Práctica 0 (23, 24 e 26 de setembro de 2024)

1 Introdución

Con esta práctica pretendemos familiarizarnos coa contorna de Jupyter Notebook para traballar coa linguaxe de programación Python. Para isto, imos aprender a utilizar algúns comandos de tres módulos de Python que serán de utilidade para realizar os exercicios e as prácticas da materia de Cálculo e Análise Numérico.

- Comezaremos con **SymPy**, que nos permitirá realizar cálculos simbólicos en Python. Permite traballar con expresións matemáticas como símbolos e fórmulas en lugar de números para poder calcular límites, derivadas ou integrais. Documentación deste módulo podemos atopala en https://docs.sympy.org/latest/index.html. Para poder utilizar este módulo debemos cargalo do seguinte xeito:

[1]:
```
import sympy as sp
```

- A continuación utilizaremos **NumPy**, que é un módulo fundamental para a computación numérica en Python. Ofrece funcionalidades para realizar operacións matriciais e cálculos numéricos eficientes. É esencial para a análise de datos e a resolución de problemas matemáticos e científicos. Documentación deste módulo podemos atopala en https://numpy.org/doc/. Para poder utilizar este módulo debemos cargalo do seguinte xeito:

[2]:
```
import numpy as np
```

- Finalmente, introducimos **Matplotlib**, que é un módulo que permite crear gráficos e visualizacións de datos de maneira sinxela en Python. Para poder utilizar este módulo debemos cargalo. Documentación para este módulo podemos atopala en https://matplotlib.org/stable/index.html. Ademais, como o imos utilizar a través da interface `pyplot`, tamén será necesario importala do seguinte xeito:

[3]:
```
import matplotlib as mp
import matplotlib.pyplot as plt
```

2 Módulo SymPy

2.1 Variables simbólicas

Para traballar con variables simbólicas é preciso definilas primeiro, farémolo usando a función `sp.Symbol`:

[4]:
```
x = sp.Symbol('x') # Define a variable simbólica x
y = sp.Symbol('y') # Define a variable simbólica y
f = 3*x + 5*y      # Define a expresión simbólica f
print(f)           # Imprimimos a expresión simbólica
```

```
display(f) # Imprimimos a expresión simbólica nun formato similar a como a␣
 ↪escribimos en papel

a, b, c = sp.symbols('a:c') # Define como simbólicas as variables a, b, c.
expresion = a**3 + b**2 + c**b
print(expresion)
display(expresion)
```

```
3*x + 5*y
```

$3x + 5y$

```
a**3 + b**2 + c**b
```

$a^3 + b^2 + c^b$

2.2 Operacións

Como acabamos de ver podemos realizar operacións coas variables simbólicas:

- `+` para suma
- `-` para resta
- `*` para multiplicación
- `/` para división
- `**` para potenciación

Estas operacións realízanse coa orde de prioridade usual, primeiro potencia, despois multiplicación ou división e finalmente suma ou resta. Para cambiar a orde debemos usar parénteses.

Por claridade na implementación e nos cálculos, será habitual que o nome da variable simbólica e o nome do obxecto **SymPy** no que se almacena coincidan, pero isto non ten por que ser así:

[5]:
```
a = sp.Symbol('x')
print(a)
a.name
```

```
x
```

[5]:
```
'x'
```

Debemos ter claro que as variables `x` ou `y`, definidas anteriormente, non son números. Tódalas variables simbólicas son obxetos da clase `sp.Symbol` e os seus atributos e métodos son completamente diferentes aos das variables numéricas ou dos vectores de **NumPy**. Podemos comprobar os métodos e atributos que teñen estes obxetos executando os seguintes comandos (que sirven para ver os métodos e atributos de calquera obxecto):

[6]:
```
print(type(x))
dir(x)
```

```
<class 'sympy.core.symbol.Symbol'>
```

2.3 Constantes enteiras e números racionais

Con **SymPy** podemos definir constantes enteiras ou números racionais de forma simbólica, usando os comandos `sp.Integer` ou `sp.Rational`, respectivamente. Por exemplo, podemos definir a constante simbólica 1/3. Se fixésemos o mesmo con números representados por defecto en Python, obteríamos resultados moi diferentes. Observa tamén, a diferenza que existe entre o tipo de dato asignado no espazo de traballo.

[7]:
```
a = sp.Rational('1/3')
b = sp.Integer('1')/sp.Integer('3')
c = 1/3
d = 1.0/3.0
print('a: ',a) # Podemos imprimir literalmente o texto que vai entre ''
print('b: ',b)
print('c: ',c)
print('d: ',d)
print(type(a))
print(type(b))
print(type(c))
print(type(d))
```

```
a:  1/3
b:  1/3
c:  0.3333333333333333
d:  0.3333333333333333
<class 'sympy.core.numbers.Rational'>
<class 'sympy.core.numbers.Rational'>
<class 'float'>
<class 'float'>
```

2.3.1 Exercicio.

Podes probar a realizar distintas operacións con variables, constantes, ... na seguinte liña de código.

Ten en conta que terás xa definidas as utilizadas nas anteriores liñas, sempre que as executases. Noutro caso deberás definir os obxectos que desexes usar.

Ollo! Se utilizamos unha mesma variable para asignarlle distintos valores ao largo do código, esta vai ter o último valor asignado.

[8]:
```
2*a+b
```

[8]:
1

- Outra forma de manexar valores constantes con obxectos do módulo **SymPy** é usar a función `sp.S` que transforma os valores dados en obxectos do módulo **SymPy**, como se pode ver ao imprimir o tipo de dato con `type`.
- Se precisamos obter o valor numérico dunha expresión simbólica, podemos usar a función `sp.N` ou ben, directamente, `float`:

[9]:
```
print('Sen usar sp.S:')
a = 2
print('a = ', a, ', tipo de dato: ', type(a))
a = sp.S(2)
b = sp.S(6)
c = a/b
d = sp.N(c)
e = float(c)
print('Usando sp.S:')
print('a = ', a, ', tipo de dato: ', type(a))
print('b = ', b, ', tipo de dato: ', type(b))
print('c = ', c, ', tipo de dato: ', type(c))
print('d = ', d, ', tipo de dato: ', type(d))
print('e = ', e, ', tipo de dato: ', type(e))
print('e =','{0:.4f}'.format(e)) # Podedes modificar os formatos de impresión
```

```
Sen usar sp.S:
a =  2 , tipo de dato:  <class 'int'>
Usando sp.S:
a =  2 , tipo de dato:  <class 'sympy.core.numbers.Integer'>
b =  6 , tipo de dato:  <class 'sympy.core.numbers.Integer'>
c =  1/3 , tipo de dato:  <class 'sympy.core.numbers.Rational'>
d =  0.333333333333333 , tipo de dato:  <class 'sympy.core.numbers.Float'>
e =  0.3333333333333333 , tipo de dato:  <class 'float'>
e = 0.3333
```

- Por exemplo as constantes π ou e están definidas como constantes simbólicas. Do mesmo xeito, para operar con variables ou constantes simbólicas, debemos utilizar funcións que sexan capaces de manipular este tipo de obxectos, todas elas implementadas no módulo **SymPy** (por exemplo, `sp.sin`, `sp.cos`, `sp.tan`, `sp.log`, `sp.exp`, ...).

