clicar em "Launch" no card do Jupyter Notebook.

Ele abrirá um navegador de arquivos no navegador, semelhante a um site. Basta escolher uma pasta onde gostaria de criar seus notebooks, e criá-los a partir do botão "New" e então escolha o *Python 3*.

O curso não tem como foco específico ensinar o uso desse ambiente, uma vez que ele é bem simples. Basicamente, você deverá criar blocos (que costumamos chamar de *chunks*) de forma a organizar o entendimento e a saída do seu código. Ao escolher a opção "Code" para seu bloco, você deverá escrever códigos Python nele. No caso de "Markdown", você deverá escrever texto de acordo com a sintaxe da linguagem de marcação Markdown.

Essa sintaxe é bem simples, dê uma olhada nessa "tabela de cola": https://github.com/luong-komorebi/Markdown-Tutorial/blob/master/README_pt-BR.md

Para qualquer tipo de bloco, basta segurar a tecla Ctrl e apertar Enter para compilá-lo.

1.6 Exercícios

1. Crie um chunk de markdown e escreva um título com o texto "Olá mundo em Python".
2. Abaixo desse chunk, crie outro com o código `print('Olá mundo')`.

1.7 Próximos passos

Embora esse capítulo não tenha tido um bloco sequer de código, acredito que agora estão preparados para aprender os básicos de Python

2 Básicos da Programação em Python

2.1 Sobre o Python

Não é por acaso que a linguagem de programação Python tem se tornado cada vez mais popular. Além do seu uso na computação, que ganhou um novo *hype* por conta da Inteligência Artificial, a linguagem é uma ótima forma de introduzir uma pessoa à programação, de modo geral.

A linguagem apresenta diversas facilidades em sua sintaxe, que não somente contibuem para a facilidade da escrita, mas também contribuem para o entendimento e documentação do código.

Novamente, para não fugir do escopo do curso, não acredito que seja necessário listar as especificações técnicas da linguagem. Acredito que há somente alguns conceitos simples que são necessários para o uso da linguagem.

- O Python é uma linguagem interpretada.

Ou seja, o software gerado pelo código é o próprio código interpretado durante a execução

- O Python interpretará linha após linha, do início ao fim do script (chunk, para nós).

Para novatos em programação, isso deve ser fácil de aceitar. Mas pessoas com certa experiência podem questionar essa afirmação. Nesse caso, estou me referindo a forma que vamos utilizar a linguagem no curso e até onde iremos com as estruturas de programação da linguagem.

- O Python diferencia letras maíusculas de minúsculas.

Isso é chamado de "sensitive case". Tome cuidado ao reproduzir os exemplos e ao programar, de modo geral.

- O Python reconhece espaços vazios.

Isso é utilizado para definir o escopo dessa linha de código. Não se preocupe caso não tenha entendido, é mais simples do que aparenta ser. Isso basicamente é pra mostrar o "quão dentro" uma linha está de uma estrutura. Abordaremos isso mais tarde.

- Texto precedido por `#` não são interpretadas

Tudo que vier após um `#` em uma linha é chamado de comentário. Ele não é compilado e utilizaremos para comentar o código.

2.2 Definindo variáveis

Uma variável é uma forma de armazenar certo valor e reutilizá-lo. Se quiser fazer uma comparação com a Matemática, apenas enxergue ele como um x_0 não um x somente. Ou seja, ele vai ter um valor específico que pode mudar. Não necessáriamente ele seria "todos os valores ao mesmo tempo" como uma variável na matemática.

Por exemplo, vamos criar a variável `name` que vai armazenar meu nome.

[1]:
```
name = "Eduardo"
```

Agora, eu posso utilizar name em qualquer lugar do meu código. No caso, se eu mudar o valor de name, todo lugar onde name está terá seu valor trocado.

Por exemplo:

[2]:
```
name = "Eduardo"
name
```

[2]:
```
'Eduardo'
```

[3]:
```
name = "Adame"
name
```

[3]:
```
'Adame'
```

Para ficar mais claro, podemos ver algo como:

[4]:
```
name = "Eduardo"
name = "Adame"
name
```

[4]:
```
'Adame'
```

Como pôde ver no *chunk* acima, embora eu tenha atribuído o valor `"Eduardo"` ao `name` na primeira linha, o valor dele foi sobrescrito pela linha seguinte.

2.3 Tipos de Variáveis

Acredito que deve ter notado que colocamos o valor entre `" "`. Por quê?

Isso tem a ver com o tipo do valor. No Python, os tipos são dinamicamente interpretados através de sua sintaxe. E, nesse caso, nós inserimos uma `string`, que é utilizada para armazenar texto.

Vamos falar sobre os principais tipos de variáveis que já vêm com o Python e como defini-las.

2.3.1 Inteiros

Os inteiros são simplesmente definidos ao igualar uma variável a um número que pertença ao conjunto dos números inteiros, seja positivo ou negativo.

[5]:
```
my_integer = 2
my_integer
```

[5]:
```
2
```

2.3.2 Floats

Os floats, diferentemente dos inteiros, podem ser qualquer número real. Utilizamos ponto (`.`) para separar as casas decimais.

[6]:
```
my_float = 2.5
my_float
```

[6]:
```
2.5
```

2.3.3 Strings

Como dito anteriormente, strings são utilizados para textos. Basta envolver seu texto em aspas (`" "`) ou (`' '`). Para o Python, tanto faz se elas são simples ou duplas.

[7]:
```
my_string = "Textos são bem úteis, não acha?"
my_string
```

[7]:
```
'Textos são bem úteis, não acha?'
```

2.3.4 Booleanos

Esse tipo de valor é utilizado para condições. Eles podem ser "Falso" (`False`) ou "Verdadeiro" (`True`). Preste atenção no sensitive case.

[8]:
```
do_i_like_math = True
do_i_like_rlang = False
do_i_like_math, do_i_like_rlang
```

[8]:
```
(True, False)
```

2.3.5 Listas

Nós utilizamos listas para agrupar outros valores. Uma lista pode armazenar qualquer tipo e em qualquer quantidade. Inclusive podemos fazer listas de listas.

[9]:
```
names = ["Eduardo", "Marcos", "Ruan", "João"]
names
```

[9]:
```
['Eduardo', 'Marcos', 'Ruan', 'João']
```

[10]:
```
my_id = ["Eduardo", 18, 1.78]
my_id
```

[10]:
```
['Eduardo', 18, 1.78]
```

Para acessar um valor em específico na lista utilizamos `[]` com o índice desse valor. A lista começa a contagem em 0. Veremos algumas outras coisas necessárias para o uso de listas posteriormente.

[11]:
```
names[1] # "Eduardo" é o 0
```

[11]:
```
'Marcos'
```

[12]:
```
my_id[2]
```

[12]:
```
1.78
```

2.4 Operações

Os operadores e as operações para cada tipo são diferentes. Principalmente quando ocorrem entre dois tipos diferentes. Os principais para cada um deles são:

2.4.1 Aritmética

Com `int` e `float` é possível utilizar os operadores comuns da Matemática. Quando ocorre entre eles, o resultado será um `float`.

Os principais operadores são:

- `+`/`-` para adição/sutração.
- `*`/`//` para multiplicação/divisão.
- `**` para potenciação.
- `%` para módulo (resto).

Para alguns exemplos, como esse, não criarei variáveis, mas é possível criar para qualquer operação.

[13]:
```
5 + 2 + 4 * 10
```

[13]:
```
47
```

[14]:
```
2 ** (1/2) # Raíz de 2
```

[14]:
```
1.4142135623730951
```

Você também pode utilizar esses operadores em cima de uma variável. Por exemplo, você tem uma variável e quer acrescer 2 nela.

[15]:
```
my_num = 14
my_num = my_num + 2
my_num
```

[15]:
```
16
```

O exemplo acima é bem convincente e, certamente, bem útil. Mas há uma forma reduzida desse tipo de expressão.

[16]:
```
my_num = 14
my_num += 2
my_num
```

[16]:
```
16
```

Esse tipo de "operador" pode ser criado ao unir um desses operadores listados acima com o símbolo de `=`.

[17]:
```
my_num = 2
my_num *= 10
my_num
```

[17]:
```
20
```

2.4.2 Booleanas

As operações booleanas são expressões que retornam ou `True` ou `False`, são utilizadas para condições.

No Python, costumamos escrever por extenso a maioria das condições.

- `and` para operação "e"
- `or` para operação "ou"
- `not` para operação de negação (inverte o valor)
- `>`/`>=` maior/maior ou igual
- `<`/`<=` menor/menor ou igual
- `==` igual
- `in`confere está em uma lista

Exemplos:

[18]:
```
30 > 15
```

[18]:
```
True
```

[19]:
```
30 < 15
```

[19]:
```
False
```

[20]:
```
14 == 14
```

[20]:
```
True
```

[21]:
```
30 < 15 and 14 == 14
```

[21]: False

[22]:
```
30 <= 15 or 14 == 14
```

[22]: True

[23]:
```
14==14 and not 30 < 15
```

[23]: True

[24]:
```
lst = [1,2]
2 in lst
```

[24]: True

[25]:
```
True and False
```

[25]: False

[26]:
```
False or not False
```

[26]: True

2.4.3 Strings

Podemos formatar strings e fazer algumas operações com elas

- `+` para concatenação
- `*` para concatenação repetidas vezes

Exemplos:

[27]:
```
'Eduardo ' + 'Adame'
```

[27]: 'Eduardo Adame'

[28]:
```
'du'*2
```

[28]: 'dudu'

Strings formatadas As strings formatadas são um modo de inserir variáveis em strings. Adicionamos um `f` antes das áspas e colocamos a variável entre chaves `{}`.

Exemplo:

[29]:
```
nome = 'Eduardo'
sobrenome = 'Adame'
```

```
idade = 18
f'Meu nome é {nome} {sobrenome} e tenho {idade} anos'
```

[29]: 'Meu nome é Eduardo Adame e tenho 18 anos'

2.5 Conversão de Tipos

Basta utilizar a função (veremos mais tarde funções em específico) que tem o nome do tipo.

- `int()` para converter para `inteiro`
- `str()` para converter para `string`
- `bool()` para converter para `booleano`
- `float()` para converter para `float`

```
[30]: int('10')
```

[30]: 10

```
[31]: int(20.0)
```

[31]: 20

```
[32]: float('12.23')
```

[32]: 12.23

```
[33]: str(30.2)
```

[33]: '30.2'

```
[34]: bool(0)
```

[34]: False

```
[35]: bool(1) #ou qualquer outro número diferente de 0
```

[35]: True

```
[36]: bool('')
```

[36]: False

```
[37]: bool('a') #ou qualquer string não vazia
```

[37]: True

2.6 Recebendo entrada do usuário

Para isso, utilizamos a função `input()`, que recebe uma `string` como parâmetro para a mensagem impressa, e retorna a entrada como uma `string` (devemos converter, se necessário). Para imprimir uma mensagem utilizamos `print()` que imprime qualquer coisa que for passada, costuma-se utilizar as operações vistas anteriormente.

Exemplo:

[38]:
```
number = input('Insira um número: ')
number * 2
```

```
Insira um número:  10
```

[38]:
```
'1010'
```

Por que ao invés de `20` recebemos `'1010'`? Porque `10` foi recebido como `'10'`, ou seja, uma `string`.

Logo, o exemplo correto seria:

[39]:
```
number = float(input('Insira um número: ')) #float para aceitar qualquer número
number * 2
```

```
Insira um número:  10
```

[39]:
```
20.0
```

2.7 Condições

Em Python, como na maioria das linguagens de programação podemos utilizar estruturas condicionais para interpretar certas linhas de código somente se certa condição for atendida.

No caso, aqui faremos o uso das palavras-chave `if`, `else` e `elif`.

A partir de agora, veremos que os espaços em branco são importantes para denotar se certa linha está dentro de uma estrutura.

2.7.1 If

[40]:
```
word = "Renato"
if 2 > 1:
    word = "Eduardo"
word
```

[40]:
```
'Eduardo'
```

Como visto acima, o if verificou se o booleano (resultado da operação) era verdadeiro. Como era, executou a linha que definia o novo valor para `word`. Caso fosse falso teríamos o seguinte resultado:

[41]:
```
word = "Renato"
if 1 > 2:
    word = "Eduardo"
word
```

[41]:
```
'Renato'
```

Agora, se quisermos fazer algo se tal condição for real (como vimos acima) mas, se caso contrário, fazer outra ação? Utilizaremos o `else`.

[42]:
```
word = "Renato"
if 1>2:
    word = "Eduardo"
else:
    word = "Flávio"
word
```

[42]:
```
'Flávio'
```

A estrutura acima é muito simples e frequentemente utilizada. Ela executa o primeiro bloco caso a condição for verdadeira, caso contrário, executa a segunda.

Mas se quisermos criar condições intermediárias (ou específicas) utilizamos o `elif`. Ele significa algo como: "Se o anterior for falso, verifica se essa outra condição é verdadeira antes de ir pro `else`".

[43]:
```
lst = [0,1,2]

if 3 in lst:
    print("3 está!") # Em notebooks o print é desnecessário, mas esse é um␣
 ↪exemplo geral.
elif 2 in lst:
    print("2 está!")
else:
    print("3 e 2 não estão!")
```

```
2 está!
```

2.8 Métodos

Métodos são funções chamadas a partir de um objeto de um tipo específico. Novamente, não precisa se preocupar caso não tenha entendido, é mais fácil do que aparenta.

Esses métodos são acessados através de `.`. E cada tipo terá os seus, ou seja, `str` tem seus próprios métodos, assim como `list` tem seus próprios.

Não é viável tratar cada um deles aqui, mas é interessante que sempre busque pela documentação quando sentir necessidade. Por exemplo, aqui estão todos os métodos de `list`.

Alguns exemplos:

[44]:
```
# Todas as letras maiúsculas em uma string
word = "eDuArdO"
word = word.upper()
word
```

[44]: 'EDUARDO'

[45]:
```
# Todas as letras minúsculas em uma string
word = "eDuArdO"
word = word.lower()
word
```

[45]: 'eduardo'

[46]:
```
# Somente a primeira letra maiúscula em uma string
word = "eDuArdO adaME"
word = word.title()
word
```

[46]: 'Eduardo Adame'

Esses métodos de strings são muito úteis em condições. Note que `'eduardo'` é diferente de `'Eduardo'`. Outros exemplos são mais úteis para limpeza de dados e/ou correção, como o abaixo.

[47]:
```
# Substitui todas ocorrências de uma substring
phrase = "Penso, logo existo"
phrase = phrase.replace('logo', 'portanto')
phrase
```

[47]: 'Penso, portanto existo'

Para listas, os seus métodos são suas principais "operações".

[48]:
```
lst = [0,2,4]
# Adiciona um item a uma lista
lst.append(6)
lst # Note que ele altera a variável diretamente
```

[48]: [0, 2, 4, 6]

[49]:
```
if 2 in lst: # Forma de evitar erros
    lst.remove(2) # Remove um item da lista
lst
```

[49]: [0, 4, 6]

[50]:
```
# Remover um item pelo seu índice
lst.pop(2)
```

```
lst
```

[50]: [0, 4]

[51]:
```
# Extende a lista com outra (semelhante a uma soma)
lst.extend([2,6])
lst
```

[51]: [0, 4, 2, 6]

[52]:
```
# Limpa a lista
lst.clear()
lst
```

[52]: []

2.9 Estruturas de Repetição

Esse tipo de estrutura, como diz o seu nome, é utilizada para repetir certo bloco de código. No caso, há duas formas de criar esse tipo de *loop*.

- `while`: Repete o bloco enquanto certa condição verdadeira
- `for`: Repete o bloco para cada item em uma lista

Nesse ponto do curso gostaria de fazer uma menção sobre uma decisão que tomei quanto ao método de ensino. Utilizar listas não é a unica forma de agrupar elementos, por exemplo, existem os `sets` e as `tuples`. Contudo, todos eles são iteráveis. Ou seja, podemos navegar por seus elementos. Portanto, tudo que se refere a iterabilidade da lista, também serve para esses outros tipos (que pretendo falar posteriomente).

Exemplos:

[61]:
```
entry = int(input('Digite um inteiro: '))
while entry != 7: # Verifica se a entrada é igual a 7.
    entry = int(input('Digite um inteiro: '))
print("Loop encerrado!")
```

```
Digite um inteiro:  2
Digite um inteiro:  3
Digite um inteiro:  4
Digite um inteiro:  8
Digite um inteiro:  7
```

```
Loop encerrado!
```

Tome cuidado com loops infinitos! A estrutura `while` é bem propícia a isso.

[54]:
```
lst = [0,2,4,6]
for item in lst: # Para cada item da lista
```

```
    print(item ** 2) # Imprima o dobro do item
```

```
0
4
16
36
```

É possível criar coisas muito poderozas com o for, inclusive na visualização de dados (em texto).

[55]:
```
lst = [0,2,3,5,6]
for item in lst: # Para cada item da lista
    sq = item **2
    if item % 2 == 0: # Checa se o número é divisível por 2 (par)
        is_even = "par"
    else:
        is_even = "ímpar"
    print(f'O número {item} é {is_even}, e seu quadrado é {sq}') # Imprima o␣
 ↪dobro do item
```

```
O número 0 é par, e seu quadrado é 0
O número 2 é par, e seu quadrado é 4
O número 3 é ímpar, e seu quadrado é 9
O número 5 é ímpar, e seu quadrado é 25
O número 6 é par, e seu quadrado é 36
```

2.10 Funções

Funções são blocos de código que podem ser reutilizados. Elas podem ser definidas de duas formas:

- Utilizando `def`
- Utilizando `lambda` (chamadas funções anônimas)

O uso mais comum, entretanto, é utilizando o `def`.

Exemplo:

[64]:
```
def square(x):
    return x**2

square(89)
```

[64]:
```
7921
```

Note que, ao utilizar `def`, nós criamos um bloco. Nós devemos nomear uma função e colocar entre parênteses seus parâmetros (ou deixá-lo vazio). É possível definir valores padrões para certos parâmetros. As variáveis criadas dentro do bloco só existirão dentro dele (inclusive sobrescrevendo temporariamente uma variável global).

O termo return se refere a linha que define o valor retornado por essa função. Quando utilizamos notebooks, o Jupyter imprime o retorno da última linha, por isso não utilizamos print. Mas, utilizando

esse exemplo, se estivéssemos no desenvolvimento regular, seria algo como `print(square(89))`.

Alguns exemplos com `def`:

[65]:
```
def count_even(lst):
    count = 0
    for item in lst:
        if item % 2 == 0:
            count += 1
    return count

count_even([1,2,4,5,8])
```

[65]:
```
3
```

[69]:
```
# Como utilizar variáveis globais

breads = 100
currency = 0
def sell():
    global breads, currency # Crie as variáveis com a palavra-chave global
    breads -=1
    currency += .5

sell()

breads,currency
```

[69]:
```
(99, 0.5)
```

[89]:
```
def print_id(name, age, course = ''):
    result = f' Nome: {name} \n Idade: {age}' # \n para quebrar a linha
    if course:
        result += f' \n Curso: {course}'
    return print(result) # Necessário para \n funcionar

print_id('Eduardo', 18)
```

```
 Nome: Eduardo
 Idade: 18
```

[87]:
```
print_id('Eduardo', 18, 'Ciência de Dados')
```

```
 Nome: Eduardo
 Idade: 18
 Curso: Ciência de Dados
```

Para funções de linha única, o `lambda` pode ser uma opção melhor. Ou quando uma função recebe outra como parâmetro. Muitos cursos não abordam esse tipo de função, mas acredito que passar

rapidamente por ela pode ser bom.

Anteriormente definimos a função `square()`, podemos definir uma função que faz o mesmo trabalho da seguinte forma:

[88]:
```
sqr_anon =  lambda x: x**2

sqr_anon(89) # É invocada da mesma forma.
```

[88]:
```
7921
```

Nesse caso, a criamos como se fosse um valor atribuído. O que está entre `lambda` e `:` são os parâmetros e o que está depois dos `:` é o retorno. Nós perderemos o poder de criar o parâmetro opcional como em `def`, mas podemos fazer algo parecido com `print_id()`.

[90]:
```
id_anon = lambda name, age: f' Nome: {name} \n Idade: {age}' # Decidi deixar o␣
 ↪print de fora.
print(id_anon('Eduardo', 18))
```

```
 Nome: Eduardo
 Idade: 18
```

2.11 Exercícios

Agora você sabe mais que o suficiente para utilizar o `sympy`. Para verificar que absorveu o aprendido aqui, tente resolver os seguintes exercícios:

1. Crie uma função que recebe dois números (floats) e retorna o menor elevado pelo maior.
2. Crie uma função que recebe duas listas, uma com o ponto inicial e outra com o ponto final `[x,y]`, e calcule a distância entre eles.
3. Crie uma função que recebe números (floats) de quantidade indefinida (pesquise sobre `*args`) e retorne a soma deles.
4. Crie um loop que faz o mesmo que a função acima, contudo, ele deverá encontrar o valor total quando o usuário inserir `'s'`.
5. Crie uma função que calcula as raízes reais de uma equação do segundo grau a partir dos coeficientes `a`, `b`, `c`.

3 Primeiros Passos com o Sympy

3.1 Instalação

Você possivelmente deve estar se perguntando como instalar o `Sympy`. Se você já utilizou algum outro módulo em Python, possivelmente imaginou em instalá-lo utilizando o `pip` (software que pedimos que garantisse sua sua instalação no capítulo 0).

Contudo, se você utiliza está utilizando notebooks com o Anaconda (nossa recomendação para esse módulo), o `Sympy` já está instalado, basta carregá-lo.

Caso esteja desenvolvendo em outro ambiente, uma forma de instalar é com o pip, por exemplo:

```
pip install sympy
```

3.2 Carregando o Módulo

Para utilizar os comandos do `Sympy` de forma nativa em nossos scripts, precisamos importá-lo globalmente. Para isso utilizamos as palavras-chave `import` e `from`. Isso não foi abordado no capítulo anterior devido a sua complexidade, mas essa é uma forma de importar módulos em Python.

Portanto, basta criar e executar a seguinte *chunk*:

[1]:
```
from sympy import *
init_printing(use_unicode=True, use_latex='mathjax') # Para imprimir LaTeX
```

O `*` significa que estamos importando o módulo por completo.

3.3 Trabalhando com expressões matemáticas

Como o `Sympy` tem como objetivo o cálculo simbólico, tudo é baseado a partir dos simbólos. Ou seja, as nossas querídas variáveis (como x, y, e z) sendo interpretadas com suas propriedades matemáticas.

Portanto, para utilizá-las, precisamos criar seus símbolos. Por enquanto, vamos utilizar somente o x. Então, para o `x` do Python significar a variável x fazemos:

[2]:
```
x = symbols('x')
x
```

[2]:
$$x$$

`Note que nossa saída matemática será processada por um compilador LaTeX para facilitar a leitura.`

`No caso, você pode utilizar x como um número, e as expressões aparecerão normalmente (sem igualdade).`

[3]:
```
x**2 - 4*x + 3
```

[3]:
$$x^2 - 4x + 3$$

`Caso você queira a solução de uma expressão que seja igual a 0 (ou suas raízes, em outras palavras), em respeito a uma variável, você pode usar a função solve(). Ela recebe dois parâmetros obrigatórios, sua expressão e a variável que você quer a solução.`

[4]:
```
solve(x**2 - 4*x + 3, x)
```

[4]:
$$[1,\ 3]$$

[5]:
```
solve(sqrt(x) - (x/2),x)
```

[5]:
$$[0,\ 4]$$

Inclusive, caso queira uma resposta como costumamos escrever no papel, ou seja, em forma de conjunto e suas condições, podemos utilizar o solveset(). A maior diferença é que ele pode receber o conjunto numérico onde você quer trabalhar através do parâmetro domain. Na maioria das vezes podemos utilizar domain=S.Reals ou domain=S.Complexes.

[6]:
```
solveset(x**2 - 4*x  +20,x, domain=S.Reals)
```

[6]:
$$\emptyset$$

[7]:
```
solveset(x**2 - 4*x  +20,x, domain=S.Complexes)
```

[7]:
$$\{2 - 4i, 2 + 4i\}$$

[8]:
```
solveset(tan(x), x, domain=S.Reals)
```

[8]:
$$\{2n\pi \mid n \in \mathbb{Z}\} \cup \{2n\pi + \pi \mid n \in \mathbb{Z}\}$$

Vamos criar uma variável para armazenar essa primeira expressão para mostrar outros exemplos

[9]:
```
expr = x**2 - 4*x + 3
```

Podemos achar valores utilizando o método subs(). Novamente, devemos especficiar a variável.

[10]:
```
expr.subs(x,2) # Se y = expr, esse é o valor de y quando x = 2.
```

[10]:
$$-1$$

[11]:
```
expr.subs(x,1)
```

[11]:
$$0$$

Se tivermos uma expressão numérica não-inteira e quisermos achar a solução em um ponto flutuante (float), podemos usar o método evalf().

[12]:
```
my_sqrt = sqrt(8)
my_sqrt
```

[12]:
$$2\sqrt{2}$$

[13]:
```
my_sqrt.evalf()
```

[13]:
$$2.82842712474619$$

Como viu acima, possivelmente há uma função do SymPy que represente uma operação ou função matemática. Por exemplo, temos sqrt(), log(), exp(), sin() e etc.

Quando sentir necessidade de utilizar uma dessa, tente antes de consultar a documentação. Caso não consiga ``adivinhar'', faça uma consulta que, com toda certeza, haverá uma função que te atenderá.