[10]:
```
p=sp.pi # Definición da constante pi
print(sp.cos(p))

e = sp.E # Definición do número e
print(sp.log(e))
print(sp.log(sp.exp(-4)))
```

```
-1
1
-4
```

- Tamén podemos definir as nosas propias funcións. A continuación mostramos un exemplo dunha función para calcular o logaritmo dun número en base a. Usamos os comando `def` para definir o encabezado da función (nome que lle poñemos, log_a, e variables de entrada, base e b). Despois do comando `return` indicamos as variables que queremos que nos devolva a función unha vez chamada, neste caso o resultado desexado.

[11]:
```
def log_a(base, b):
    y = sp.log(b)/sp.log(base)
    return float(y)
```

[12]:
```
log_a(10,100), log_a(3,81) # Chamamos á función creada por nós
```

[12]:
```
(2.0, 4.0)
```

[13]:
```
float(sp.log(100,10)), float(sp.log(81,3)) # Xa estaba implementado en Sympy.␣
↪Coidado coa orde das entradas.
```

[13]:
```
(2.0, 4.0)
```

2.3.2 Exercicio

Podedes probar a definir outras funcións na seguinte liña de código.

[ ]:
```

```

2.4 Hipóteses sobre as variables

Cando se define unha variable simbólica, pódeselle asignar certa información adicional sobre o tipo de valores que pode alcanzar ou as hipóteses que se lle aplicarán. Por exemplo, podemos decidir antes de realizar calquera cálculo se a variable toma valores enteiros ou reais, se é positiva ou negativa, maior que un certo número, ... Este tipo de información engádese no momento da definición da variable simbólica como un argumento opcional.

[14]:
```
x = sp.Symbol('x', nonnegative = True) # A raíz cadrada dun número non negativo␣
↪é real
y = sp.sqrt(x)
print(y.is_real)

x = sp.Symbol('x', integer = True) # A potencia dun número enteiro é enteira
y = x**sp.S(2)
print(y.is_integer)

a = sp.Symbol('a')
b = sp.sqrt(a)
print(b.is_real)

a = sp.Symbol('a')
b = a**sp.S(2)
print(b.is_integer)
```

```
True
True
None
None
```

- Tamén é posible realizar comprobacións sobre se certas desigualdades son verdadeiras ou non, sempre e cando se teña coidado coas hipóteses que se establezan ao definir as variables simbólicas. A resposta será `True` ou `False` dependendo de veracidade ou falsidade da expresión.

[15]:
```
x = sp.Symbol('x', real = True)
p = sp.Symbol('p', positive = True)
q = sp.Symbol('q', real = True)
y = sp.Abs(x) + p   # sp.Abs é a función valor absoluto
z = sp.Abs(x) + q
print(y > 0) # Comprobamos se y > 0
print(z > 0)
```

```
True
q + Abs(x) > 0
```

2.4.1 Exercicio

Porque a segunda expresión no anterior exemplo non é verdadeira nin falsa?

2.5 Expresións simbólicas

A partir das variables simbólicas podemos definir expresións matemáticas para manipulalas, factorizalas, expandilas, simplificalas ou imprimilas de maneira semellante ao que faríamos con lapis e papel.

[16]:
```
x,y = sp.symbols('x,y', real=True)
expr = (x-3)*(x-3)**2*(y-2)
expr_long = sp.expand(expr) # Expandir expresión

print('Forma 1:\n ',expr_long) # Imprimir de forma estándar
print('Forma 2: ')
display(expr_long) # Impresión mellorada

expr_short = sp.factor(expr)
print('Factorizada: ',expr_short) # Factorizar expresión

expr2 = -3+(x**2-6*x+9)/(x-3)
expr2_simple = sp.simplify(expr2) # Simplificar expresión
print('Segunda expresión:')
display(expr2)
print('Simplificada: ', expr2_simple)
```

```
Forma 1:
  x**3*y - 2*x**3 - 9*x**2*y + 18*x**2 + 27*x*y - 54*x - 27*y + 54
Forma 2:
```

$$x^3y - 2x^3 - 9x^2y + 18x^2 + 27xy - 54x - 27y + 54$$

```
Factorizada:  (x - 3)**3*(y - 2)
Segunda expresión:
```

$$-3 + \frac{x^2 - 6x + 9}{x - 3}$$

```
Simplificada:  x - 6
```

2.5.1 Substitucións

Dada unha expresión en **SymPy**, tamén se pode manipular, substituíndo unhas variables simbólicas por outras ou mesmo por constantes. Para facer este tipo de substitucións utilízase a función `subs`. Os valores que se utilizan na substitución veñen definidos por un dicionario de Python:

[17]:
```
x,y = sp.symbols('x,y', real=True)
expr = x*x + x*y + y*x + y*y
res = expr.subs({x:1, y:2}) # Substitutición das variables simbólicas por␣
 ↪constantes
print(res)

expr_sub = expr.subs({x:1-y}) # Substitución da variable simbólica por unha␣
 ↪expresión
display(expr_sub)
print('Simplificada:', sp.simplify(expr_sub))
```

```
9
```

$$y^2 + 2y\,(1 - y) + (1 - y)^2$$

```
Simplificada: 1
```

2.5.2 Exercicio

1. Define a expresión dada pola suma de $a + a^2 + a^3 + \ldots + a^N$, onde a é unha variable real arbitraria e N un valor enteiro positivo.
2. Cal é o valor exacto da expresión anterior cando $N = 15$ e $a = 5/6$? Cal é o valor numérico en coma flotante?

[18]:
```
a = sp.symbols('a', real=True) # Variable real
N = sp.symbols('N', integer=True) # Variable enteira
n = sp.symbols('n', integer=True) # Variable enteira

faN = sp.Sum(a**n, (n,1,N))          # Sumatorio
display(faN)

# Exercicio 2
```

$$\sum_{n=1}^{N} a^n$$

2.6 Funcións

Para simplificar a sustitución nas expresións podemos definir funcións en **SymPy** utilizando `sp.Lambda`. As funcións así definidas, poden ser avaliadas nun punto ao igual que se fai con calquera outra función xa predefinida en **SymPy**. Deste xeito non precisamos utilizar a sustitución co comando `subs`.

[19]:
```
x = sp.symbols('x', real=True) # Define a variable simbólica real x
f_expr = x*sp.cos(4*x) # Isto é unha expresión
f = sp.Lambda((x),f_expr) # Creamos a función "f" a partir da expresión "f_expr"
display(f)
print('Valor de f(2)=',f(2)) # Avaliación da función
print('Valor de f(2)=',f_expr.subs(x,2)) # Substitución na expresión
print(f(x)==f_expr) # Comprobamos que son o mesmo
```

$$(x \mapsto x\cos\left(4x\right))$$

```
Valor de f(2)= 2*cos(8)
Valor de f(2)= 2*cos(8)
True
```

2.6.1 Exercicio

Podes practicar definindo outras expresións e funcións na seguinte liña de código.