Existem algumas funções que ``fazem Álgebra'' por si só. Veja alguns exemplos:

[14]:
```
# Simplifica
simplify((x**2 + x)/x)
```

[14]: $x + 1$

[15]:
```
# Fatora
factor(1-1/x)
```

[15]: $\frac{x - 1}{x}$

[16]:
```
# Expande
expand((x**2 + 3*x)**3)
```

[16]: $x^6 + 9x^5 + 27x^4 + 27x^3$

[17]:
```
# Agrupa potências de uma variável (que vai como segundo parâmetro)
collect(x**2 + 4*x - 2*x**2 + x -20 + x**3 + 2, x)
```

[17]: $x^3 - x^2 + 5x - 18$

[18]:
```
# Separa fração em frações parciais
apart((x**2 + 8*x-18)/(x**3 + 3*x**2))
```

[18]: $-\frac{11}{3\,(x+3)} + \frac{14}{3x} - \frac{6}{x^2}$

Além desses principais, ainda há trigsimp() e expand_trig() que simplificam e expandem funções trigonométricas (a partir das identidades de adição de arco). E outras que fazem o mesmo para potências, logaritmos e outros tipos de funções. Nesse caso, acho que vale a pena dar uma olhada na documentação. Elas todas são bem parecidas.

Finalizando esse tópico inicial, temos como substituir uma função em termos de outra. Por exemplo:

[19]:
```
sin(x).rewrite(cos)
```

[19]: $\cos\left(x - \frac{\pi}{2}\right)$

[20]:
```
(x**3).rewrite(exp)
```

[20]: $e^{3\log(x)}$

3.3.1 Equações

Ok, nós vimos como utilizar expressões. Mas, como tratamos equações? Como vimos no capítulo anterior = significa atribuição e == é uma operação booleana (ou seja, recebemos True ou False).

Para equações criamos uma classe Eq(). Não se preocupe com a nomenclatura, é só uma forma de criar um objeto do tipo Eq. Para criar esse objeto, passamos dois argumentos: cada lado da equação, respectivamente. Veja:

[21]:
```
eq = Eq(x**2, 2)
eq
```

[21]: $x^2 = 2$

Podemos utilizar o mesmo método para encontrar suas raízes.

[22]:
```
solve(eq,x)
```

[22]: $\left[-\sqrt{2},\ \sqrt{2}\right]$

3.3.2 Igualdade

Como verificar igualdade entre duas expressões? O == só servirá para expressões identicas (não somente em valor, mas também nos termos expressos). Para isso, utilizamos o método equals(). Veja:

[23]:
```
expr_1 = sin(x)**2
```

[24]:
```
expr_2 = .5*(1 -cos(2*x))
```

Como veremos abaixo, pelo == as expressões seriam diferentes.

[25]:
```
expr_1 == expr_2
```

[25]: False

Vejamos pelo método equals():

[26]:
```
expr_1.equals(expr_2)
```

[26]: True

Podemos visualizar a igualdade da seguinte forma:

[27]:
```
Eq(expr_1,expr_2)
```

[27]: $\sin^2(x) = 0.5 - 0.5\cos(2x)$

3.3.3 Sistemas de Equações

Podemos, também utilizando solve() encontrar as soluções de um sistema de equações. Basta passar uma lista com as equações como parâmetro.

[28]:
```
y, z = symbols('y z') # Criando o y simbólico
Eqs = []
Eqs.append(Eq(3*x - 3*y, 20))
Eqs.append(Eq(-7*x + 9*y, -10))
solve(Eqs, x, y)
```

[28]:
$$\left\{x : 25,\ y : \frac{55}{3}\right\}$$

Caso esteja tentando resolver um sistema linear (como o acima), é possível utilizar o linsolve().

[29]:
```
linsolve(Eqs, x, y)
```

[29]:
$$\left\{\left(25,\ \frac{55}{3}\right)\right\}$$

[30]:
```
Eqs = []
Eqs.append(Eq(3*x - 3*y + 2*z, 20))
Eqs.append(Eq(-7*x + 9*y -4*z, -10))
Eqs.append(Eq(-7*x + 9*y + 5*z, 40))
linsolve(Eqs, x, y,z)
```

[30]:
$$\left\{\left(\frac{175}{9},\ \frac{445}{27},\ \frac{50}{9}\right)\right\}$$

[31]:
```
Eqs = []
Eqs.append(Eq(x**2 + y**2, 18)) #  Elipse de centro (0,0) e R^2 = 18
Eqs.append(Eq(x, y)) # Reta identidade
nonlinsolve(Eqs, x, y) # Sistema não-linear (solve também serve). No nosso␣
↪caso, 2 pontos de intersecção.
```

[31]:
$$\{(-3,\ -3), (3,\ 3)\}$$

3.4 Matrizes

É bem trivial trabalhar com matrizes no Sympy. De modo geral, basta criar um objeto a partir da classe Matrix. E passamos uma lista de listas, sendo cada uma das listas uma linha. Veja:

[32]:
```
Matrix([[1,2,3],[2,3,1]])
```

[32]:
$$\begin{bmatrix}1 & 2 & 3\\ 2 & 3 & 1\end{bmatrix}$$

E podemos manipulá-las normalmente, com as operações comuns. Além disso, há algumas outras operações especiais.

[33]:
```
A = Matrix([[1,2,3],[2,3,1]])
B = Matrix([[3,2], [2,2], [1,4]])
A, B
```

[33]:
$$\left(\begin{bmatrix}1 & 2 & 3\\2 & 3 & 1\end{bmatrix}, \begin{bmatrix}3 & 2\\2 & 2\\1 & 4\end{bmatrix}\right)$$

[34]:
```
A * B # Multiplicação
```

[34]:
$$\begin{bmatrix}10 & 18\\13 & 14\end{bmatrix}$$

[35]:
```
B.T # Transpor
```

[35]:
$$\begin{bmatrix}3 & 2 & 1\\2 & 2 & 4\end{bmatrix}$$

[36]:
```
A + B.T # Soma
```

[36]:
$$\begin{bmatrix}4 & 4 & 4\\4 & 5 & 5\end{bmatrix}$$

[37]:
```
A.row(1) # Começa em 0
```

[37]:
$$\begin{bmatrix}2 & 3 & 1\end{bmatrix}$$

[38]:
```
B.col(0) # Também começa em 0
```

[38]:
$$\begin{bmatrix}3\\2\\1\end{bmatrix}$$

[39]:
```
B = B.col_insert(2,Matrix([2,3,2])) # Insere Coluna
B
```

[39]:
$$\begin{bmatrix}3 & 2 & 2\\2 & 2 & 3\\1 & 4 & 2\end{bmatrix}$$

[40]:
```
B**-1
```

[40]:
$$\begin{bmatrix}\frac{4}{7} & -\frac{2}{7} & -\frac{1}{7}\\\frac{1}{14} & -\frac{2}{7} & \frac{5}{14}\\-\frac{3}{7} & \frac{5}{7} & -\frac{1}{7}\end{bmatrix}$$

3.5 Exercícios

Utilizando o que aprendeu nesse capítulo, tente resolver os seguintes exercícios:

1. Encontre as raízes de cada uma das expressões abaixo. Depois encontre um par (x, y) para cada:

$$x^3 - 8x^2 + 4x + 3$$

$$\sin(x) + 2\cos(x)$$

$$\log\left|\frac{x^2 - x}{2}\right|$$

$$e^{-x^3+5x^2-x} - 1$$

2. Encontre as soluções das equações abaixo:

$$x^4 - 4x^3 + x^2 - 30 = -x^2 + x - 40$$

$$2^{x^2-x} = 3^x$$

$$\log\left|x^3 - 2x^2 + x\right| = \log\left|x^2 + 6x\right|$$

3. Verifique se as igualdades são verdadeiras:

$$\Gamma\left(\frac{3}{2}\right) = \frac{\sqrt{\pi}}{2}$$

$$\sin(2x^2) - \cos(x^2 + x) = \sin(x)\sin\left(x^2\right) + 2\sin\left(x^2\right)\cos\left(x^2\right) - \cos(x)\cos\left(x^2\right)$$

$$(x^2 - 3x)(2x^4 + x^3 - x)(-4x^2) = 8x^8 + 20x^7 + 12x^6 + 4x^5 - 12x^4$$

4. Encontre as soluções do sistema:

$$\begin{cases} 4x - 3y + 2z = 60 \\ (x - 10)^2 + y^2 + z^2 = 72 \\ 2x + 9y + z = 20 \end{cases}$$

4 Aplicações em Cálculo Diferencial e Integral

De modo geral, o principal objetivo do curso é garantir que seus alunos estejam proeficientes no uso de SymPy no Cálculo. Na minha opinião, esse é o capítulo mais importante do curso. Dê seu máximo para absorver o conteúdo aqui apresentado.

Antes de comerçarmos, certifique-se que fez as devidas importações e atribuições:

[1]:
```
from sympy import *
x, y, z = symbols('x y z')
init_printing(use_unicode=True, use_latex='mathjax')
```

4.1 Intervalos

Nós sabemos que o Cálculo é, genericamente, o estudo das mudanças. E nós costumamos definir intervalos para trabalhar com nossas funções e expressões. É bem simples de criá-los e utilizá-los no SymPy.

Para criar um intervalo, criamos um objeto a partir da classe Interval e/ou um método seu para definir se está aberto em algum dos lados. Veja os exemplos:

[2]:
```
# Intervalo Fechado
Interval(0,10)
```

[2]: $[0, 10]$

[3]:
```
# Intervalo Aberto
Interval.open(-10, 20)
```

[3]: $(-10, 20)$

[4]:
```
# Intervalo Aberto em um dos lados
Interval.Ropen(10,30) # R - Direita
```

[4]: $[10, 30)$

[5]:
```
Interval.Lopen(10,30) # L - Direita
```

[5]: $(10, 30]$

[6]:
```
Interval(0,oo) # oo representa o infinito em sympy. Note que onde oo estiver␣
↪será aberto.
```

[6]: $[0, \infty)$

4.2 Análises de Domínio/Intervalo

Existem diversas funções embutidas no SymPy para avaliar o comportamento das funções/expressões ao longo de seu domínio ou de um intervalo específico. Normalmente elas retornarão um booleano.

4.2.1 Verificar se é crescente ou decrescente.

[7]:
```
## x² em seu domínio não é crescente
is_increasing(x**2)
```

[7]: False

[8]:
```
## x² em (0, oo) é crescente
is_increasing(x**2, Interval.open(0, oo))
```

[8]: True

[9]:
```
## O contrário vale para decreasing
is_decreasing(x**2, Interval.open(-oo, 0))
```

[9]: True

Podemos verificar também se ela é estritamente crescente ou decrescente, ou seja, se ela é injetiva.

[10]:
```
## x³ é crescente em todo seu domínio. (d/dx = 3x² >= 0)
is_increasing(x**3)
```

[10]: True

[11]:
```
## x³ não é estritamente crescente em seu domínio (3*0² = 0)
is_strictly_increasing(x**3)
```

[12]:
```
## 1/(e^x) é estritamente decrescente em seu domínio
is_strictly_decreasing(1/(exp(x)))
```

[12]: True

Podemos também verificar se ela é monótona com is_monotonic(). Para finalizar, podemos verificar se há pontos (e quais são) com singularidades. Ou seja, que requerem certa atenção. Normalmente, são pontos que não têm limite.

[13]:
```
singularities(1/x,x)
```

[13]: $\{0\}$

4.3 Limites

Assim como veremos posteriormente nas derivadas e nas integrais, há duas formas de criar e calcular limites no SymPy. A primeria forma é através da classe Limit, que criará um limite e não calculará seu valor. Ou seja, utilize ela para armazenar a expressão do limite. Caso queira somente calcular o limite. Utilizamos a função limit().

[14]:
```
Limit(sin(x)/x, x, 0, '+') ## sin(x)/x, x -> 0+
```

[14]:
$$\lim_{x \to 0^+} \left(\frac{\sin(x)}{x} \right)$$

[15]:
```
Limit(1/x, x, 0, '-') ## sin(x)/x, x -> 0-
```

[15]:
$$\lim_{x \to 0^-} \frac{1}{x}$$

[16]:
```
limit(sin(x)/x, x, 0, '+')
```

[16]:
$$1$$

[17]:
```
limit(1/x, x, 0) # '+' por padrão
```

[17]:
$$\infty$$

[18]:
```
limit(1/x, x, 0, '-')
```

[18]:
$$-\infty$$

[19]:
```
limit(1/x, x, 0, '+-') # Dois lados
```

[19]:
$$\tilde{\infty}$$

[20]:
```
my_sin = Limit(sin(x)/x, x, 0, '+')
my_sin.doit() # Método doit() calcula uma expressão.
```

[20]:
$$1$$

4.4 Derivadas

Assim com os limites, podemos criar a derivada (sem calculá-la) através da classe Derivative(). E calcular diretamente através da diff().

[21]:
```
Derivative(exp(2*x**3),x)
```

[21]:
$$\frac{d}{dx} e^{2x^3}$$

[22]:
```
diff(sin(x**2),x)
```

[22]:

$$2x\cos\left(x^2\right)$$

[23]:
```
diff(sin(x**2),x, x) ## Calcular a segunda derivada
```

[23]:
$$2\left(-2x^2\sin\left(x^2\right)+\cos\left(x^2\right)\right)$$

[24]:
```
diff(sin(x**2),x, x, x) ## Calcular a terceira derivada
```

[24]:
$$-4x\left(2x^2\cos\left(x^2\right)+3\sin\left(x^2\right)\right)$$

[25]:
```
diff(sin(x**2),x, 3) ## Calcular a terceira derivada de outra forma
```

[25]:
$$-4x\left(2x^2\cos\left(x^2\right)+3\sin\left(x^2\right)\right)$$

[26]:
```
diff(sin(x**2),x, 10) ## Calcular a décima derivada
```

[26]:
$$32\left(-32x^{10}\sin\left(x^2\right)+720x^8\cos\left(x^2\right)+5040x^6\sin\left(x^2\right)-12600x^4\cos\left(x^2\right)-9450x^2\sin\left(x^2\right)+945\cos\left(x^2\right)\right)$$

[27]:
```
my_deriv = Derivative(exp(2*x**3),x)
my_deriv.doit()
```

[27]:
$$6x^2e^{2x^3}$$

[28]:
```
## Podemos, com o método diff()
expr = exp(2*x**3)
expr.diff(x)
```

[28]:
$$6x^2e^{2x^3}$$

[29]:
```
expr.diff(x,3) # Terceira derivada
```

[29]:
$$12\left(18x^6+18x^3+1\right)e^{2x^3}$$

4.5 Integrais

Assim como os Limites e as Derivadas que vimos acima, podemos criar uma Integral através da classe Integral() caso queiramos ter somente a expressão, e caso queiramos o resultado de uma Integral, basta utilizar a função integrate().

O Sympy não acresce a constante de integração nas Integrais Indefinidas, então é importante se lembrar delaa quando for resolver algum exercício.

[30]:
```
Integral(1/x, x)
```

[30]:
$$\int\frac{1}{x}\,dx$$

[31]:
```
Integral(1/x, (x, 1, 10)) # Note que passamos (simbolo, inf, sup)
```

[31]:

$$\int_{1}^{10} \frac{1}{x}\, dx$$

[32]:
```
integrate(1/x, (x,1,10))
```

[32]:
$$\log(10)$$

[33]:
```
my_integral = Integral(1/x + 1/y, (x, 1, 10), (y, 1, 10)) # Integral dupla, duas variáveis.
my_integral
```

[33]:
$$\int_{1}^{10}\int_{1}^{10} \left(\frac{1}{y} + \frac{1}{x}\right)\, dx\, dy$$

[34]:
```
my_integral.doit()
```

[34]:
$$18\log(10)$$

[35]:
```
Integral(exp(x**2 - 10), x,x) # Integral dupla indefinida, mesma variável
```

[35]:
$$\iint e^{x^2-10}\, dx\, dx$$

4.6 Outras funções

4.6.1 Séries

Você pode utilizar o método series() em uma expressão para fazer sua expansão em série.

[36]:
```
asin(x).series(x,0, 10) # (x, x_0, n)
```

[36]:
$$x + \frac{x^3}{6} + \frac{3x^5}{40} + \frac{5x^7}{112} + \frac{35x^9}{1152} + O\left(x^{10}\right)$$

4.6.2 Equações Diferenciais

Ao criar uma função simbólica, você pode utilizar derivadas e a função dsolve() para encontrar a solução de uma expressão e ou equação diferencial. Nesse caso, o SymPy insere as constantes quando necessário.

[37]:
```
f = Function('f')
my_deq = Eq(Derivative(f(x),x,2),f(x))
my_deq
```

[37]:
$$\frac{d^2}{dx^2} f(x) = f(x)$$

[38]:
```
dsolve(my_deq)
```

[38]: $f(x) = C_1 e^{-x} + C_2 e^{x}$

4.7 Exercícios

Como nos últimos capítulos, resolva os seguintes exercícios com o que aprendeu ao longo do curso.

1. Para cada uma das funções abaixo, encontre:

a) O domínio da função;

b) As assíntotas horizontais e verticais, caso existam;

c) Sua derivada, os intervalos de crescimento e decrescimento de f,os pontos de máximo e mínimo, caso existam;

d) Os intervalos onde o gráfico da f é côncavo para cima e onde é côncavo para baixo.

$$f(x) = \frac{x^3}{x+4}$$

$$g(x) = \frac{x^3 - 4x^2 + 5}{x^2 - x}$$

$$h(x) = x^2 - 4x + \frac{x}{x-10}$$

2. Calcule:

$$\int x^3 - 2x^2 + 3x + 10 \quad dx$$

$$\int x^3 \cdot \sin(2x) \quad dx$$

$$\int_0^{10} \tan^3(x) \sec^3(x) \quad dx$$

$$\int_1^{\infty} -\frac{1}{x^2} \quad dx$$

3. Qual a menor distância vertical entre as funções $f(x) = 32x^2$ e $g(x) = -\frac{8}{x^2}$?

5 Criando Gráficos

O uso de gráficos na matemática, principalmente no Cálculo e na Geometria Analítica, é de extrema importância. Embora esse não seja o foco do SymPy, é possível criar os chamados *plots* com certa facilidade. Inclusive, como é utilizado o matplotlib para criar os gráficos, há uma grande margem para alterações. Isso permite gráficos quase totalmente customizáveis.

Os tipos de plots abordados nesse capítulos serão os criados pelas funções:

- plot()
- plot_implicit()
- plot3d()

O seu uso é facilitado mais ainda pelo ambiente Jupyter, e podemos trabalhar com eles de forma semelhante ao que fora abordado no último capítulo. Ou seja, podemos criar um objeto para trabalharmos com ele, e depois ver o resultado. Ou, podemos simplesmente criar um plot.

Antes de começarmos, vamos a importação e as definições essenciais:

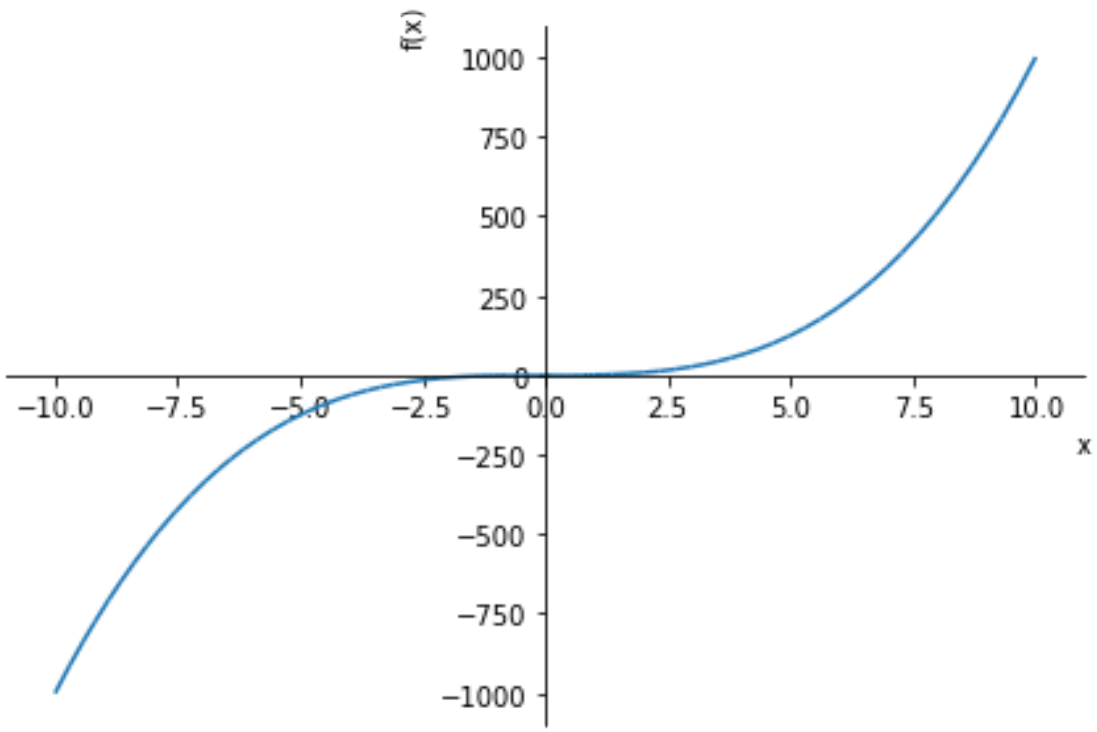
[1]:
```
from sympy import *
x, y, z = symbols('x y z')
init_printing(use_unicode=True, use_latex='mathjax')
```

5.1 Plot 2D

As duas primeiras funções que trabalharemos criam plots em 2D. Para fazer um plot de uma função, basta utilizar a função plot(). Confira abaixo:

5.1.1 Funções

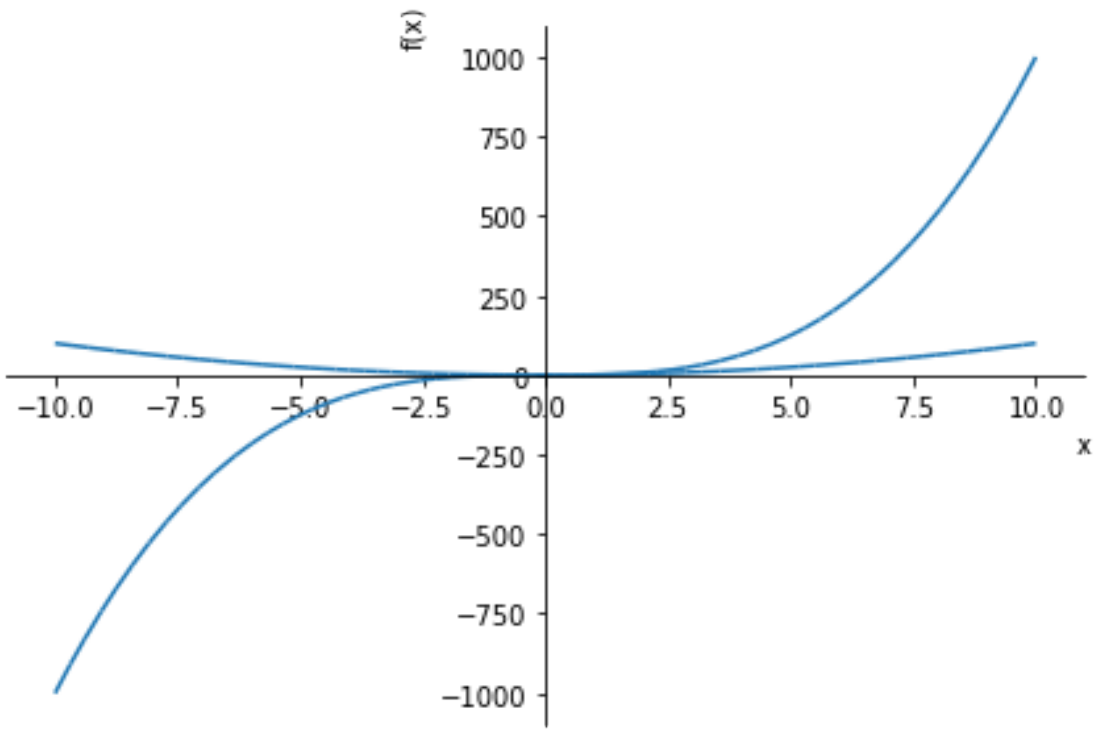
[2]:
```
plot(x**3)
```



[2]: <sympy.plotting.plot.Plot at 0x7f2220cd9710>

Podemos, inclusive, fazer vários plots unidos.