[ ]:
```

```

2.7 Resolución ecuacións

Dada unha expresión en **Sympy**, podemos formular e resolver unha ecuación coa mesma usando a función `solve`, que entende que a expresión se iguala a 0 (é dicir, debemos pasar todo ao primero membro). Por exemplo, se queremos resolver $e^{x+1} = 5$ debemos pensar en $e^{x+1} - 5 = 0$:

[20]:
```
x = sp.Symbol('x', real = True)
expr = sp.exp(x+1)-5
solucion = sp.solve(expr,x)
print("Queremos resolver a ecuación exp(x+1)=5")
print("Solución: ", solucion)
# Nota: en casos sixelos, como este, podemos escribir directamente
# solucion = sp.solve(sp.exp(x+1)-5,x)
```

```
Queremos resolver a ecuación exp(x+1)=5
Solución:  [-1 + log(5)]
```

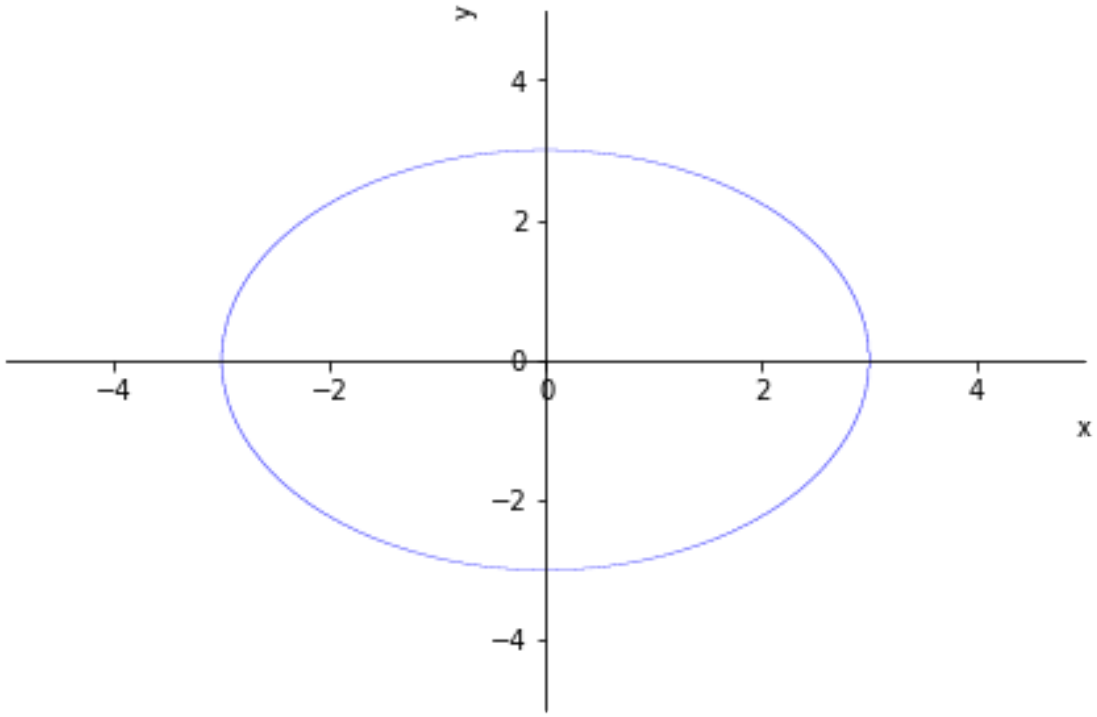
2.8 Representación gráfica

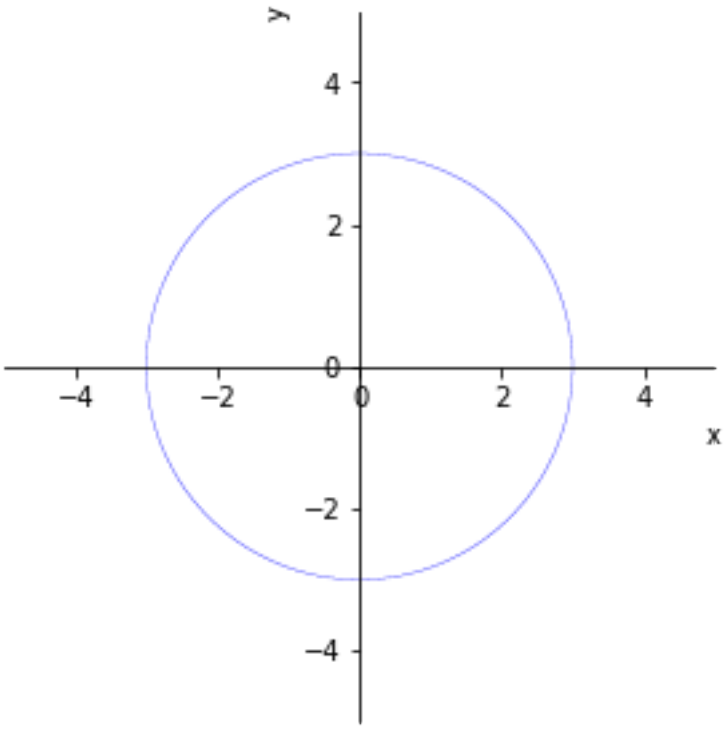
A documentación oficial de **SymPy** sobre representación gráfica está en https://docs.sympy.org/latest/modules/plotting.html.