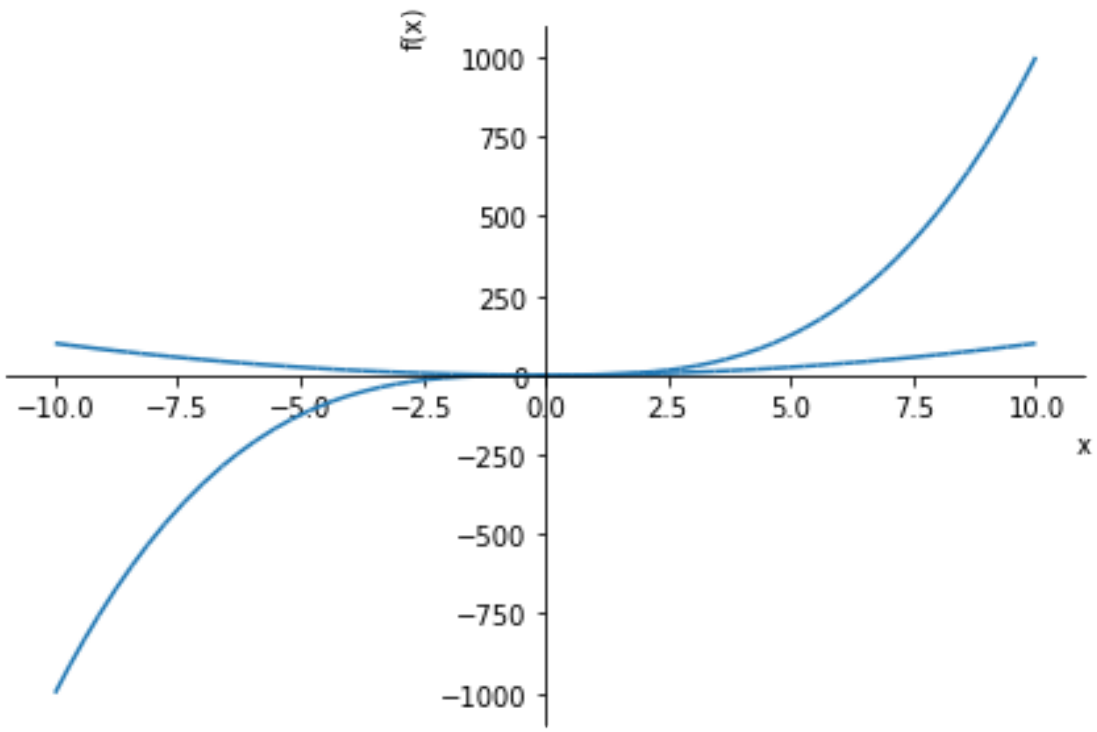
[3]:
```
plot(x**3, x**2)
```



[3]: <sympy.plotting.plot.Plot at 0x7f21f6b30ad0>

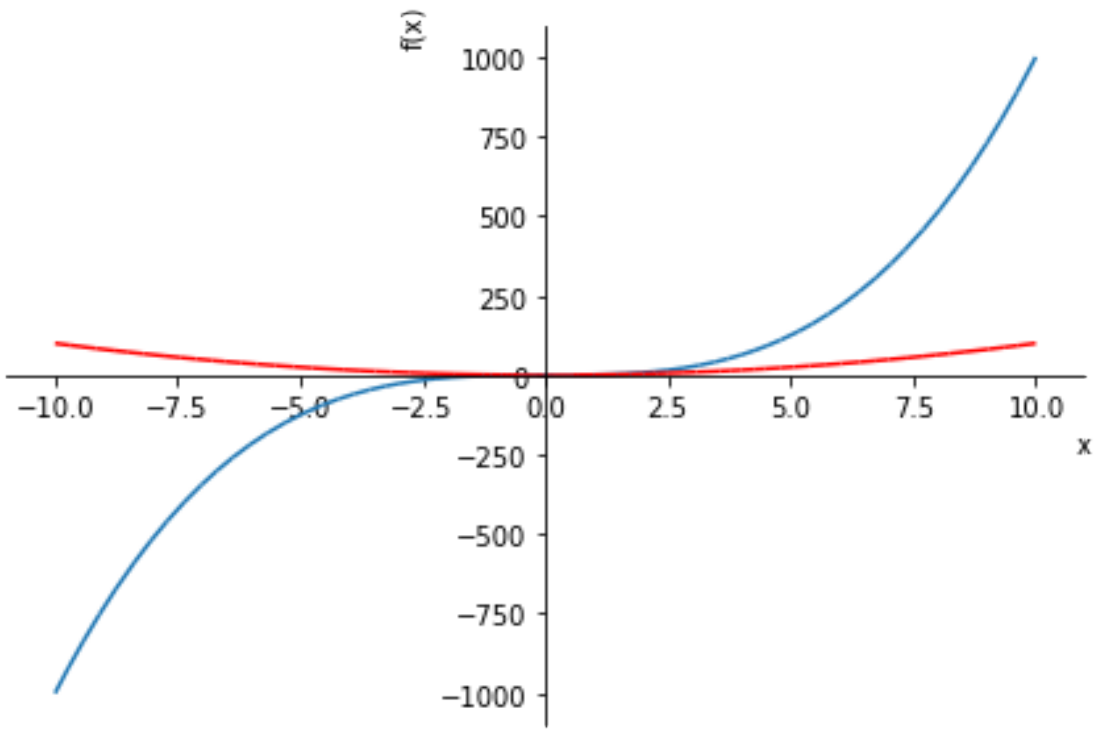
Obviamente, o gráfico acima está longe de ser o melhor possível. Mas com apenas uma função fizemos algo relevante. Mas, se armazenarmos, note o que podemos fazer.

[4]:
```
my_plot = plot(x**3, x**2, show=False) # show=False para não plotar.
my_plot.show() # .show() para exibir
```



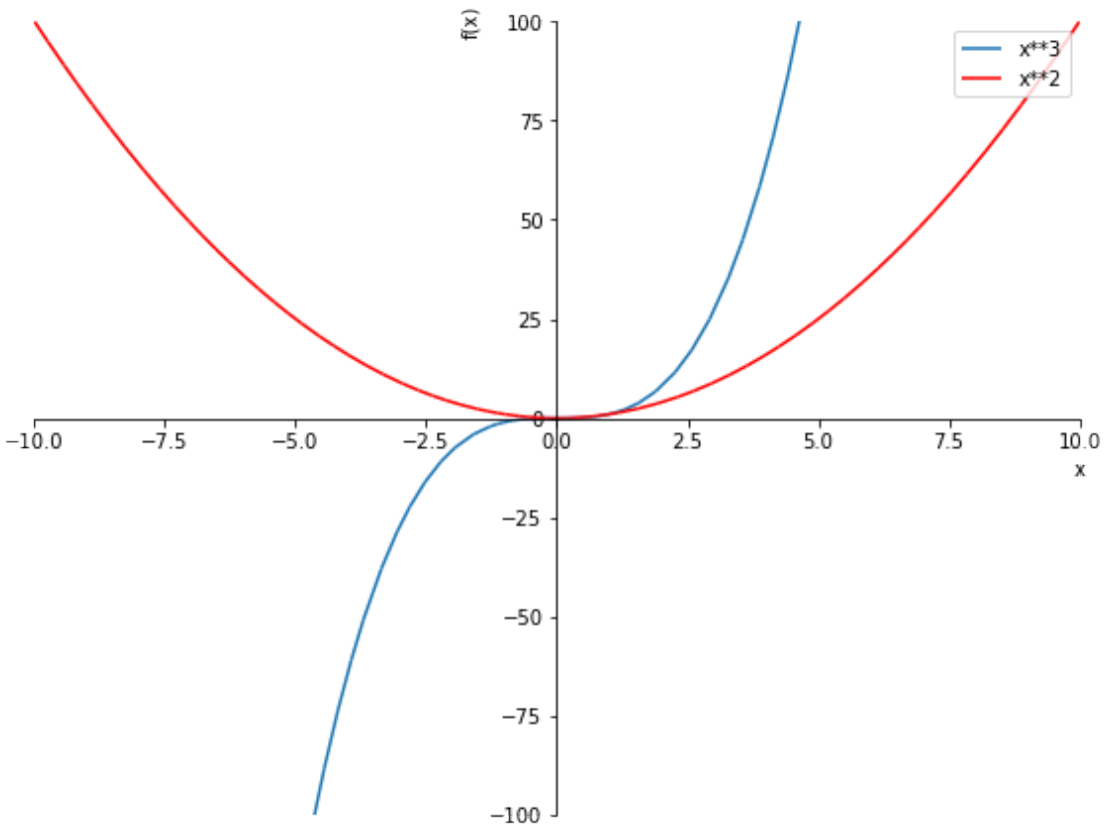
Por enquanto, nada mudou. Mas podemos ir trabalhando em seus atributos assim.

[5]:
```
my_plot[1].line_color = 'red' # x²
my_plot.show()
```



Estamos melhorando. Veja que my_plot armazena as funções em sequência. Trabalhamos em x^2 individualmente quando usamos my_plot[1].

[6]:
```
my_plot.legend = True # Legenda
my_plot.xlim = (-10,10) # Esse é o máximo por padrão, veremos como alterar isso␣
↪mais tarde
my_plot.ylim = (-100,100)
my_plot.size = (8,6) # Tamanho da figura
my_plot.show()
```



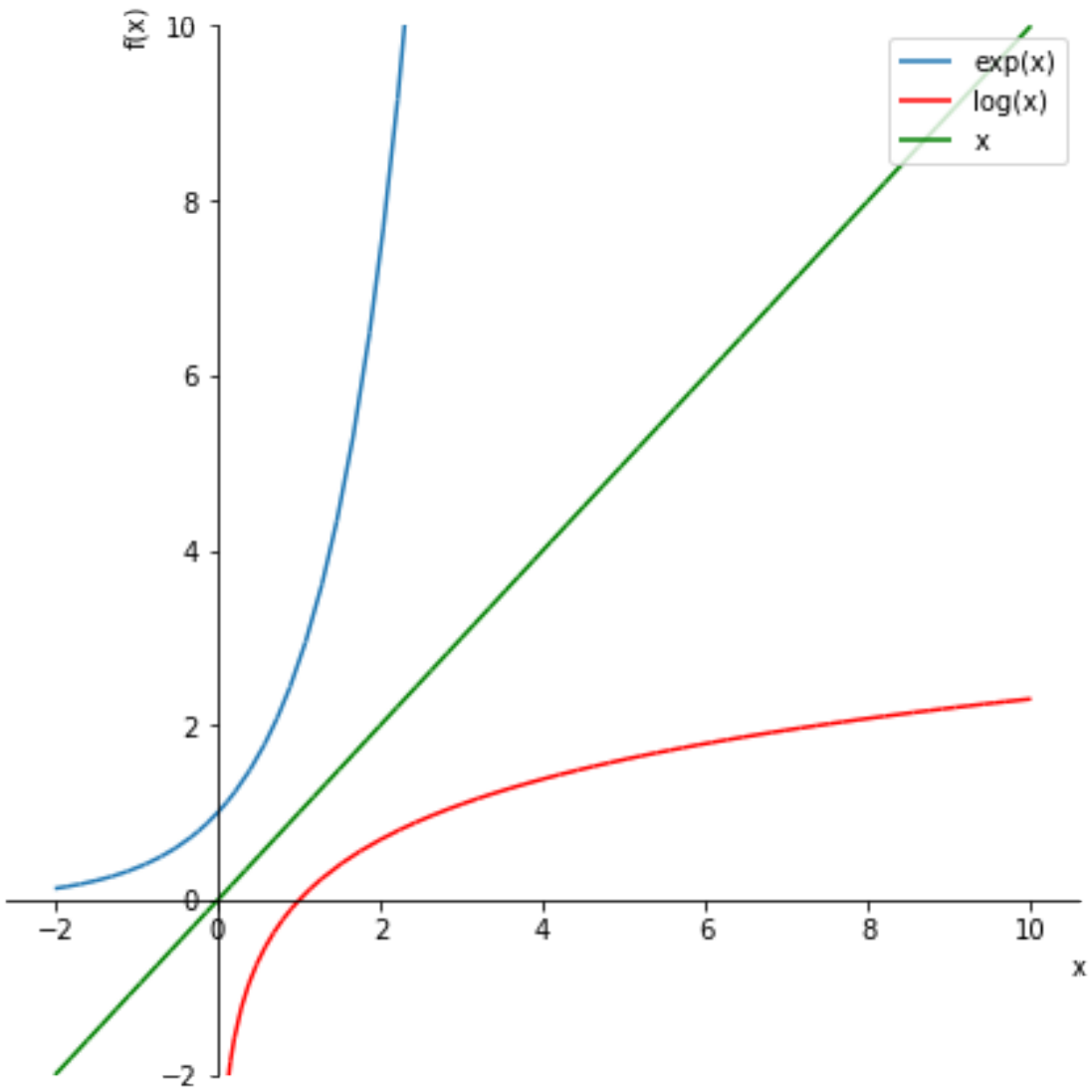
Acho que conseguimos um bom resultado. Contudo, você concorda que é um pouco cansativo acessar cada um desses valores e ir alterando, certo? E, se não definíssemos os limites em x para $[-10, 10]$, você veria que a função não continuaria. Isso ocorre pois não definimos o alcance de nosso plot ao criá-lo.

Aliado a isso, é possível criar dois plots diferentes e uní-los com o método .extend().

Com o que aprendemos podemos fazer, com um único comando, um bom plot de uma única função. Caso queiramos mais de uma (como fizemos acima), basta alterar as cores de suas linhas.

Caso você conheça matplotlib, é possível utilizar todos os parâmetros para anotações e etc. Não abordarei isso aqui pois acredito que foge do nosso objetivo, mas é algo interessante.