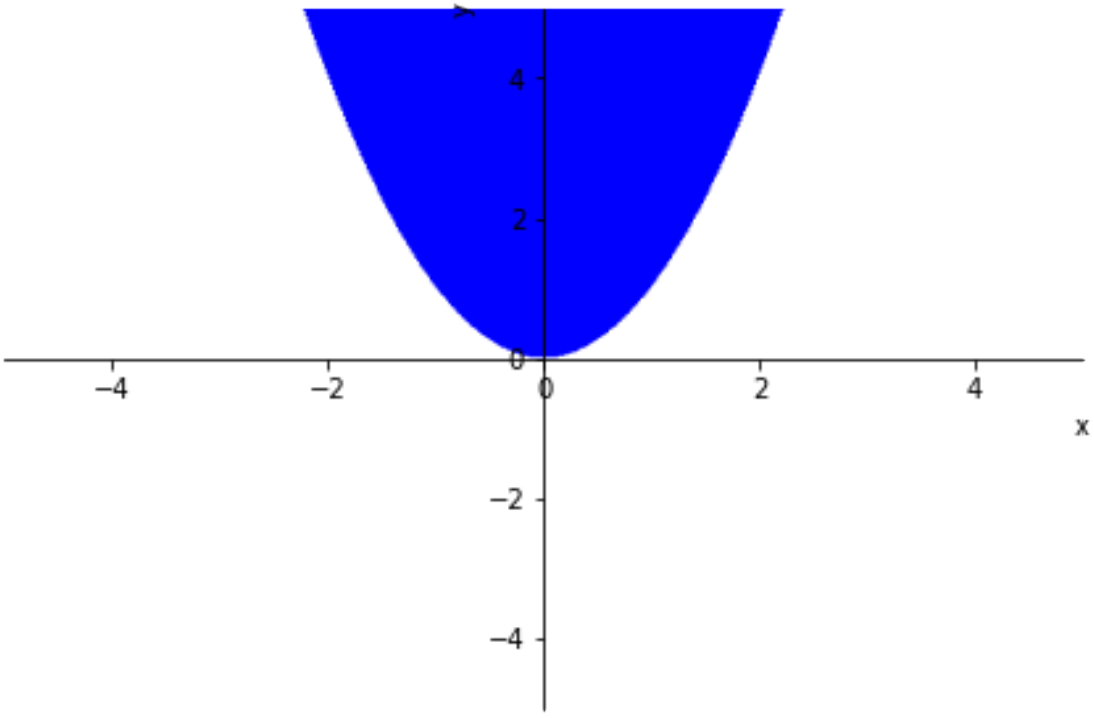
2.8.1 Rexións no plano

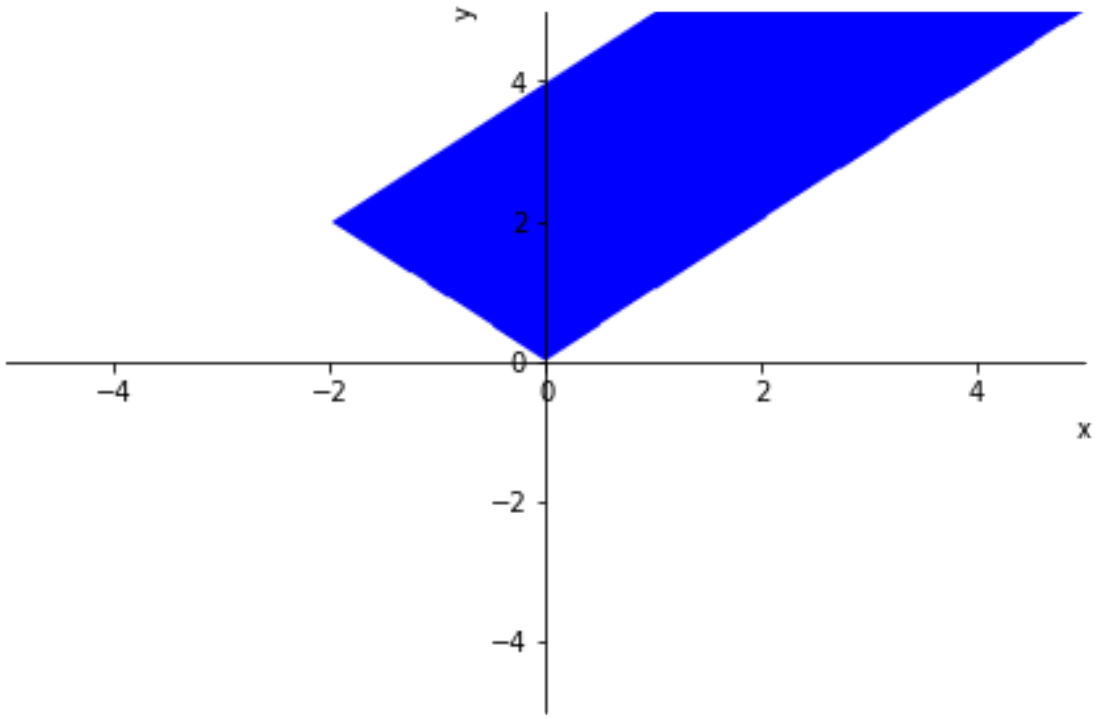
Podemos representar rexións coa instrucción `plot_implicit`, que pode ter como argumento de entrada tanto igualdades como desigualdades:

[21]:
```
x, y = sp.symbols('x,y', real=True)

p1 = sp.plot_implicit(sp.Eq(x**2 + y**2, 3**2))  # Cunha igualdade:␣
↪Representamos graficamente a circunferencia de centro (0,0) e raio 3.
p1 = sp.plot_implicit(sp.Eq(x**2 + y**2, 3**2), aspect_ratio=(1.,1.)) # Podemos␣
↪visualizar mellor o círculo considerando a mesma escala en cada eixo.
p2 = sp.plot_implicit(y > x**2)  # Unha desigualdade
p3 = sp.plot_implicit(sp.And(sp.And(y > x, y > -x), y < x + 4))  # Dúas␣
↪desigualdades, ligadas co operador lóxico `sp.And` (noutros casos, usaremos␣
↪`sp.Or`)
```




y
4
2
0
−4
−2
0
2
4
x
−2
−4

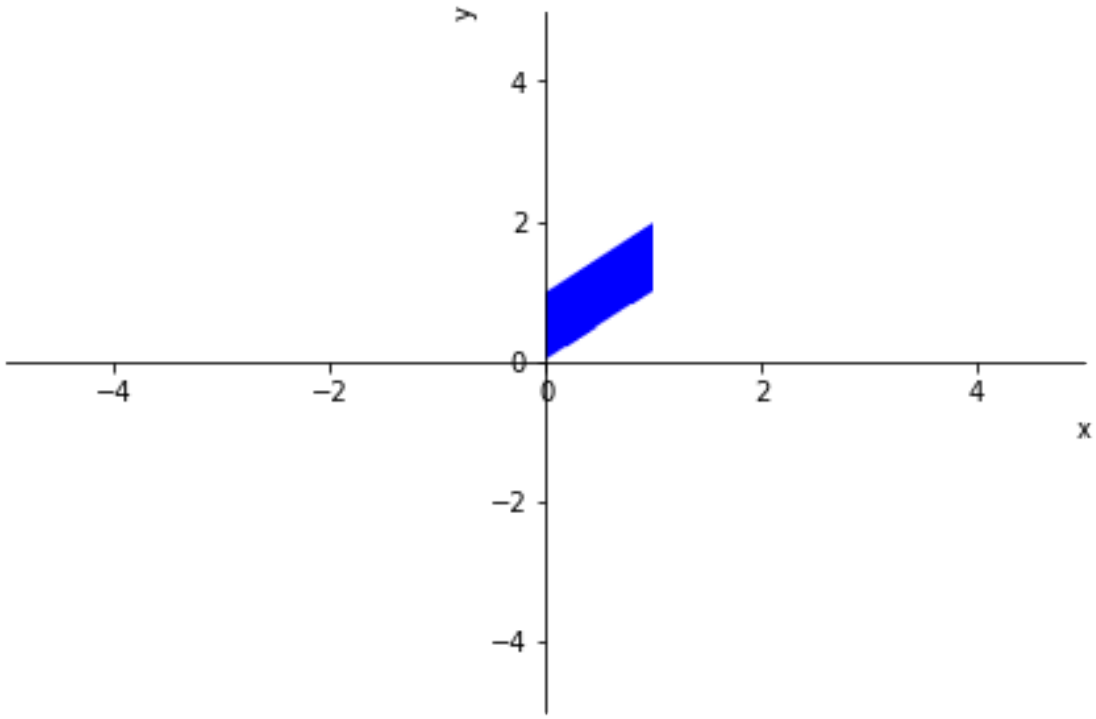

y
4
2
0
−4
−2
0
2
4
x
−2
−4



2.8.2 Exercicio:

Representa graficamente as siguientes rexións ou conxuntos no plano: - O conxunto de puntos que satisfán $y = 2x^2 - 3$. - A rexión, no plano, dos puntos que cumplen $y < x^2$ e $|y| < 2$. - A rexión, no plano, dos puntos que cumplen $y > x$ ou $y < -2x$. - A rexión, no plano, dos puntos que están no interior dun paralelogramo de vértices $(0,0)$, $(0,1)$, $(1,1)$ e $(1,2)$.

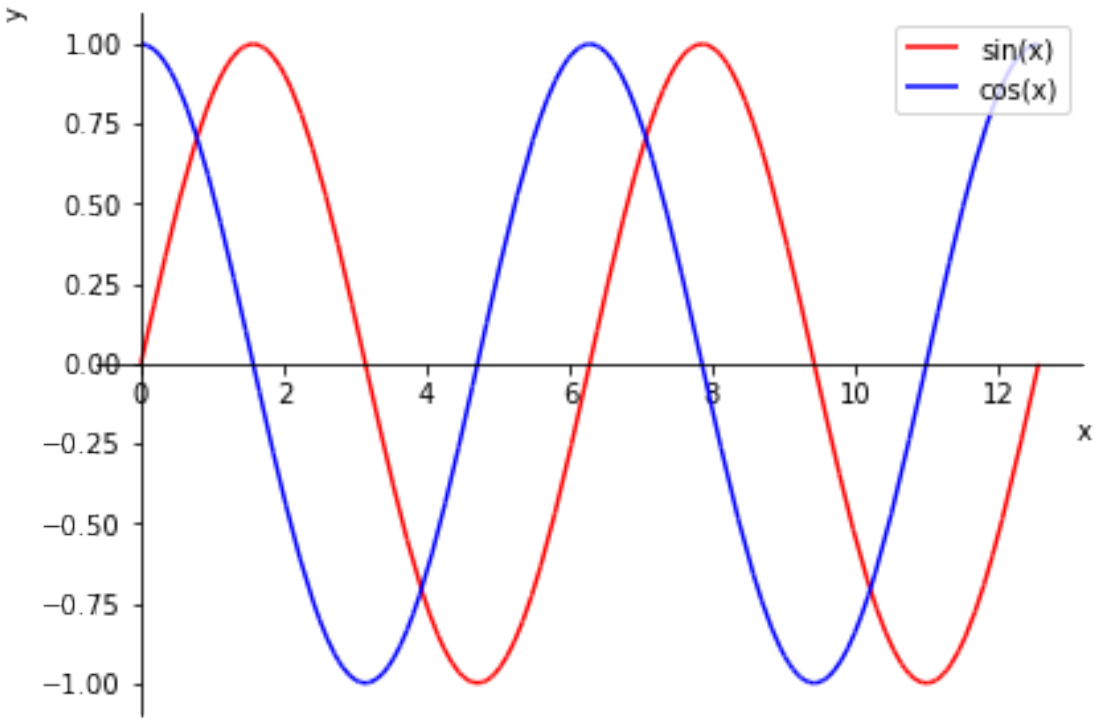
[22]:
```
#p4 = sp.plot_implicit(sp.Eq(2*x**2 - 3, y))
#p5 = sp.plot_implicit(sp.And(y<x**2,sp.Abs(y)<2))
#p6 = sp.plot_implicit(sp.Or(y>x,y<-2*x))
p7 = sp.plot_implicit(sp.And(sp.And(sp.And(x>0,x<1),y>x),y<x+1))
```



2.8.3 Gráficas de funcións

Para representar en gráficas funcións con **SymPy** utilizaremos a orde `plot` e a continuación a expresión da función que desexemos representar. No seguinte exemplo mostramos algunhas das opcións na representación:

[23]:
```
x = sp.symbols('x', real=True)
p = sp.plot(sp.sin(x), sp.cos(x), (x, 0, 4*sp.pi), show=False)
p[0].line_color='r'
p[1].line_color='b'
p.xlabel='x'
p.ylabel='y'
p.legend=True
p.show()
```



2.8.4 Exercicio

Podes representar outras funcións e/ou modificar as opcións na seguinte liña de código.

[ ]:

2.9 Funcións definidas a pedazos

A pesar de que en **SymPy** temos a función `Piecewise` para definir funcións a pedazos, esta ten moitas limitacións (por exemplo, para o cálculo de límites laterais). En xeral, é máis útil definir as funcións a pedazos usando a función de **Heaviside**, θ, que ven dada por:

$$\theta(x) = \begin{cases} 0 & \text{se } t \leq 0, \\ 1 & \text{se } t > 0. \end{cases}$$

Entón, unha función definida por

$$f(x) = \begin{cases} f_1(x) & \text{se } x \leq 0, \\ f_2(x) & \text{se } x > 0, \end{cases}$$

escribiríase como

$$f(x) = f_1(x) + (f_2(x) - f_1(x))\theta(x).$$

Escribimos por exemplo a función:

$$g(x) = \begin{cases} \frac{1}{x} & \text{se } x \leq 0, \\ 1 & \text{se } x > 0, \end{cases}$$

[24]:
```
# Definición da función
g1 = 1/x; g2 = 1
g_expr = g1 + (g2 - g1) * sp.Heaviside(x, 0)
g_pedazos = sp.Lambda(x, g_expr)
display(g_pedazos)
# Comprobar a definición da función g
display(sp.simplify(g_pedazos(x).rewrite(sp.Piecewise)))
```

$$\left(x \mapsto \left(1 - \frac{1}{x}\right)\theta\left(x\right) + \frac{1}{x}\right)$$

$$\begin{cases} \frac{1}{x} & \text{for } x \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

2.10 Límites

Podemos calcular límites utilizando a función `sp.limit`, vexamos como:

[25]:
```
g_expr = sp.cos(x)/(x+1)
g = sp.Lambda(x, g_expr)

display(g)

display(sp.limit(g(x),x,-1,dir='-')) # Límite pola esquerda
display(sp.limit(g(x),x,-1,dir='+')) # Límite pola dereita
```

$$\left(x \mapsto \frac{\cos\left(x\right)}{x+1}\right)$$

$-\infty$

∞

- Neste caso, o límite $\lim_{x \to -1} g(x)$ non existe xa que os límites laterais non coinciden. Pero unha incorrecta utilización do paquete pódenos levar a unha **conclusión errónea**:

[26]:
```
display(sp.limit(g(x),x,-1)) # Da un resultado (incorrecto) xa que, por defecto,␣
↪utiliza o valor do límite pola dereita
```

∞

- Tamén podemos calcular o límite da anterior función definida a pedazos:

[27]:
```
display(sp.simplify(g_pedazos(x).rewrite(sp.Piecewise)))
display(sp.limit(g_pedazos(x),x,0,dir='-')) # Límite pola esquerda
display(sp.limit(g_pedazos(x),x,0,dir='+')) # Límite pola dereita
```

$$\begin{cases} \frac{1}{x} & \text{for } x \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

$-\infty$

1

- Para calcular límites no infinito, $x \to +\infty$ ou $x \to -\infty$ representamos o valor de ∞ en **SymPy** por `sp.oo`.

[28]:
```
display(sp.limit(sp.exp(x),x,-sp.oo))
display(sp.limit(sp.exp(x),x,sp.oo))
```

0

∞

2.10.1 Exercicio

Representa graficamente a seguinte función e calcula os límites que se indican:

$$f(x) = \begin{cases} \cos(x) & \text{se } x < 1, \\ \frac{x^2}{x+1} & \text{se } x \geq 1, \end{cases}$$

- $\lim_{x\to 0} f(x)$,
- $\lim_{x\to 2} f(x)$,
- $\lim_{x\to 1^-} f(x)$,
- $\lim_{x\to 1^+} f(x)$.