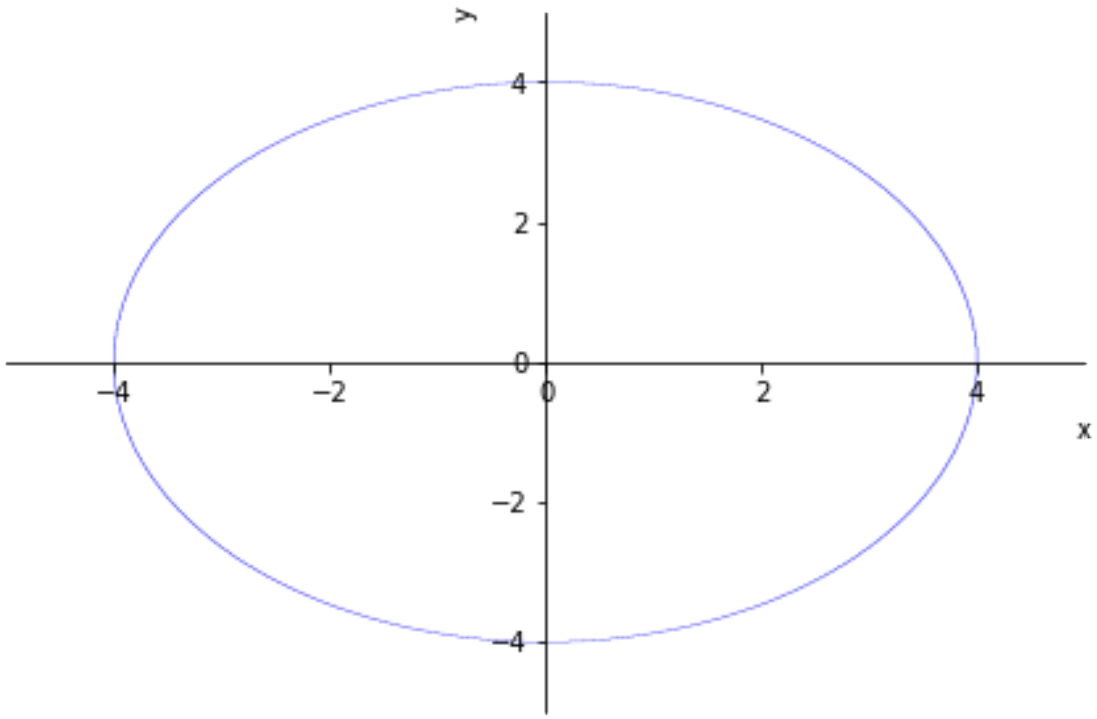
[7]:
```
p1 = plot(exp(x), log(x),x, (x, -2, 10), ylim = (-2,10), legend = True, size =␣
↪(6,6), show=False)
p1[1].line_color = 'red'
p1[2].line_color = 'green'
p1.show()
```



5.1.2 Equações Implícitas

Quando temos uma equação com variável implícita, utilizamos a plot_implicit(). Ela segue a mesma lógica de plot() em sua construção. Veja uma circunferência.

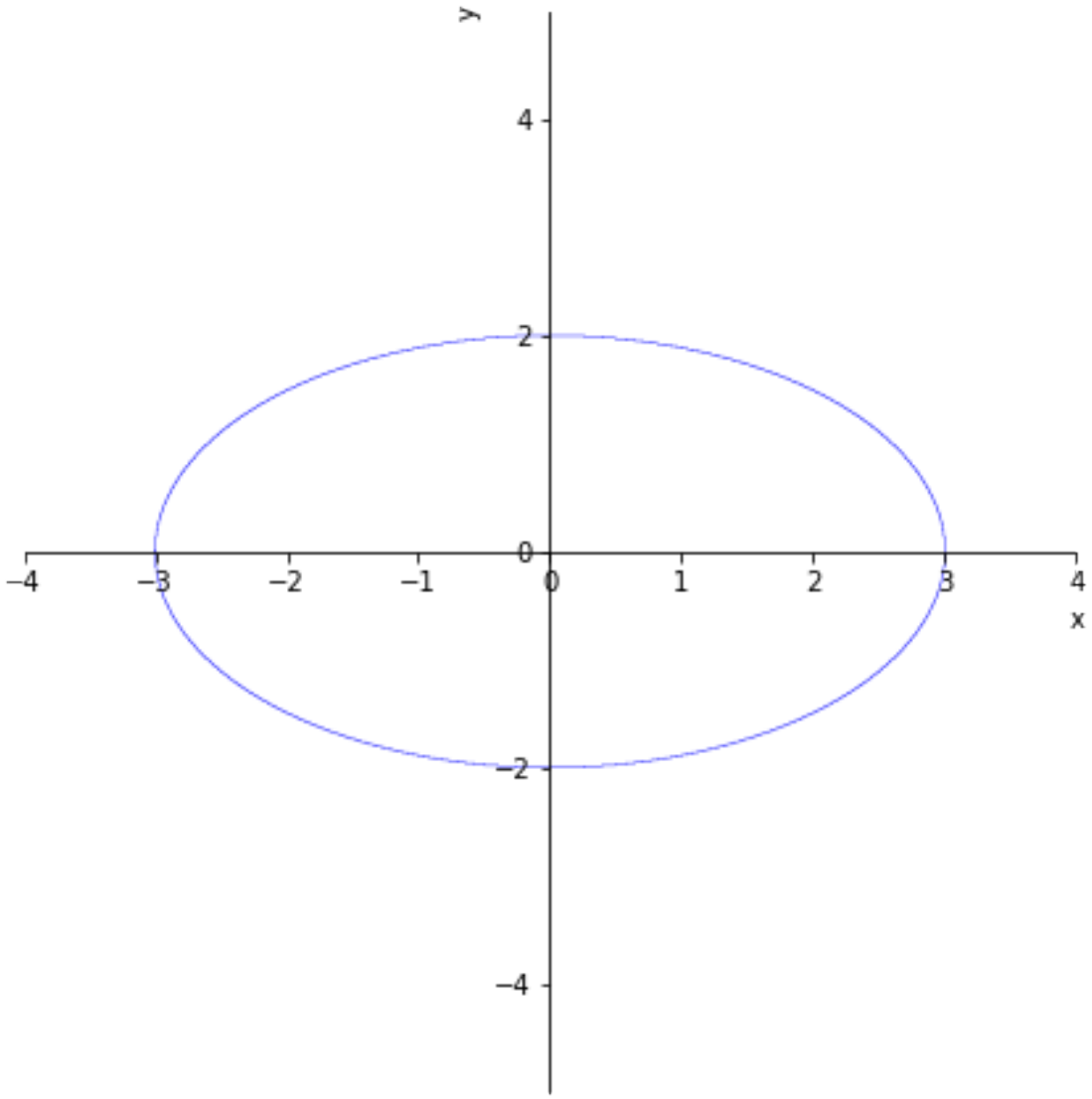
[8]:
```
cir = Eq(x**2 + y**2, 16) # R = 2
plot_implicit(cir)
```



[8]: <sympy.plotting.plot.Plot at 0x7f21f4425ad0>

Também podemos elaborá-lo.

[9]:
```
plot_implicit(Eq(x**2/9 + y**2/4, 1), (x, -4, 4), ylim = (-4,4), size = (6,6))␣
↪# Elípse
# axis = False, deixa sem os eixos
```

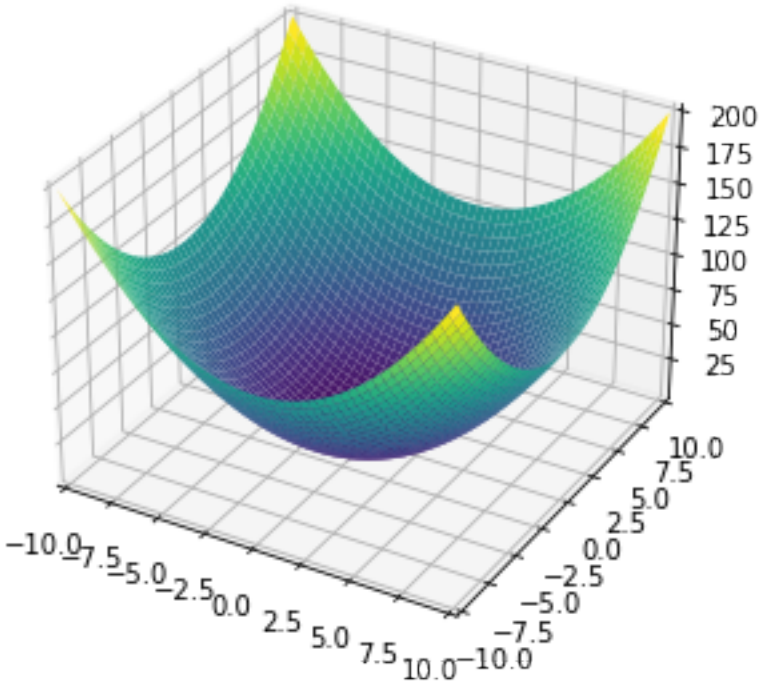


[9]: <sympy.plotting.plot.Plot at 0x7f21f4553b50>

5.2 Plot 3D

Utilizando funções de duas variáveis, nós conseguimos fazer plots em 3d com a função plot3d().

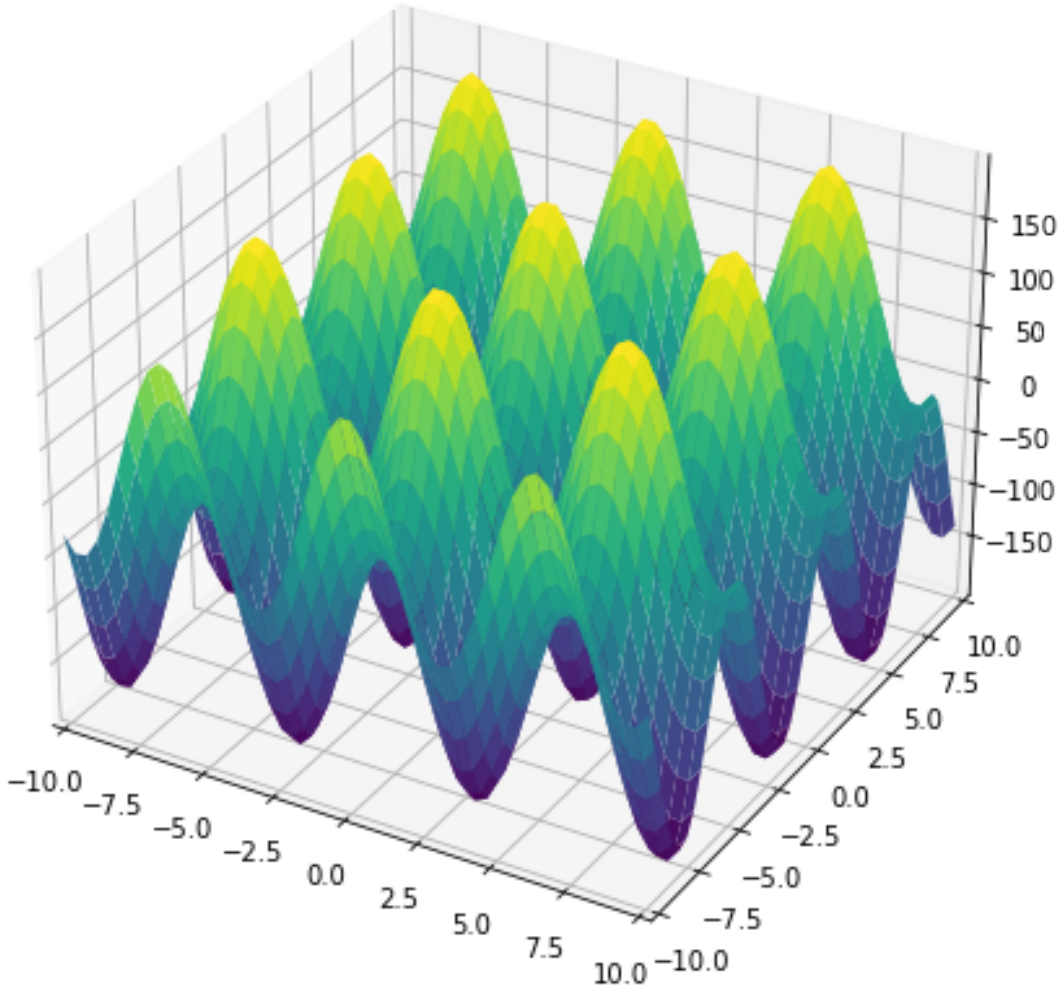
[10]:
```
from sympy.plotting import plot3d # Caso queira importá-la diretamente
plot3d(x**2 + y**2)
```



[10]: <sympy.plotting.plot.Plot at 0x7f21f43d7510>

Os parâmetros são muito parecidos com a função plot()

[11]:
```
plot3d(100*(sin(x) + cos(y)), size = (6,6))
```



[11]: <sympy.plotting.plot.Plot at 0x7f21f41bd8d0>

Além desses plots, há os plots paramétricos. Recomendo que dê uma olhada na documentação. No mais, é realmente simples criar plots no Sympy.

5.3 Exercícios

1. Faça o plot das seguintes funções, escolhendo os melhores valores para os parâmetros:

$$f(x) = 4x^2 - 3x + 26$$

$$g(x) = \log(x^2 + 10)$$

$$h(x) = 10x^4 + 7x^3 - 10x + 20$$

$$p(x) = \sin(x^2 - x \cdot \pi) + \cos\left(x + \frac{\pi}{6}\right)$$

2. Faça o plot das seguintes equações, escolhendo os melhores valores para os parâmetros:

$$2x - 5y = 20$$

$$x^2 + y^2 = 60$$

$$\frac{x^2}{10} - \frac{y^2}{8} = 1$$

$$2x - 5y^2 = 10$$

6 Extras

Como a principal intenção do curso é prepará-los para o uso de SymPy nas disciplinas que envolvem Cálculo, ao finalizar o capítulo anterior você deve estar pronto para resolver seus problemas utilizando esse módulo. Contudo, eu acredito que há muito a se falar sobre esse módulo. E, portanto, esse capítulo fará uma abordagem rápida sobre algumas coisas que são possíveis com ele.

6.1 Geometria

Isso mesmo, nós podemos resolver problemas de Geometria tanto de forma simbólica, como de forma numérica. A ideia principal não é ficar criando plots com o sistema completo, mas sim trabalhar matemáticamente (indo para o lado da Geometria Analítica).

Antes de começarmos essa seção e as próximas, faremos as devidas importações e definições:

[1]:
```
from sympy import *
from sympy.geometry import * # Importante garantir que foi importado␣
 ↪corretamente
x, y, z = symbols('x y z')
init_printing(use_unicode=True, use_latex='mathjax')
```

6.1.1 2D

Começando pela geometria em 2D, podemos seguir o processo de criar os pontos, as linhas (a partir dos pontos) e as formas 2D a partir dos segmentos. É bem simples e intuitivo, veja:

[2]:
```
O = Point(0,0)
A = Point(1,2)
B = Point(3,-4)
C = Point(-2, 3)
```

Você pode fazer as operações padrões entre pontos normalmente.

[3]:
```
B - A ## AB
```

[3]: $Point2D\left(2, -6\right)$

Contudo, o indicado é utilizar as classes, que já terão suas propriedades a fácil acesso.

[4]:
```
Segment(A,B) ## AB Simbólicamente
```

[4]: $Segment2D\left(Point2D\left(1, 2\right), Point2D\left(3, -4\right)\right)$

[5]:
```
AC = Segment(A,C)
AC.slope ## inclinação
```

[5]: $-\frac{1}{3}$

[6]:
```
AC.length ## comprimento
```

[6]: $\sqrt{10}$

[7]:
```
AC.midpoint ## ponto médio
```

[7]: $Point2D\left(-\frac{1}{2}, \frac{5}{2}\right)$

[8]:
```
AC.contains(A) ## Contém A?
```

[8]:
```
True
```

[9]:
```
AC.distance(B) ## Menor distância ao ponto B
```

[9]: $2\sqrt{10}$

Nós podemos criar linhas também

[10]:
```
Line(A,B)
```

[10]: $Line2D\left(Point2D\left(1, 2\right), Point2D\left(3, -4\right)\right)$

[11]:
```
l1 = Line(A,B)
l1.equation() ## Equação da reta = 0
```

[11]:
$$6x + 2y - 10$$

[12]:
```
l1.coefficients ## Coeficientes da reta
```

[12]:
$$(6,\ 2,\ -10)$$

Podemos criar uma reta ao dar um ponto inicial e uma inclinação, lembrando que: $y - y_0 = m(x - x_0)$

[13]:
```
l2 = Line(C, slope = 3)
l2.equation()
```

[13]:
$$-3x + y - 9$$

[14]:
```
l3 = l2.perpendicular_line(A) ## Retorna uma reta perpendicular que passa pelo
↪ponto dado
l3.equation()
```

[14]:
$$-x - 3y + 7$$

[15]:
```
l3.slope # -m^-1
```

[15]:
$$-\frac{1}{3}$$

E nós podemos ver a intersecção entre duas entidades geométricas.

[16]:
```
intersection(l2,l3)
```

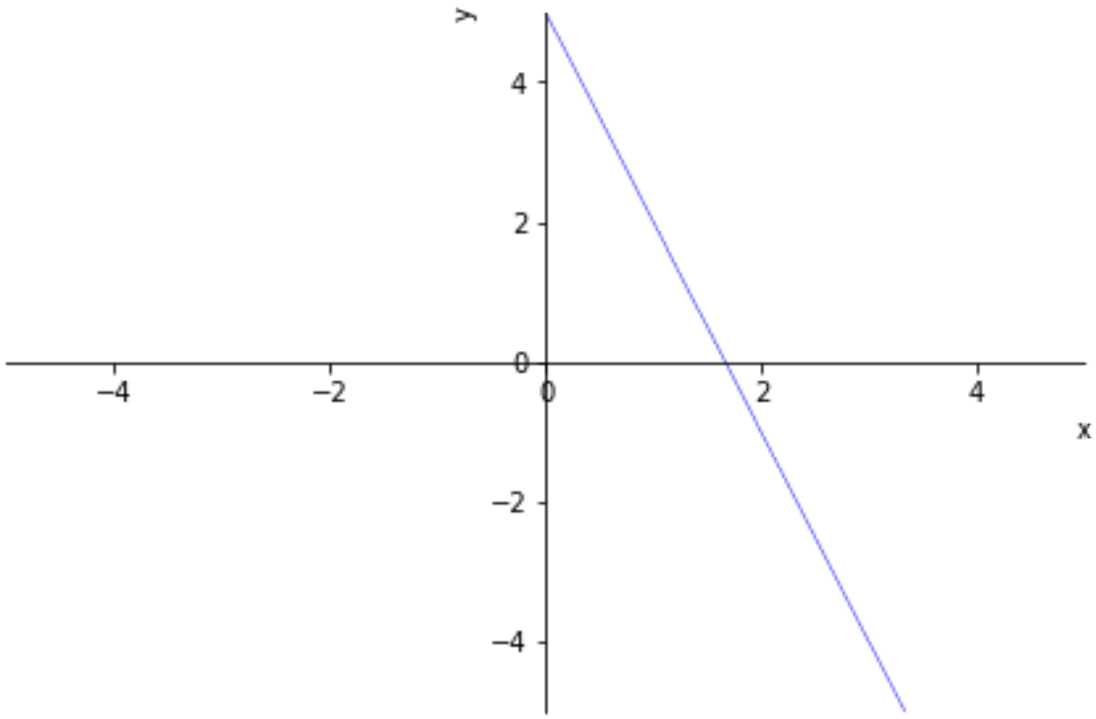
[16]:
$$[Point2D\,(-2, 3)]$$

[17]:
```
intersection(l1, l3) # Ponto A
```

[17]:
$$[Point2D\,(1, 2)]$$

Para plotar, de modo geral, fazemos o uso do que aprendemos no último capítulo, a função plot_implicit.

[18]:
```
plot_implicit(l1.equation())
```



[18]:
```
<sympy.plotting.plot.Plot at 0x7f9593fee1d0>
```

Podemos criar figuras geométricas e encontrar suas áreas e verficiar intersecções. Veja os exemplos:

[19]:
```
trig = Triangle(A,B,C) # Cria um Triângulo
trig
```

[19]: $Triangle\left(Point2D\left(1,2\right), Point2D\left(3,-4\right), Point2D\left(-2,3\right)\right)$

[20]:
```
trig.area ## Não utiliza valores absolutos
```

[20]: -8

[21]:
```
abs(trig.area) ## Correto
```

[21]: 8

[22]:
```
trig.perimeter ## Perímetro
```

[22]: $\sqrt{74} + 3\sqrt{10}$

[23]:
```
trig.orthocenter ## Centro Ortogonal
```

[23]: $Point2D\left(\frac{25}{4}, \frac{23}{4}\right)$

[24]:
```
trig.circumcenter ## Circuncentro
```

[24]:
$$Point2D\left(-\frac{17}{8}, -\frac{19}{8}\right)$$

[25]:
```
trig.altitudes ## Alturas
```

[25]:
$$\left\{Point2D\left(-2, 3\right) : Segment2D\left(Point2D\left(-2, 3\right), Point2D\left(\frac{2}{5}, \frac{19}{5}\right)\right),\ \ Point2D\left(1, 2\right) : Segment2D\left(Point2D\right.\right.$$

[26]:
```
trig.incircle ## Círculo interno
```

[26]:
$$Circle\left(Point2D\left(\frac{-\sqrt{10}+\sqrt{74}}{\sqrt{74}+3\sqrt{10}}, \frac{2\left(\sqrt{10}+\sqrt{74}\right)}{\sqrt{74}+3\sqrt{10}}\right), -\frac{16}{\sqrt{74}+3\sqrt{10}}\right)$$

[27]:
```
trig.incircle.equation()
```

[27]:
$$\left(x-\frac{-\sqrt{10}+\sqrt{74}}{\sqrt{74}+3\sqrt{10}}\right)^2+\left(y-\frac{2\left(\sqrt{10}+\sqrt{74}\right)}{\sqrt{74}+3\sqrt{10}}\right)^2-\frac{256}{\left(\sqrt{74}+3\sqrt{10}\right)^2}$$

[28]:
```
trig.bisectors() ## Bissetrizes
```

[28]:
$$\left\{Point2D\left(-2, 3\right) : Segment2D\left(Point2D\left(-2, 3\right), Point2D\left(\frac{11}{16}+\frac{\sqrt{185}}{16}, \frac{47}{16}-\frac{3\sqrt{185}}{16}\right)\right),\ \ Point2D\left(1, 2\right) : Se\right.$$

[29]:
```
trig.bisectors()[A] ## Bissetriz que passa no ponto A
```

[29]:
$$Segment2D\left(Point2D\left(1, 2\right), Point2D\left(-\frac{1}{3}, \frac{2}{3}\right)\right)$$

[30]:
```
trig.is_right() ## É triângulo retângulo?
```

[30]:
```
False
```

[31]:
```
trig.is_scalene() ## É triângulo escaleno?
```

[31]:
```
True
```

[32]:
```
circ = Circle(A, 3) ## Centro e Raio
circ
```

[32]:
$$Circle\left(Point2D\left(1, 2\right), 3\right)$$

[33]:
```
circ.equation()
```

[33]:
$$\left(x-1\right)^2+\left(y-2\right)^2-9$$

[34]:
```
circ.circumference
```

[34]:
$$6\pi$$

[35]:
```
circ.area
```

[35]:
$$9\pi$$

[36]:
```
intersection(trig,circ)
```

[36]:
$$\left[Point2D\left(-\frac{19}{37}+\frac{5\sqrt{410}}{74},\frac{34}{37}-\frac{7\sqrt{410}}{74}\right),\ Point2D\left(1-\frac{9\sqrt{10}}{10},\frac{3\sqrt{10}}{10}+2\right),\ Point2D\left(\frac{3\sqrt{10}}{10}+1,2-\frac{9\sqrt{10}}{10}\right)\right.$$

[37]:
```
elips = Ellipse(B, 3, 2) ## Centro, Raio Horizontal, Raio Vertical
elips
```

[37]:
$$Ellipse\left(Point2D\left(3,-4\right),3,2\right)$$

[38]:
```
elips.equation()
```

[38]:
$$\left(\frac{x}{3}-1\right)^2+\left(\frac{y}{2}+2\right)^2-1$$

[39]:
```
elips.circumference ## Não há formulas
```

[39]:
$$12E\left(\frac{5}{9}\right)$$

[40]:
```
elips.circumference.evalf() ## Valor numérico
```

[40]:
$$15.8654395892906$$

[41]:
```
elips.area
```

[41]:
$$6\pi$$

[42]:
```
elips.eccentricity
```

[42]:
$$\frac{\sqrt{5}}{3}$$

[43]:
```
elips.foci ## Focos
```

[43]:
$$\left(Point2D\left(3-\sqrt{5},-4\right),\ Point2D\left(\sqrt{5}+3,-4\right)\right)$$

[44]:
```
elips.focus_distance ## Distância Focal
```

[44]:
$$\sqrt{5}$$

[45]:
```
D = Point(0,10)
quad = Polygon(A,B,C,D) ## Criando Polígono de N vértices
quad
```

[45]:

$Polygon\left(Point2D\left(1,2\right),Point2D\left(3,-4\right),Point2D\left(-2,3\right),Point2D\left(0,10\right)\right)$

[46]:
```
abs(quad.area)
```

[46]: $\frac{39}{2}$

[47]:
```
quad.angles
```

[47]: $\left\{Point2D\left(-2,3\right):-\operatorname{acos}\left(-\frac{39\sqrt{3922}}{3922}\right)+2\pi,\ \ Point2D\left(0,10\right):-\operatorname{acos}\left(\frac{54\sqrt{3445}}{3445}\right)+2\pi,\ \ Point2D\left(1,2\right):\text{aco}\right.$

[48]:
```
quad.angles[A] ## No ponto A
```

[48]: $\operatorname{acos}\left(-\frac{5\sqrt{26}}{26}\right)$

[49]:
```
from sympy.physics.units import degree ## Importação das unidades
(quad.angles[A]/degree.scale_factor).evalf() ## Transforma em Graus
```

[49]: 168.69006752598

[50]:
```
reg = RegularPolygon(A,1,4) # Centro, Raio, Qtd. Lados
reg
```

[50]: $RegularPolygon\left(Point2D\left(1,2\right),1,4,0\right)$

[51]:
```
reg.angles # Retângulo
```

[51]: $\left\{Point2D\left(0,2\right):\frac{\pi}{2},\ \ Point2D\left(1,1\right):\frac{\pi}{2},\ \ Point2D\left(1,3\right):\frac{\pi}{2},\ \ Point2D\left(2,2\right):\frac{\pi}{2}\right\}$

[52]:
```
reg.vertices # Vértices
```

[52]: $\left[Point2D\left(2,2\right),\ \ Point2D\left(1,3\right),\ \ Point2D\left(0,2\right),\ \ Point2D\left(1,1\right)\right]$

Para finalizar com a Geometria, é importante relembrar que é possível fazer tudo isso com valores simbólicos. Por exemplo, um quadrado em função de um lado x:

[53]:
```
sim_quad = Polygon(Point(x/2, x/2), Point(-x/2, x/2),Point(-x/2, -x/2),Point(x/
↪2, -x/2))
sim_quad
```

[53]: $Polygon\left(Point2D\left(\frac{x}{2},\frac{x}{2}\right),Point2D\left(-\frac{x}{2},\frac{x}{2}\right),Point2D\left(-\frac{x}{2},-\frac{x}{2}\right),Point2D\left(\frac{x}{2},-\frac{x}{2}\right)\right)$

[54]:
```
sim_quad.area
```

[54]: x^2

6.1.2 3D

Para a terceira dimensão, podemos utilizar os Pontos com três coordenadas para gerar nossas formas.

[55]:
```
M = Point(1, 2, 3)
N = Point(-2, 3, 4)
P = Point(5, -8, 10)
Line(M,N)
```

[55]:
$$Line3D\left(Point3D\left(1,2,3\right),Point3D\left(-2,3,4\right)\right)$$

[56]:
```
Line(M,N).equation()
```

[56]:
$$(x + 3y - 7,\ \ x + 3z - 10)$$

[57]:
```
Plane(M,N,P) # Plano
```

[57]:
$$Plane\left(Point3D\left(1,2,3\right),(17,\ \ 25,\ \ 26)\right)$$

[58]:
```
Plane(M,N,P).equation()
```

[58]:
$$17x + 25y + 26z - 145$$

6.2 Reações em Vigas (Mecânica)

Nós podemos analisar as tensões em vigas utilizando o sympy.physics.continuum_mechanics. Caso você procure aplicar carregamento em uma viga e então avaliar as reações e seus gráficos, certamente isso vai te auxiliar.

Veja um exemplo (no caso, para fazer sentido, leva os mesmos valores de um exemplo da documentação):

[84]:
```
from sympy.physics.continuum_mechanics.beam import Beam
E, I = symbols('E I') ## Símbolos para o Módulo de Elasticidade e o Momento de␣
 ↪Inércia
R1, R2 = symbols('R1 R2') ## Símbolos para as forças
b = Beam(50, 20, 30) ## Criando a viga (comprimento, E, I)
b.apply_load(R1, 0, -1) ## Aplicando carregamentos (intensidade, início, ordem)
###
### Momentos, order = -2
### Forças Pontuais, order =-1
### Forças distribuídas linearmente, order = 0
### Veja os outros na documentação
###
b.apply_load(R1, 10, -1)
b.apply_load(R2, 30, -1)
b.apply_load(90, 5, 0, 23)
```

```
b.apply_load(10, 30, 1, 50)
b.load ## Carregamento
```

[84]:
$$R_1\langle x\rangle^{-1} + R_1\langle x-10\rangle^{-1} + R_2\langle x-30\rangle^{-1} + 90\langle x-5\rangle^{0} - 90\langle x-23\rangle^{0} + 10\langle x-30\rangle^{1} - 200\langle x-50\rangle^{0} - 10\langle x-50\rangle^{1}$$

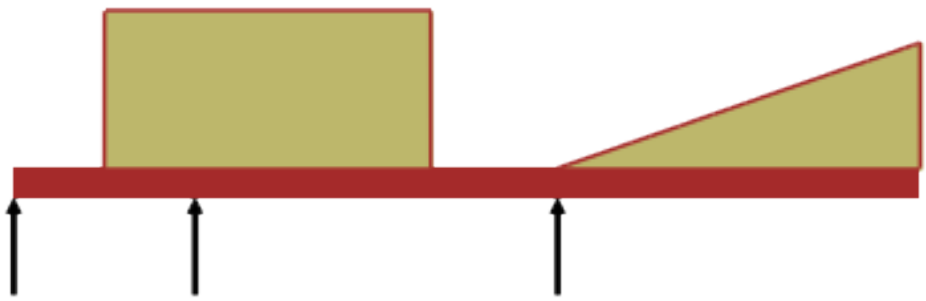
[81]:
```
b.shear_force() ## Força Cortante
```

[81]:
$$-M_0\langle x\rangle^{-1} - R_0\langle x\rangle^{0} - R_{20}\langle x-20\rangle^{0} - R_{50}\langle x-50\rangle^{0} - \frac{224\langle x\rangle^{0}}{15} - 90\langle x-5\rangle^{1} - \frac{224\langle x-10\rangle^{0}}{15} + 90\langle x-23\rangle^{1} + \frac{54748\langle x-30\rangle^{0}}{15} - 5\langle x-30\rangle^{2} + 200\langle x-50\rangle^{1} + 5\langle x-50\rangle^{2}$$

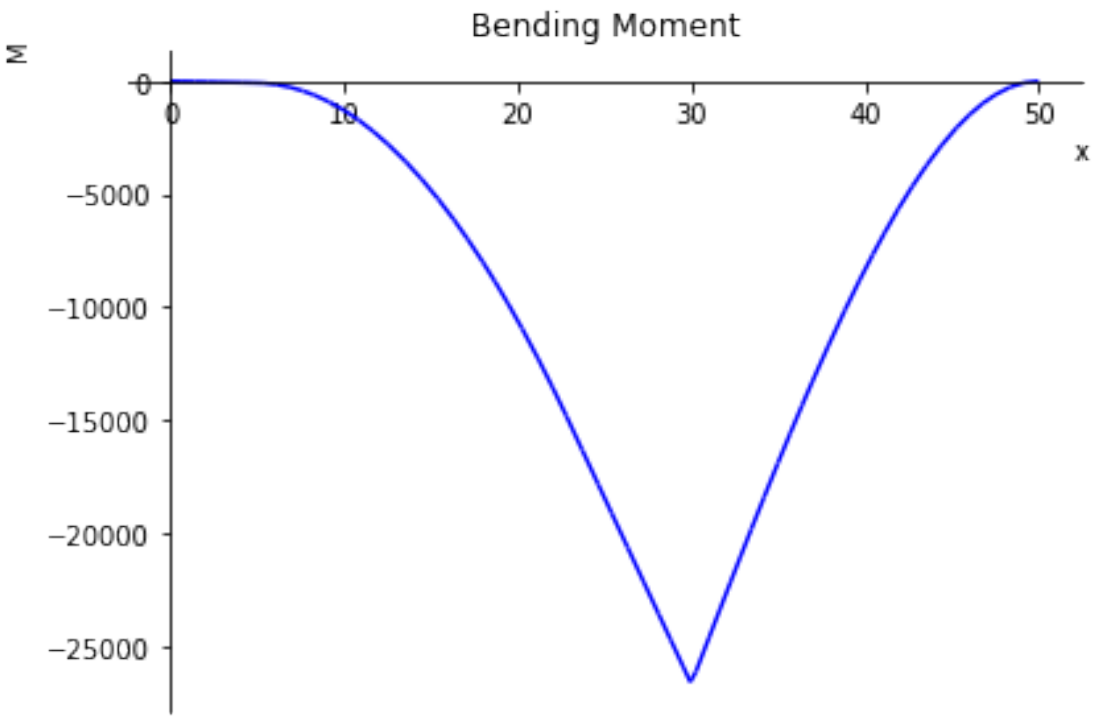
[82]:
```
b.bending_moment() ## Momento Fletor
```

[82]:
$$-M_0\langle x\rangle^{0} - R_0\langle x\rangle^{1} - R_{20}\langle x-20\rangle^{1} - R_{50}\langle x-50\rangle^{1} - \frac{224\langle x\rangle^{1}}{15} - 45\langle x-5\rangle^{2} - \frac{224\langle x-10\rangle^{1}}{15} + 45\langle x-23\rangle^{2} + \frac{54748\langle x-30\rangle^{1}}{15} - \frac{5\langle x-30\rangle^{3}}{3} + 100\langle x-50\rangle^{2} + \frac{5\langle x-50\rangle^{3}}{3}$$

[86]:
```
p = b.draw()
p.show() ## Ilustração Gráfica
```

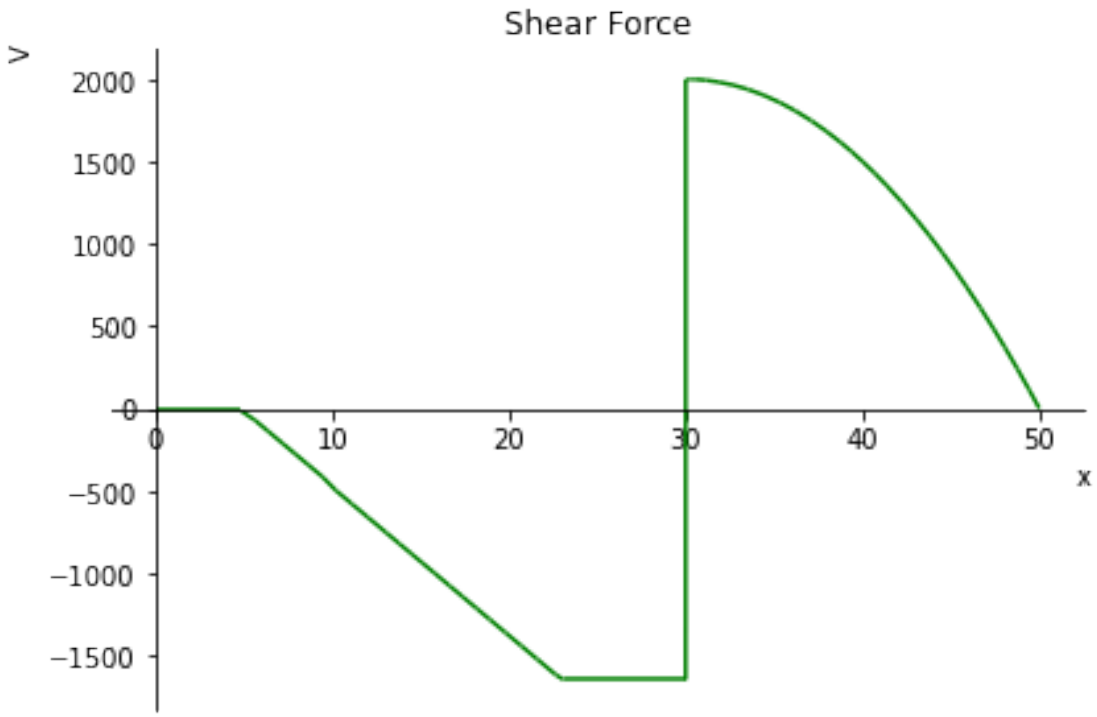


[85]:
```
b.solve_for_reaction_loads(R1, R2) ## Solucionando
b.plot_bending_moment() ## Plot Momento Fletor
```



[85]: <sympy.plotting.plot.Plot at 0x7f9576b37cd0>

[87]:
```
b.plot_shear_force() ## Plotando Força Cortante
```



[87]:
```
<sympy.plotting.plot.Plot at 0x7f9576c90d50>
```

[88]:
```
b.apply_support(50, 'pin') ## Criando apoios
b.apply_support(0, 'fixed')
b.apply_support(20, 'roller')
b.load
```

[88]:
$$M_0\langle x\rangle^{-2} + R_0\langle x\rangle^{-1} + R_{20}\langle x-20\rangle^{-1} + R_{50}\langle x-50\rangle^{-1} + \frac{224\langle x\rangle^{-1}}{15} + 90\langle x-5\rangle^{0} + \frac{224\langle x-10\rangle^{-1}}{15} - 90\langle x-23\rangle^{0} - \frac{54748\langle x-30\rangle^{-1}}{15} + 10\langle x-30\rangle^{1} - 200\langle x-50\rangle^{0} - 10\langle x-50\rangle^{1}$$

[89]:
```
p = b.draw()
p.show()
```