[29]:
```
x = sp.symbols('x', real=True) # Definimos a variable x
f1 = sp.cos(x)
f2 = x**2/(x+1)
fx_ex = f1 + (f2 - f1) * sp.Heaviside(x-1, 1)
f = sp.Lambda(x, fx_ex)
print('f(x) =')
display(sp.simplify(f(x).rewrite(sp.Piecewise)))

# Calcular os límites
print('Límite en 0: ')
display(sp.limit(f(x),x,0))
print('Límite en 2: ')
display(sp.limit(f(x),x,2))
print('Límite en 1 pola dereita: ')
display(sp.limit(f(x),x,1,dir='+'))
print('Límite en 1 pola esquerda: ')
display(sp.limit(f(x),x,1,dir='-'))
```

```
display(float(sp.limit(f(x),x,1,dir='-')))


# Figura
p = sp.plot(f(x), (x, -10, 4), show=False)
p.line_color='b'
p.xlabel='x'
p.ylabel='y'
#p.legend=True
p.ylim=(-4,4)
p.show()

# Facemos zoom
p = sp.plot(f(x), (x, 0.8, 1.2), show=False)
p.line_color='b'
p.xlabel='x'
p.ylabel='y'
#p.legend=True
p.ylim=(0.3,0.8)
p.show()
```

```
f(x) =
```

$$\begin{cases} \cos(x) & \text{for } x < 1 \\ \frac{x^2}{x+1} & \text{otherwise} \end{cases}$$

```
Límite en 0:
```

$$1$$

```
Límite en 2:
```

$$\frac{4}{3}$$

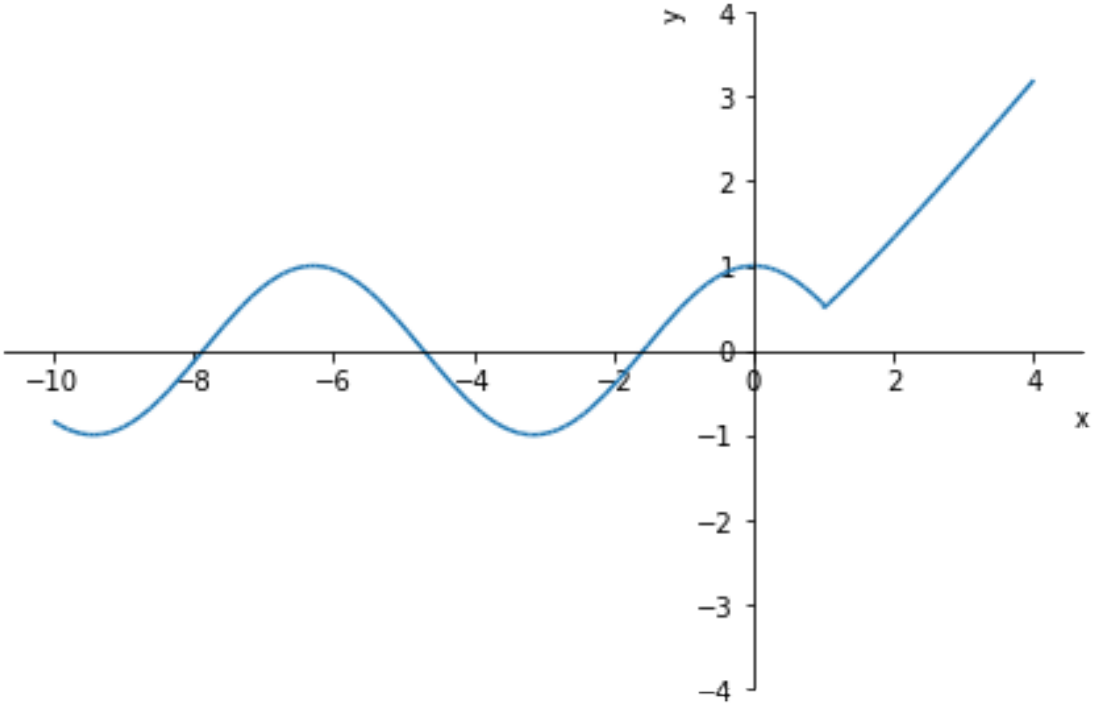
```
Límite en 1 pola dereita:
```

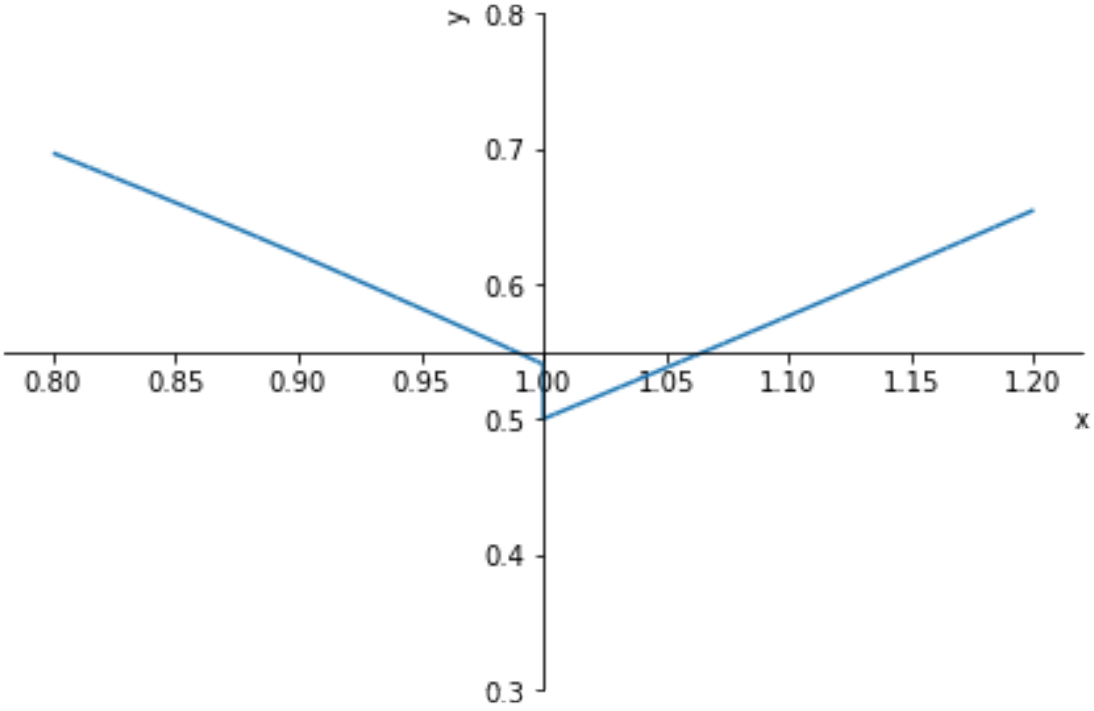
$$\frac{1}{2}$$

```
Límite en 1 pola esquerda:
```

$$\cos(1)$$

```
0.5403023058681398
```


y
4
3
2
1
0
−1
−2
−3
−4
−10
−8
−6
−4
−2
0
2
4
x


y
0.8
0.7
0.6
0.5
0.4
0.3
0.80
0.85
0.90
0.95
1.00
1.05
1.10
1.15
1.20
x

2.10.2 Exercicio

Observa como modificamos a función de Heaviside no anterior exercicio para poder escribir a función pedida. Utiliza a seguinte liña para describir outras funcións a pedazos.

[ ]:

3 Módulo NumPy

3.1 Vectores de números

En Python, existen moitas formas de gardar datos numéricos (ou non), como as estruturas de lista ou tupla. En particular, as listas poden conter unha secuencia finita de números ordenados (e utilizar un índice para acceder a cada un dos elementos da lista). Ademais, son suficientemente flexibles para conter datos de diferente natureza (combinación de números enteiros, reais, listas de listas, ...).

Pero esta flexibilidade das listas en Python fai que o seu rendemento computacional sexa moi limitado. Na maioría das aplicacións científicas en matemáticas (coas súas aplicacións en diferentes áreas da enxeñaría informática, intelixencia artificial, ciencia de datos, ...), os problemas reais requiren operacións sobre grandes conxuntos de datos e, por tanto, a velocidade computacional é moi importante para estes problemas. Para traballar de forma eficiente con estes problemas, **NumPy** proporciona funcións especializadas e estruturas de datos adecuadas. En particular, para o caso de conxuntos de números do mesmo tipo (perdendo parte da flexibilidade das listas, pero gañando eficacia computacional).

Vectores unidimensionais: Un vector unidimensional é unha colección ordenada de números aos que se pode acceder mediante un índice (co que se preserva a orde). Por defecto, os vectores en **NumPy** son vectores fila.

Creación de vectores e indexado: Para crear un vector **NumPy** de lonxitude 10 e inicializado con ceros, utilizamos a función `np.zeros()`:

[30]:
```
u = np.zeros(10)
print(u)
print(type(u))
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
<class 'numpy.ndarray'>
```

O tipo por defecto dos elementos contidos nos vectores en **NumPy** é `float64` (que é o tipo gardado en `np.float`). Se queremos usar outros tipos, temos que utilizar o argumento opcional `dtype`. O tipo dos elementos dun vector pódese comprobar mediante o atributo `dtype` dos vectores **NumPy**:

[31]:
```
print(u.dtype)
w = np.zeros(5, dtype=np.int)
print(w)
print(type(w))
print(w.dtype)
```

```
float64
[0 0 0 0 0]
<class 'numpy.ndarray'>
int32
```

Non se pode engadir un valor do tipo **cadea de texto** (tipo `string`) a un obxecto `numpy.ndarray`, xa que tódolos elementos do vector deben ser do mesmo tipo (ou dun tipo que admita unha conversión) e deben ter tamén o mesmo tamaño.

[32]:
```
v = np.zeros(10, dtype=np.int)
print(u + v) # Implicitamente faise unha conversión de tipo de int64 a float64
print(u + w) # ERRO: ¡Os vectores non teñen o mesmo tamaño!
```

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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-32-88119238798a> in <module>
      1 v = np.zeros(10, dtype=np.int)
      2 print(u + v) # Implicitamente faise unha conversión de tipo de int64 a␣
 ↪float64
----> 3 print(u + w) # ERRO: ¡Os vectores non teñen o mesmo tamaño!

ValueError: operands could not be broadcast together with shapes (10,) (5,)
```

3.2 Lonxitude de vectores

Para comprobar o tamaño dun vector, podemos usar a función `len`:

[33]:
```
print(len(u))
```

```
10
```

Outra forma de comprobar a dimensión dun vector é usar `u.shape`, que nos devolve unha tupla coas dimensións do vector (o tamaño do vector en cada dirección). No caso dos vectores, só hai unha única dirección, mentres que en conxuntos de datos con múltiples índices (matrices ou tensores *n*-dimensionais), `shape` infórmanos sobre o tamaño destas estruturas de datos en cada dirección.

[34]:
```
print(u.shape)
# probamos agora cunha matriz
A =  np.zeros((2,3), dtype=np.int)
print(A)
print(A.shape)
```

```
(10,)
[[0 0 0]
 [0 0 0]]
(2, 3)
```

3.3 Indexación

Podemos cambiar as entradas dun vector utilizando a indexación.

> **NOTA IMPORTANTE**: Hai que acordarse de que en Python os valores dos índices empezan en 0.

[35]:
```
print(u)
u[0] = 10.0
u[3] = -4.3
u[9] = 1.0
print(u)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[10.   0.   0.  -4.3  0.   0.   0.   0.   0.   1. ]
```

3.4 Formas de crear vectores

- Usando a función `np.ones` para crear un vector de uns.

[36]:
```
w = np.ones(5)
print(w)
print(w.dtype)
```

```
[1. 1. 1. 1. 1.]
float64
```

- Usando a función `random.rand` para crear un vector de valores aleatorios.

[37]:
```
w = np.random.rand(6)
print(w)
```

```
[0.65137336 0.60550115 0.67856141 0.13180408 0.51809417 0.85035035]
```

- Podemos crear vector un de números de tipo `numpy.array` a partir dunha lista Python de números:

[38]:
```
u = [4.0, 8.0, 9.0, 11.0, -2.0]
v = np.array(u)
print(v)
```

```
[ 4.  8.  9. 11. -2.]
```

- Dous métodos importantes para crear vectores de números son:
  - `np.arange` para crear vectores con valores enteiros consecutivos (de forma semellante á función de Python `range`).
  - `np.linspace` para crear vectores de números equiespaciados cun valor inicial e final (ambos incluídos) e dun tamaño dado.

[39]:
```
u = np.arange(2, 6)
print(u)
print(u.dtype)
```

```
[2 3 4 5]
int32
```

[40]:
```
w = np.linspace(0., 100., 6)
print(w)
print(w.dtype)
```

```
[  0.  20.  40.  60.  80. 100.]
float64
```

3.4.1 Exercicio

Podes utilizar a seguinte liña de códido para definir os teus propios vectores.

[ ]:
```

```

3.5 Funcións e operacións sobre vectores

Os vectores en **NumPy** soportan as operaciones aritméticas básicas (producto, sumas, restas, ...).

[41]:
```
a = np.array([1.0, 0.2, 1.2])
b = np.array([2.0, 0.1, 2.1])
print(a)
print(b)

# Suma de a e b
c = a + b
print(c)
```

```
[1.  0.2 1.2]
[2.  0.1 2.1]
[3.  0.3 3.3]
```

- Produto por un escalar:

[42]:
```
c = 10.0*a
print(c)
```

```
[10.  2. 12.]
```

- Elevar tódolos elementos a unha potencia.

[43]:
```
a = np.array([2, 3, 4])
print(a**2)
```

```
[ 4  9 16]
```

- Aplicar as funcións usuais do cálculo a cada unha das súas compoñentes.

[44]:
```
# Crear un vector [0, pi/2, pi, 3pi/2]
a = np.array([0.0, np.pi/2, np.pi, 3*np.pi/2])
print(a)
```

```
# Calcular o seno de cada compoñente do vector
b = np.sin(a)
print(b)
```

```
[0.         1.57079633 3.14159265 4.71238898]
[ 0.0000000e+00  1.0000000e+00  1.2246468e-16 -1.0000000e+00]
```

Debemos destacar que a función que se está a usar é `np.sin`, que depende directamente do módulo **NumPy**. O uso de calquera outra implementación da función noutros módulos (por exemplo, no módulo **Math**), podería dar lugar a erros.

Evidentemente, tamén podemos calcular o seno de cada coeficiente do vector, accedendo a cada un dos elementos mediante o seu índice e facendo os cálculos no interior dun bucle `for`. Neste caso, o programa é máis longo e difícil de ler. Ademais, en moitos casos será máis lento. A manipulación de vectores e calquera dos cálculos realizados entre eles sen acceder aos índices adoitan coñecerse como **vectorización**. Cando sexa posible empregala, a vectorización incrementará o rendemento e a velocidade dos códigos de cálculo.

[45]:
```
b = np.zeros(len(a))
for i in range(len(a)):
    b[i] = np.sin(a[i])

print(b)
```

```
[ 0.0000000e+00  1.0000000e+00  1.2246468e-16 -1.0000000e+00]
```

3.5.1 Exercicio

Podes utilizar a seguinte liña de códido para realizar distintas operacións con vectores.

[ ]:
```

```

3.6 Construción de subvectores

Cando se traballa con vectores de números, é común ter que extraer un subconxunto destes para crear un novo vector. Por exemplo, obter os tres primeiros coeficientes dun vector ou, no caso de matrices, restrinxir os cálculos á súa segunda columna. Este tipo de operacións coñécese como división de vectores (ou, en inglés, *array slicing*).

Imos explorar como proceder con diversos exemplos.

[46]:
```
a = np.random.rand(5)
print(a)

# Usar ':' implica o conxunto enteiro no rango dos índices, é dicir, desde 0 ata␣
 ↪(lonxitude-1)
b = a[:]
print("Usamos '[:]' {}".format(b))
```

```
# Usar '1:3' implica os índices 1 -> 3 (sen incluír a 3)
b = a[1:3]
print("Usamos '[1:3]': {}".format(b))

# Usar '2:-1' implica os índices 2 -> o primeiro desde o final (sen incluílo)
b = a[2:-1]
print("Usamos '[2:-1]': {}".format(b))

# Usar '2:-2' implica os índices 2 -> o segundo desde o final (sen incluílo)
b = a[2:-2]
print("Usamos '[2:-2]': {}".format(b))
```

```
[0.08227251 0.79784524 0.85255174 0.37818881 0.39790643]
Usamos '[:]' [0.08227251 0.79784524 0.85255174 0.37818881 0.39790643]
Usamos '[1:3]': [0.79784524 0.85255174]
Usamos '[2:-1]': [0.85255174 0.37818881]
Usamos '[2:-2]': [0.85255174]
```

> **NOTA**: O uso do índice `-1` corresponde ao último elemento do vector. Do mesmo xeito, o índice `-2` corresponde ao penúltimo elemento, etc.. Este convenio de facer referencia a índices desde o final dun vector é moi útil, xa que co uso destes índices negativos pódese facer referencia aos últimos coeficientes dun vector sen ter que facer referencia explícita ao seu tamaño (ou, incluso, sen coñecelo explicitamente).

[47]:
```
# Usar ':3' implica usar índices desde o principio ata 3 (sen incluír o índice 3)
b = a[:3]
print("Usamos '[:3]': {}".format(b))

# Usar '4:' implica os índices desde 4 -> ata o final
b = a[4:]
print("Usamos '[4:]': {}".format(b))

# Usar ':' implica tódolos índices desde o primeiro ao último
b = a[:]
print("Usamos '[:]': {}".format(b))
```

```
Usamos '[:3]': [0.08227251 0.79784524 0.85255174]
Usamos '[4:]': [0.39790643]
Usamos '[:]': [0.08227251 0.79784524 0.85255174 0.37818881 0.39790643]
```

3.6.1 Submatrices

O que acabamos de utilizar para os vectores podémolo facer con matrices.

[48]:
```
B = np.array([[1.3, 0], [0, 2.0]])
print(B)

# Extraer a segunda fila
row = B[1, :]
```

```
print(row)

# Extraer a primeira columna (almacenada nun vector fila)
col = B[:, 0]
print(col)
```

```
[[1.3 0. ]
 [0.  2. ]]
[0. 2.]
[1.3 0. ]
```

3.6.2 Exercicio

Podes seguir practicando como extraer distintos elementos de vectores e matrices na seguinte liña de código.

[ ]:

4 Gráficas con Matplotlib

Matplotlib é unha biblioteca de Python que se pode utilizar fóra dos Notebooks Jupyter para representar graficamente tanto funcións como entidades xeométricas (xa sexa na pantalla ou para gardalas nun ficheiro). Para que as representacións gráficas aparezan incrustadas neste documento, usaremos o comando máxico `%matplotlib inline`. Polo contrario, se é necesario interactuar coas gráficas, deberiamos usar a opción `%matplotlib notebook`.

[49]:
```
%matplotlib inline
```

4.1 Gráficas de funcións dunha variable

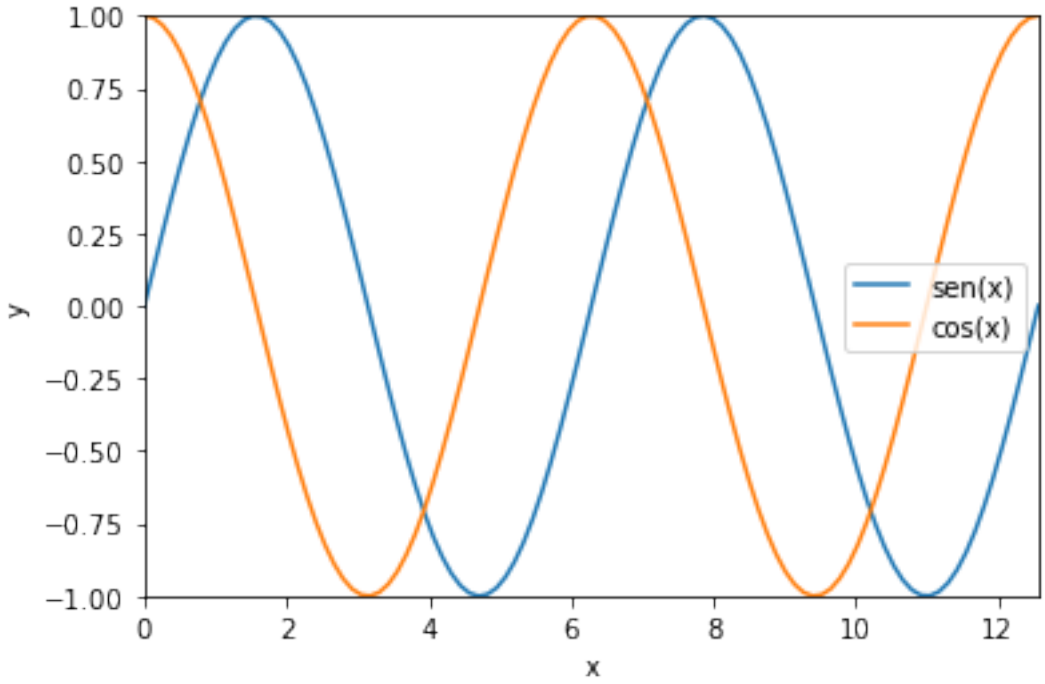
As gráficas de funcións dunha variable créanse a partir da avaliación do valor da función nun gran número de puntos almacenados nun vector **NumPy**. Ao usar un número de puntos suficiente, créase un efecto cinematográfico, e a gráfica da función parecerá suave. Sen embargo, é importante entender que o que se está representando graficamente é unha concatenación de segmentos rectos que unen os puntos avaliados. Para representar

$$f(x) = \sin(x), \qquad g(x) = \cos(x), \qquad x \in [0, 4\pi],$$

debemos crear un vector de valores nos que se vai avaliar a función:

[50]:
```
# Creación dos puntos onde se avalía a función
x = np.linspace(0, 4*np.pi, 100)

# Gráfica de sin(x) e cos(x), cunha etiqueta para cada unha
plt.plot(x, np.sin(x), label='sen(x)')
plt.plot(x, np.cos(x), label='cos(x)')

# Etiquetas dos eixos
plt.xlabel('x')
```

```
plt.ylabel('y')

# Engadir a lenda (mostrando as etiquetas dos "plot")
plt.legend()

# Definir os límites dos eixos x e y
plt.xlim(x[0], x[-1])
plt.ylim([-1.,1.])

plt.show()
```



Para representar gráficos de funcións con Matplotlib hai múltiples opcións, que podes ver aquí: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html.

4.1.1 Exercicio

Na seguinte liña de código podes experimentar a realizar outros gráficos máis complexos.

[ ]: