Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

6 de febrero de 2025

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro
ISBN-13 (versión electrónica): ???-?-????-????-?

Primera Edición (versión preliminar)
Escrito con XƎLaTeX.

Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2024
Web: http://pyciencia.taniquetil.com.ar/

10 9 8 7 6 5 4 3 2 1

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científiques, a partir de nuestra experiencia en el dictado del curso ”Herramientas Computacionales para Científicos”que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene tres grandes partes. La primera habla de Python, tanto de forma introductoria como también sobre otros temas que son fundamentales y algunas bibliotecas importantes. La segunda trata algunas herramientas fundamentales que son base para el trabajo en el resto del libro. Finalmente la tercera parte muestra cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En todos los casos este libro esquiva la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de les científiques para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

Índice general

Versión Preliminar

Parte I
Python

Esta primera parte comprende varios capítulos orientados a proveer la información necesaria de Python para poder entender el resto del libro.

Se sugiere a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

Parte II
Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos sobre herramientas fundamentales que serán aprovechados en la Parte III al abordar temas de aplicaciones específicas.

Versión Preliminar

Parte III
Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte II.

Versión Preliminar

1 | Matemática simbólica

Es común utilizar las computadoras para realizar cálculo numérico, que implica realizar veloces operaciones aritméticas utilizando muchas veces aproximaciones numéricas y siempre números con precisión determinada. Sin embargo, es posible también en muchos casos realizar operaciones exactas en forma simbólica (como resolver ecuaciones, derivar, integrar, simplificar expresiones) utilizando para ello símbolos y variables. En Python, esto es posible gracias al módulo (o biblioteca?) SymPy [1], que provee lo que se denomina un «sistema de álgebra computacional» (o CAS, del inglés *computer algebra system*).

Módulo	Versión
SymPy	1.13.1

Código disponible

Una característica de SymPy que lo diferencia de otros CAS es que está escrito en Python y puede ejecutarse completamente en Python como cualquier otro módulo (NumPy, SciPy, etc.), tanto en forma interactiva como dentro de un programa. En particular, al usarlo en un *notebook* de Jupyter, podemos mostrar las salidas interpretando LaTeX lo cual facilita enormemente la lectura de expresiones complejas.

1.1. Conceptos y operaciones básicas

Un concepto central en SymPy es el de símbolo. Éste representa, por medio de una variable simbólica, un valor abstracto o desconocido que puede ser utilizado en expresiones matemáticas. De este modo, es posible realizar cálculos simbólicos con dichas variables que son tratadas como entidades abstractas en vez de números específicos.

Los objetos de tipo símbolo se pueden instanciar a partir de `sympy.Symbol()` o `sympy.symbols()`. En el primer caso solo se instancia un único símbolo, mientras que el segundo caso permite instanciar varios símbolos en una sola instrucción. En ambos casos, además del nombre del símbolo creado, dichos objetos contienen atributos y métodos que describen sus propiedades y permiten operar sobre ellos. Del mismo modo que los objetos en Python, es posible asignarles nombres a los símbolos de modo de poder usarlos posteriormente.

Veamos cómo instanciar un símbolo x al que le asignaremos el nombre `x`, luego de importar el módulo `sympy` en la primera línea:

Celda 1 de conceptos_basicos.ipynb

```
import sympy as sym

x = sym.Symbol('x')
(x + 1)**2
```

$$(x+1)^2$$

El método `Symbol()` de SymPy recibe como argumento una cadena e instancia un objeto `Symbol` al que le asignamos el nombre `x`. Este nombre puede ser utilizado en expresiones, y al no tener un valor numérico asignado estas expresiones quedan en forma simbólica (y en el *notebook* se muestra «renderizado» a partir de la salida en LaTeX). En la celda siguiente instanciamos varios objetos utilizando `sympy.symbols()` mediante una cadena que tiene palabras (separadas por espacios, pero también pueden separarse mediante comas) compuestas por una o varias letras, incluyendo algunas que denotan letras del alfabeto griego. A dichos objetos les asignamos nombres que son utilizados para construir una expresión:

Celda 02

```
y, z, a, b = sym.symbols('y z alpha beta')
z + a + a + b * b
```

$$2\alpha + \beta^2 + z$$

Es posible, naturalmente, asignar un símbolo a un nombre arbitrario (por ejemplo el símbolo z a la variable `h`), sin embargo esto no constituye una buena práctica. Es importante señalar la diferencia entre un símbolo y el nombre que le asignamos. Veamos el siguiente ejemplo:

Celda 03

```
expresión = x + 2
x = 2
expresión
```

$$x+2$$

Este ejemplo muestra el procedimiento habitual de Python para asociar nombres a objetos tal como describimos en la Sección **??**. Primero asignamos al nombre `expresión` una operación suma en la que participa la variable `x`, que a su vez «apunta» hacia el símbolo x. En la línea siguiente «reorientamos» el nombre `x` hacia el objeto de tipo entero `2`. Cuando mostramos `expresión`, tal vez podríamos esperar obtener el valor 4, pero no es lo que ocurre, sino que recuperamos la definición inicial de `expresión` con el símbolo x, sin que el nuevo valor asignado al nombre `x` sea tenido en cuenta en la evaluación de `expresión`. Si esto último es lo que estábamos buscando, lo que tenemos que hacer es sustituir el símbolo x por el valor 2 en `expresión`:

Celda 04

```
x = sym.symbols('x')
expresión = x + 2
expresión.subs(x, 2)
```

$$4$$

En esta última celda, el nombre `x` representa un símbolo abstracto x, que «podría» ser un número entero, real, complejo, una función o cualquier otro ente matemático que pueda ser usado en expresiones y que permita su tratamiento simbólico. Es posible suministrar más información sobre el objeto de modo que SymPy pueda manipularlo apropiadamente. Podemos ver un ejemplo de esto en las celdas siguientes:

Celda 05

```
x = sym.Symbol('x')
y = sym.Symbol('y', positive=True)
sym.sqrt(x ** 2)
```

$$\sqrt{x^2}$$

Celda 06

```
sym.sqrt(y ** 2)
```

$$y$$

Aquí hemos definido dos variables simbólicas, estableciendo además que y es positiva. De este modo, al aplicar la función `sqrt()` sobre ambas, para el caso de x no se produce la simplificación automática de la expresión como si sucede con y. Del mismo modo, podemos definir variables que representen números enteros, pares e impares. Con esta información, SymPy puede simplificar o evaluar analíticamente las expresiones:

Celda 07

```
m = sym.Symbol('m')
n = sym.Symbol('n', integer=True)
l = sym.Symbol('l', odd=True)
sym.cos(m * sym.pi)
```

$$\cos(\pi m)$$

Celda 08

```
sym.cos(n * sym.pi)
```

$$(-1)^n$$

CELDA 09

```
sym.cos(l * sym.pi)
```

-1

Es posible preguntar si un símbolo posee determinados atributos. Por ejemplo, en el último caso, podemos interrogar al símbolo l si es un número par o impar:

CELDA 10

```
print(f' l es impar: {l.is_odd}, es par: {l.is_even}')
```

```
 l es impar: True, es par: False
```

Como hemos visto en la celda 2, es posible definir múltiples símbolos en una sola instrucción utilizando `symbols()`. En este caso, al establecer condiciones como las de las celdas precedentes, éstas aplican a todo el conjunto de símbolos creados. Es posible obtener el conjunto de atributos asignados a un determinado símbolo mediante `assumptions0`, que contiene un diccionario cuyas claves son las propiedades y los valores son **`True`**, **`False`** o **`None`**, dependiendo si esta propiedad ha sido definida o no:

CELDA 11

```
print(l.assumptions0)
```

```
{'odd': True, 'extended_nonzero': True, 'even': False, 'extended_real': True, 'finite': True,
    ↳ 'irrational': False, 'hermitian': True, 'commutative': True, 'noninteger': False, 'nonzero': True,
    ↳ 'algebraic': True, 'transcendental': False, 'zero': False, 'complex': True, 'real': True,
    ↳ 'imaginary': False, 'integer': True, 'rational': True, 'infinite': False}
```

Una característica que puede originar cierta confusión es que, dado que SymPy no extiende la sintaxis de Python, el símbolo = no representa una igualdad en SymPy, sino que sigue siendo el operador de asignación usual de Python. Tampoco == sirve para evaluar la igualdad matemática en SymPy, tal como se ve a continuación:

CELDA 12

```
(x + 2) ** 2 == x ** 2 + 4 * x + 4
```

```
False
```

En este caso, podemos identificar claramente que estamos desarrollando el cuadrado de un binomio, por lo que matemáticamente las expresiones a izquierda y derecha del operador de comparación son iguales. Sin embargo, el operador == compara estrictamente expresiones, que en el ejemplo de la celda 12 son estructuralmente diferentes en ambos lados del operador: a la izquierda tenemos un binomio elevado al cuadrado, mientras que a la derecha vemos la suma de tres términos. Para realizar la comparación en sentido matemático, podemos ver que $(x+2)^2 - x^2 - 4x - 4 = 0$, entonces podemos intentar simplificar la diferencia entre ambos miembros y verificar la igualdad cuando esa simplificación dé como resultado un cero. Para realizar la simplificación algebraica de expresiones utilizamos el método `simplify()`:

Celda 13

```
izquierda = (x + 2) ** 2
derecha = x ** 2 + 4 * x + 4
sym.simplify(izquierda - derecha)
```

$$0$$

Celda 14

```
derecha_2 = x ** 2 + 2 ** 2
sym.simplify(izquierda - derecha_2)
```

$$4x$$

En la celda 14 podemos comprobar que las expresiones no son matemáticamente iguales excepto para $x = 0$. Este método para verificar igualdades de expresiones no es infalible, dado que se puede probar teóricamente que es imposible determinar si dos expresiones simbólicas son idénticamente iguales en general[1]. No obstante funciona aceptablemente para expresiones habituales no muy complejas. SymPy también dispone del método `equals()`, que comprueba la igualdad de expresiones por medio de evaluaciones numéricas en puntos aleatorios:

Celda 15

```
print(izquierda.equals(derecha))
print(izquierda.equals(derecha_2))
```

```
True
False
```

En la celda 4 vimos que podemos reemplazar un símbolo por un valor numérico utilizando el método `subs()`. En general, podemos usar este método para sustituir todas las instancias de algo en una expresión con alguna otra cosa. Por ejemplo:

Celda 16

```
expresión = 1 + sym.cos(x)
expresión.subs(x, 0)
```

$$2$$

Celda 17

```
expresión.subs(sym.cos(x), sym.sqrt(1 - sym.sin(x) ** 2))
```

$$\sqrt{1 - \sin^2(x)} + 1$$

1 Ver la entrada del teorema de Richardson en Wikipedia.

Celda 18

```
expresión
```

$\cos(x) + 1$

En el caso de la celda 16, simplemente reemplazamos el símbolo x por el valor numérico 0, y como resultado se muestra la evaluación de la expresión. En la celda 17, la sustitución consiste en reemplazar $\cos(x)$ por $\sqrt{1 - \operatorname{sen}^2(x)}$, sin realizar evaluaciones numéricas. Finalmente, vemos que `subs()` devuelve un nueva expresión, dado que los objetos de SymPy son inmutables, tal como se ve al mostrar el valor que contiene la variable `expresión`.

Es posible también construir expresiones simbólicas de SymPy a partir de cadenas, utilizando para ello el método `sympify()` (no confundir con `simplify()`):

Celda 19

```
cadena = 'x ** 2 - 5 * x + 1'
expr = sym.sympify(cadena)
expr
```

$x^2 - 5x + 1$

Como dijimos al inicio de esta sección, los símbolos en SymPy permiten representar diversos objetos matemáticos, entre los que se encuentran particularmente los números. Dado que éstos comparten atributos y métodos de `Symbol`, deben ser objetos distintos a los números propios de Python (como `int` y `float`). La conversión de números de Python a objetos de SymPy se realiza automáticamente por SymPy, o podemos hacerlo en forma explícita, y luego podemos acceder a diversos atributos, por ejemplo:

Celda 20

```
i = sym.Integer(13)
type(i)
```

```
sympy.core.numbers.Integer
```

Celda 21

```
i.is_Integer, i.is_real, i.is_odd
```

```
(True, True, True)
```

Celda 22

```
f = sym.Float(3.14)
type(f)
```

```
sympy.core.numbers.Float
```

Celda 23

```
f.is_Integer, f.is_real, f.is_odd
```

```
(False, True, False)
```

Podemos volver a convertir instancias de números de SymPy en los correspondientes tipos intrínsecos de Python con las funciones usuales `int()` y `float()`:

Celda 24

```
i_py = int(i)
f_py = float(f)
type(i_py), type(f_py)
```

```
(int, float)
```

En la celda 7 vimos que podemos instanciar un objeto de `Symbol` pasando el parámetro `integer=True`. De esta forma, este objeto representa «algún» entero, mientras que una instancia de `Integer` representa un entero específico. En ambos casos, el atributo `is_integer` es `True`, pero solo en el segundo es `True` el atributo `is_Integer` (notar la «I» mayúscula). Por ejemplo:

Celda 25

```
i = sym.Symbol('i', integer=True)
j = sym.Integer(4)
print(f"i: {i.is_integer}, {i.is_Integer}, {i.is_positive}, {i.is_Symbol}")
print(f"j: {j.is_integer}, {j.is_Integer}, {j.is_positive}, {j.is_Symbol}")
```

```
i: True, False, None, True
j: True, True, True, False
```

Los números enteros en SymPy son de precisión arbitraria, de modo que no tienen límites inferior y superior fijos como ocurre cuando se representan enteros con un número fijo de bits (como en el caso de los enteros de NumPy):

Celda 26

```
print(sym.factorial(100))
```

```
93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000
```

En la práctica, por supuesto, existen límites dados por los recursos finitos de la computadora, como la cantidad de memoria disponible. Al igual que para los enteros, los objetos `Float` de SymPy también son de precisión arbitraria, en contraste con los `float` intrínsecos de Python o de NumPy. El constructor de `Float` recibe dos argumentos: el primero es un `float` de Python o una cadena, mientras que el segundo, opcional, representa la precisión requerida. Por ejemplo, veamos el tratamiento diferente que recibe el número 0,3 en Python y como objeto de SymPy:

Versión Preliminar

Celda 27

```
print(f"{0.3:.25f}")    # representación como cadena con 25 decimales
print(sym.Float(0.3, 25))
print(sym.Float('0.3', 25))
```

```
0.2999999999999999888977698
0.2999999999999999888977698
0.3000000000000000000000000
```

Sabemos que en la representación de punto flotante binaria, el número 0,3 no se puede representar exactamente, y así lo muestra la primera línea de la salida de la celda precedente. Al instanciar un objeto `Float` de SymPy a partir del número `0.3`, obtenemos la misma representación inexacta que el caso anterior, dado que SymPy recibe como argumento un número que Python no puede representar correctamente. Sin embargo, utilizando el constructor usando la cadena `'0.3'`, SymPy muestra correctamente la representación del número con la precisión requerida como segundo argumento.

SymPy también ofrece la posibilidad de expresar números racionales, y operar sobre ellos con la matemática usual. Por ejemplo:

Celda 28

```
a = sym.Rational(3, 5)
b = sym.Rational(2, 3)
a + b
```

$$\frac{19}{15}$$

Celda 29

```
a * b
```

$$\frac{2}{5}$$

Dado un número en forma racional, podemos acceder separadamente a su numerador y denominador a través de los atributos `.p` y `.q`:

Celda 30

```
(a * b).p, (a * b).q
```

```
(2, 5)
```

Finalmente, SymPy contiene también constantes numéricas que se pueden utilizar en expresiones, como π, e, ∞ (que se representa con dos letras «o» minúsculas seguidas), entre otras, así como constantes físicas y unidades:

Versión Preliminar

Celda 31

```
print(sym.pi.evalf(), sym.E.evalf(), sym.oo.evalf())
```

```
3.14159265358979 2.71828182845905 oo
```

1.2. Simplificación de expresiones

Una de las características más utilizadas en un CAS es la manipulación algrebraica de expresiones complejas para obtener expresiones más simples. SymPy contiene muchas funciones que realizan diferentes tareas de simplificación. Una de ellas, que mostramos en un ejemplo anterior, es `simplify()`. Esta función intenta aplicar, en forma heurística, todas las funciones específicas que contiene SymPy con el propósito de obtener la forma más simple de una expresión. Veamos algunos casos:

Celda 1 de algebra.ipynb

```
import sympy as sym
x, y, z = sym.symbols('x y z')
```

Celda 02

```
sym.simplify( (x ** 3 - 2 * x ** 2 - 13 * x - 10) / (x ** 2 - 4 * x - 5))
```

$x + 2$

Celda 03

```
sym.simplify(2 * sym.cos(x)**2 + 2 * sym.sin(x)**2)
```

2

Celda 04

```
sym.simplify(sym.gamma(x)/sym.gamma(x - 1))
```

$x - 1$

En la salida de la celda 2 obtenemos una expresión simple para un cociente de polinomios. En la celda siguiente el resultado se obtiene a partir de la simplificación de una identidad trigonométrica. En el último ejemplo, `sp.gamma(x)` representa la función gamma $\Gamma(x)$. Si bien `simplify` intenta encontrar la expresión más simple, a veces no estamos de acuerdo con el resultado que nos ofrece. Por ejemplo:

Celda 05

```
sym.simplify(x ** 2 + 4 * x + 4)
```

$x^2 + 4x + 4$

Como vemos, la salida de la función es simplemente su argumento, cuando tal vez esperábamos obtener $(x+2)^2$. Veremos a continuación que podemos intentar realizar la simplificación utilizando un método más específico.

1.2.1. Simplificación de funciones polinómicas y racionales

Una de las funciones de simplificación más utilizadas en SymPy es `expand()`, que dado un polinomio, lo expresa en la forma canónica como suma de monomios:

Celda 06

```
sym.expand((x - 2) * (x + 3) * (x - 1))
```

$$x^3 - 7x + 6$$

Celda 07

```
sym.expand((x + 1) ** 4)
```

$$x^4 + 4x^3 + 6x^2 + 4x + 1$$

Tal como su nombre lo indica, `expand()` no parece realizar tareas de simplificación dado que devuelve expresiones más grandes. Generalmente este es el caso, pero en ocasiones se obtienen expresiones más simples debido a cancelaciones de términos:

Celda 08

```
sym.expand((x + 1)*(x - 2) - (x - 1)*x)
```

$$-2$$

Otra función de simplificación muy útil es `factor()`, que recibe polinomios como argumentos y los expresa como productos de factores irreducibles sobre los números racionales. Por ejemplo:

Celda 09

```
sym.factor(x ** 3 + x ** 2 - x - 1)
```

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

Celda 10

```
sym.factor(x ** 2 * z + 4 * x * y * z + 4 * y ** 2 * z)
```

$$z\,(x+2y)^2$$

Es posible también acceder a la lista de los factores de una expresión, junto con sus correspondientes exponentes, por medio de `factor_list()`:

Celda 11

```
sym.factor_list(x ** 2 * z + 4 * x * y * z + 4 * y ** 2 * z)
```

```
(1, [(z, 1), (x + 2*y, 2)])
```

Para polinomios, `factor()` y `expand()` son funciones opuestas. Sin embargo, los argumentos de estas funciones no necesariamente deben ser polinomios, ya que ambas intentarán factorizar o expandir cualquier tipo de expresiones (aunque los factores pueden no ser irreducibles si los argumentos no son polinomios sobre los racionales):

Celda 12

```
sym.expand((sym.cos(x) + sym.sin(x)) ** 2)
```

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

Celda 13

```
sym.factor(sym.cos(x) ** 2 + sym.sin(x) ** 2 - 2 * sym.sin(x) * sym.cos(x))
```

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

La función `collect()` permite agrupar las potencias de una variable determinada en una expresión. Por ejemplo:

Celda 14

```
expresión = x ** 4 + 3 * (x * z) ** 3 + x ** 2 + x * y - z * x
sym.collect(expresión , x)
```

$$x^4 + 3x^3z^3 + x^2 + x(y - z)$$

Si queremos aislar el coeficiente de una potencia específica de una expresión, podemos usar el método `coeff()` tal como hacemos en la celda siguiente, para obtener el coeficiente de x^3 de `expresión`:

Celda 15

```
expresión.coeff(x, 3)
```

$$3z^3$$

La función `cancel()` expresa una función racional en su argumento en la forma canónica p/q, donde numerador y denominador son polinomios sin factores comunes y los coeficientes de las máximas potencias de p y q son enteros (no fracciones). Por ejemplo:

Versión Preliminar

Celda 16

```
sym.cancel((x ** 2 + 4 * x + 4) / (x ** 2 + 2 * x))
```

$$\frac{x+2}{x}$$

Celda 17

```
expresión = 1 / x + (5 * x / 4 - 4)/(x - 2)
expresión
```

$$\frac{\frac{5x}{4}-4}{x-2}+\frac{1}{x}$$

Celda 18

```
sym.cancel(expresión)
```

$$\frac{5x^2-12x-8}{4x^2-8x}$$

Por último, mencionamos la función `apart()`, que realiza una descomposición en fracciones simples de una expresión racional:

Celda 19

```
e_racional = (x ** 2 + 3 * x + 1) / (x + 1) ** 3
e_racional
```

$$\frac{x^2+3x+1}{(x+1)^3}$$

Celda 20

```
sym.apart(e_racional)
```

$$\frac{1}{x+1}+\frac{1}{(x+1)^2}-\frac{1}{(x+1)^3}$$

1.2.2. Simplificaciones trigonométricas

Para el caso de las funciones trigonométricas, podemos utilizar `trigsimp()`, que intenta simplificar expresiones utilizando identidades trigonométricas, y `expand_trig()` que aplica identidades para la suma o productos de ángulos:

Celda 21

```
sym.trigsimp(sym.sin(x) ** 2 + sym.cos(x) ** 2)
```

$$1$$

Celda 22

```
sym.trigsimp(sym.sin(x) / sym.tan(x))
```

$$\cos(x)$$

Celda 23

```
sym.trigsimp(sym.sinh(x) ** 2 + sym.cosh(x) **2)
```

$$\cosh(2x)$$

Celda 24

```
sym.expand_trig(sym.sin(x) + sym.sin(3 * x))
```

$$-4\sin^3(x) + 4\sin(x)$$

1.2.3. Simplificación de potencias

SymPy es capaz de simplificar expresiones utilizando las reglas usuales de potenciación:

1. $x^a x^b = x^{a+b}$
2. $x^a y^a = (xy)^a$
3. $(x^a)^b = x^{ab}$

No obstante, no realiza simplificaciones cuando dichas expresiones no son válidas en general. Por ejemplo: $x^a x^b = x^{a+b}$ es una igualdad válida, pero $x^a y^a = (xy)^a$ solo lo es cuando x y y no son negativos y a es real. Por defecto, SymPy asume que los símbolos sobre los que opera son complejos, pero como vimos previamente, es posible indicar atributos durante la creación de estos símbolos. Por ejemplo:

Celda 25

```
x, y = sym.symbols('x y', positive=True)
a, b = sym.symbols('a b', real=True)
m, n, l = sym.symbols('m n l')
```

Celda 26

```
sym.powsimp(x ** a * x ** b)
```

$$x^{a+b}$$

Celda 27

```
sym.powsimp(x ** a * y ** a)
```

$(xy)^a$

Celda 28

```
sym.powsimp(m ** l * n ** l)
```

$m^l n^l$

En el último caso, no hemos indicado atributos en la creación de m, n, y l que permitan a SymPy simplificar la expresión, sin embargo, podemos forzar a que se realice la simplificación independientemente de los atributos:

Celda 29

```
sym.powsimp(m ** l * n ** l, force=True)
```

$(mn)^l$

Las operaciones de expansión de expresiones con potencias aplican las identidades 1 y 2 de derecha a izquierda, respectivamente, mediante las funciones `expand_power_exp()` y `expand_power_base()`:

Celda 30

```
sym.expand_power_exp(x ** (a + b))
```

$x^a x^b$

Celda 31

```
sym.expand_power_base((x * y) ** a)
```

$x^a y^a$

Al igual que en la simplificación, la identidad 2 no se aplica si no es válida, aunque podemos forzar a que se produzca:

Celda 32

```
sym.expand_power_base((m * n) ** l)
```

$(mn)^l$

Celda 33

```
sym.expand_power_base((m * n) ** l, force=True)
```

$m^l n^l$

Para finalizar, podemos aplicar la identidad 3 usando `powdenest()`, siempre que los atributos de los símbolos garanticen su validez:

Celda 34

```
sym.powdenest((x ** a) ** b)
```

x^{ab}

Celda 35

```
sym.powdenest((m ** a) ** b)
```

$(m^a)^b$

Celda 36

```
sym.powdenest((m ** a) ** b, force=True)
```

m^{ab}

1.2.4. Simplificación de exponenciales y logaritmos

Las principales identidades que intenta aplicar SymPy para simplificar expresiones con logaritmos son:

1. $\log(xy) = \log(x) + \log(y)$
2. $\log(x^n) = n\log(x)$

Estas identidades no son válidas para valores complejos arbitrarios de x y y, por lo que para que las expresiones puedan simplificarse es necesario establecer los atributos correspondientes:

Celda 37

```
x, y = sym.symbols('x y', positive=True)
n = sym.symbols('n', real=True)
p, q = sym.symbols('p q')
```

Celda 38

```
sym.expand_log(sym.log(x * y))
```

$\log(x) + \log(y)$

Celda 39

```
sym.expand_log(sym.log(x / y))
```

$\log(x) - \log(y)$

Celda 40

```
sym.expand_log(sym.log(x ** 3))
```

$3\log(x)$

Celda 41

```
sym.expand_log(sym.log(x ** n))
```

$n\log(x)$

Celda 42

```
sym.expand_log(sym.log(p * q))
```

$\log(pq)$

Celda 43

```
sym.expand_log(sym.log(p * q), force=True)
```

$\log(p) + \log(q)$

La operación inversa a la expansión se realiza con `logcombine()`:

Celda 44

```
sym.logcombine(sym.log(x) + sym.log(y))
```

$\log(xy)$

Celda 45

```
sym.logcombine(n * sym.log(x))
```

$\log(x^n)$

Celda 46

```
sym.logcombine(n * sym.log(p))
```

$n\log(p)$

Celda 47

```
sym.logcombine(n * sym.log(p), force=True)
```

$\log(p^n)$

1.2.5. Funciones especiales

Para finalizar esta sección mencionaremos algunas funciones más que permiten manipular expresiones para expresarlas de un modo más simple (o conveniente). Una de ellas es `rewrite()`, que sirve para escribir una expresión en términos de una función, tal como vemos en los ejemplos siguientes:

Celda 48

```
sym.tan(x).rewrite(sym.sin)
```

$$\frac{2\sin^2(x)}{\sin(2x)}$$

Celda 49

```
sym.factorial(x).rewrite(sym.gamma)
```

$$\Gamma(x+1)$$

Es posible también simplificar expresiones combinatorias por medio de `combsimp()`:

Celda 50

```
n, k = sym.symbols('n k', integer=True)
sym.combsimp(sym.factorial(n + 1) / sym.factorial(n - 1))
```

$$n(n+1)$$

Celda 51

```
sym.combsimp(sym.binomial(n + 2, k + 2) / sym.binomial(n, k))
```

$$\frac{(n+1)(n+2)}{(k+1)(k+2)}$$

Del mismo modo podemos intentar transformar expresiones que contienen funciones gamma o combinatorias con argumentos no enteros, usando la función `gammasimp()`:

Celda 52

```
sym.gammasimp(sym.gamma(2 + x) * sym.gamma(1 - x))
```

$$\frac{\pi x(x+1)}{\sin(\pi x)}$$

Por último, mostramos un ejemplo en el que expendimos una función especial (gamma) utilizando alguna identidad conocida, utilizando la función `expand_func()`:

Celda 53

```
sym.expand_func(sym.gamma(x + 4))
```

$$x(x+1)(x+2)(x+3)\Gamma(x)$$

1.3. Solución de ecuaciones

SymPy es capaz de resolver, en forma simbólica, ecuaciones lineales y no lineales, sistemas de ecuaciones lineales, desigualdades, ecuaciones diofánticas y ecuaciones diferenciales[2]. En caso que la solución analítica de una ecuación (o sistema de ecuaciones) no exista, SymPy provee la posibilidad de resolverlas en forma numérica.

Antes de iniciar un recorrido sobre algunas de estas posibilidades, es necesario recordar que una ecuación en SymPy no se representa con un símbolo `=` o `==`, sino a través de un objeto `Eq`. Por ejemplo:

Celda 1 de ecuaciones.ipynb

```
import sympy as sym

x, y = sym.symbols('x y')
sym.Eq(2 * x + 8, 16 * y)
```

$$2x+8=16y$$

Por otra parte, cualquier expresión que no sea una instancia de `Eq` se asume igual a cero en el contexto de funciones que resuelven ecuaciones (usualmente denominadas «*solvers*»). Dado que en el ejemplo anterior $2x+8=16y$ es equivalente a $2x+8-16y=0$, podemos pasar esta última expresión como argumento a un *solver* para intentar obtener una solución a dicha ecuación:

Celda 02

```
sym.solveset(sym.Eq(2 * x + 8, 16 * y), x)
```

$$\{8y-4\}$$

Celda 03

```
sym.solveset(sym.Eq(2 * x + 8 - 16 * y, 0), x)
```

$$\{8y-4\}$$

2 Para ver cómo resolver ecuaciones diferenciales, ver el Capítulo ??.

Celda 04

```
sym.solveset(2 * x + 8 - 16 * y, x)
```

$\{8y - 4\}$

Existen dos funciones para resolver ecuaciones algebraicas: `solve()` y `solveset()`. La primera opción es útil para obtener representaciones simbólicas de las posibles soluciones de una ecuación, y para sustituir dichas representaciones en otras ecuaciones o expresiones. `solveset()` es una alternativa que representa las soluciones utilizando conjuntos matemáticos, y permite obtener todas las soluciones incluso si el conjunto es infinito (las recomendaciones de uso de una función u otra se pueden ver acá.). Veamos algunos ejemplos.

Celda 05

```
sym.solve(x ** 2 + 2 * x - 3)
```

```
[-3, 1]
```

Celda 06

```
sym.solveset(x ** 2 + 2 * x - 3, x)
```

$\{-3, 1\}$

En las dos celdas precedentes resolvemos la ecuación $x^2 + 2x - 3 = 0$ (o lo que es lo mismo, determinamos las raíces del polinomio). En el caso de `solve()`, obtenemos una lista enumerando las soluciones para la variable x, mientras que con `solveset()` lo que resulta es un conjunto finito. En la expresión que representa al polinomio, el único símbolo «indeterminado» es la variable independiente x, mientras que los coeficientes y potencias son números. Por este motivo es opcional indicar sobre qué símbolo debe resolverse la ecuación (no especificado en la celda 5). Naturalmente, podemos utilizar expresiones que contengan otros símbolos además de la variable que queremos determinar como solución, y en este caso debemos indicar cuál es esa variable:

Celda 07

```
a, b, c = sym.symbols('a b c')
sym.solveset(a *x ** 2 + b * x + c, x)
```

$$\left\{-\frac{b}{2a} - \frac{\sqrt{-4ac + b^2}}{2a}, -\frac{b}{2a} + \frac{\sqrt{-4ac + b^2}}{2a}\right\}$$

SymPy permite, en algunos casos, hallar también la solución de ecuaciones trigonométricas. En el ejemplo siguiente podemos apreciar la diferencia en los resultados de `solve()` y `solveset()`:

Celda 08

```
sym.solve(sym.cos(x) - sym.sin(x), x)
```

```
[pi/4]
```

Celda 09

```
sym.solveset(sym.cos(x) - sym.sin(x), x)
```

$$\left\{2n\pi + \frac{5\pi}{4} \;\middle|\; n \in \mathbb{Z}\right\} \cup \left\{2n\pi + \frac{\pi}{4} \;\middle|\; n \in \mathbb{Z}\right\}$$

En los casos en que no exista una solución analítica, o que SymPy sea incapaz de hallarla, devolverá una solución formal que eventualmente puede ser resuelta en forma numérica:

mientras que si no existe la solución de una ecuación, estas funciones devolverán una lista vacía o un conjunto vacío:

Celda 12

```
sym.solve(sym.exp(x), x)
```

```
[]
```

Celda 13

```
sym.solveset(sym.exp(x), x)
```

$\emptyset$

La solución de un sistema de ecuaciones consiste en extender el procedimiento que utilizamos para ecuaciones de una sola incógnita. Simplemente es necesario pasar como argumento una lista de las ecuaciones a resolver, y otra con la lista de símbolos para los que se busca la solución (o las soluciones). Por ejemplo:

Celda 14

```
ecuación_1 = 2 * x - 5 * y + 4
ecuación_2 = -3 * x + 4 * y + 1
sym.solve([ecuación_1, ecuación_2], [x, y], dict=True)
```

```
[{x: 3, y: 2}]
```

Celda 15

```
ecuación_1 = x ** 2 - 2 * y
ecuación_2 = y ** 2 - 2 * x
soluciones = sym.solve([ecuación_1, ecuación_2], [x, y], dict=True)
soluciones
```

```
[{x: 0, y: 0},
 {x: 2, y: 2},
 {x: (-1 - sqrt(3)*I)**2/2, y: -1 - sqrt(3)*I},
 {x: (-1 + sqrt(3)*I)**2/2, y: -1 + sqrt(3)*I}]
```

En estas últimas dos celdas, `solve()` devuelve una lista en la que cada elemento representa una solución del sistema. Además, hemos pasado como argumento opcional `dict=True` de forma que la respuesta de SymPy sea en forma de diccionario, lo que resulta útil para reutilizar los valores obtenidos de las soluciones. Por ejemplo, podemos verificar que se satisfacen las ecuaciones:

Versión Preliminar

CELDA 16

```
[ecuación_1.subs(sol).simplify() == 0 and ecuación_2.subs(sol).simplify() == 0
 for sol in soluciones]
```

```
[True, True, True, True]
```

Por defecto, SymPy ofrecerá soluciones en el campo complejo para las incógnitas:

CELDA 17

```
sym.solve(x ** 4 - 2 ** 4, x)
```

```
[-2, 2, -2*I, 2*I]
```

CELDA 18

```
sym.solveset(x ** 4 - 2 ** 4, x)
```

$$\{-2, 2, -2i, 2i\}$$

Es posible restringir el dominio de búsqueda de soluciones, por ejemplo limitando la incógnita al campo real o a un intervalo dado. Las funciones `solve()` y `solveset()` tienen abordajes diferentes para ello.

En el caso de `solve()`, podemos limitar el dominio al campo real incorporando una condición a la definición de la incógnita:

CELDA 19

```
x = sym.Symbol('x', real=True)
sym.solve(x ** 4 - 2 ** 4, x)
```

```
[-2, 2]
```

o utilizando un método de Python para restringir las soluciones a un determinado intervalo:

CELDA 20

```
polinomio = x ** 4 + 2 * x ** 3 - 25 * x ** 2 - 26 * x + 120
solución = sym.solve(polinomio, x)
solución
```

```
[-5, -3, 2, 4]
```

CELDA 21

```
sol_intervalo = [v for v in solución if (v.is_real and sym.Or(v < 0 , v > 3))]
sol_intervalo
```

```
[-5, -3, 4]
```

donde hemos hecho uso de la disyunción lógica `Or`. Para restringir el dominio al utilizar `solveset()`, dicho dominio se debe pasar como argumento:

VERSIÓN PRELIMINAR

Celda 22

```
sym.solveset(x ** 4 - 2 ** 4, x, domain=sym.S.Reals)
```

$\{-2, 2\}$

Celda 23

```
sym.solveset(polinomio, x, sym.Interval(0, 3))
```

$\{2\}$

En la celda 21, `sym.S.Reals` es un «singleton» que representa los números reales desde $-\infty$ a $+\infty$.

Como vimos en un ejemplo precedente, cuando SymPy no es capaz de resolver una ecuación en forma analítica devuelve una expresión sin evaluar o una excepción:

Celda 24

```
sym.solve(sym.cos(x) - x, x)
```

```
NotImplementedError                       Traceback (most recent call last)
Cell In[24], line 1
----> 1 sym.solve(sym.cos(x) - x, x)

File ~/.config/jupyterlab-desktop/jlab_server/lib/python3.12/site-
    ↳ packages/sympy/solvers/solvers.py:1170, in solve(f, *symbols, **flags)
   1168         solution = _solve_undetermined(f[0], symbols, flags)
   1169     if not solution:
-> 1170         solution = _solve(f[0], *symbols, **flags)
   1171 else:
   1172     linear, solution = _solve_system(f, symbols, **flags)

File ~/.config/jupyterlab-desktop/jlab_server/lib/python3.12/site-
    ↳ packages/sympy/solvers/solvers.py:1729, in _solve(f, *symbols, **flags)
   1726 # ----------- end of fallback ----------------------------
   1728 if result is False:
-> 1729     raise NotImplementedError('\n'.join([msg, not_impl_msg % f]))
   1731 result = _remove_duplicate_solutions(result)
   1733 if flags.get('simplify', True):

NotImplementedError: multiple generators [x, cos(x)]
No algorithms are implemented to solve equation -x + cos(x)
```

En estos casos, podemos resolver la ecuación en forma numérica con la función `nsolve()`, a la que es necesario pasar como argumento la expresión a resolver, el símbolo que representa la incógnita, y un valor próximo a la solución que inicia la secuencia de aproximaciones numéricas:

Es habitual representar un sistema de ecuaciones lineales en forma matricial. SymPy permite resolver ecuaciones de esta forma, ya sea con un *solver* como `solve()` y también con factorizaciones como la descomposición LU o QR. Mostramos algunos ejemplos de esta forma de resolver ecuaciones matriciales en el capítulo ??.

Versión Preliminar

1.4. Funciones y cálculo diferencial e integral

En esta sección exploraremos superficialmente las capacidades de SymPy para el cálculo diferencial e integral, herramientas fundamentales en el análisis matemático. Comenzaremos con el concepto de funciones en el contexto de SymPy y a continuación abordaremos los cálculos de límites, derivadas e integrales. Finalmente mostraremos como evaluar expresiones que contienen sumas, productos y la posibilidad de expandir funciones en series de potencia o de Fourier.

1.4.1. Funciones

Así como podemos definir símbolos en SymPy, podemos también definir funciones. SymPy permite definir funciones abstractas (o indefinidas), y funciones definidas. Las funciones indefinidas son instancias de `sympy.Function` y, al igual que con los símbolos, el primer argumento del constructor de `Function` es el nombre que le asignamos.

Una función indefinida no se puede evaluar, ya que no tiene definido el cuerpo de la función. Al igual que con los símbolos, se pueden establecer atributos sobre funciones que permitan su manipulación aprovechando propiedades derivadas de estos atributos. Veamos algunos ejemplos.

Celda 1 de calculo.ipynb

```
import sympy as sym

x, y, z = sym.symbols('x y z')
f = sym.Function('f')
type(f)
```

```
sympy.core.function.UndefinedFunction
```

En la celda precedente, luego de importar el módulo `sympy`, definimos los símbolos x, y y z como hicimos anteriormente, y luego asignamos a la variable `f` una función indefinida. Esta función, además de carecer de cuerpo, no ha sido aplicada a ningún conjunto de símbolos que representen sus argumentos, pero podemos hacerlo sobre los símbolos ya definidos:

Celda 02

```
f(x)
```

$$f(x)$$

Celda 03

```
f(x, y, z)
```

$$f(x, y, z)$$

También podemos definir funciones indefinidas que dependan explícitamente de argumentos:

Celda 04

```
g = sym.Function('g')(x, y, z)
g
```

$g(x, y, z)$

Podemos acceder al conjunto de argumentos establecidos para f y g con el método `free_symbols`:

Celda 05

```
f.free_symbols, g.free_symbols
```

```
(set(), {x, y, z})
```

resultando el conjunto vacío para el caso de f. Se pueden establecer propiedades sobre funciones del mismo modo que como hicimos para símbolos:

Celda 06

```
f_real = sym.Function('f', real=True)
f_real(x).is_real
```

```
True
```

o también heredando propiedades de símbolos:

Celda 07

```
f_simbolo = sym.Symbol('f', real=True)
f_real_herencia = sym.Function(f_simbolo)
f_real_herencia.is_real
```

```
True
```

Las funciones indefinidas son útiles en diversos contextos, por ejemplo, para escribir ecuaciones diferenciales en las que no conocemos la forma de la función, y justamente resolver la ecuación es lo que permite establecer cuál es su cuerpo. Por otro lado, es posible también definir funciones definidas cuya dependencia operacional con sus argumentos se encuentra implementada, ya sea como subclase de `sympy.Function` o como combinación de las múltiples funciones definidas en SymPy. Por ejemplo, la función coseno se encuentra entre las disponibles en SymPy, y representa una función no evaluada que se puede aplicar a un símbolo, a un número o a una expresión:

Celda 08

```
type(sym.cos)
```

```
sympy.core.function.FunctionClass
```

Celda 09

```
sym.cos(sym.pi * 2)
```

1

Cuando el argumento es un símbolo, la función se mantiene sin evaluar excepto que sea posible su evaluación hacia un valor numérico o una expresión a partir de las propiedades de su argumento, tal como se muestra en la celda siguiente:

Celda 10

```
n = sym.Symbol('n', integer=True, odd=True)
sym.cos(n * sym.pi)
```

-1

Podemos definir el cuerpo de nuestra propia función al estilo usual de funciones de Python. En este caso, si tenemos previamente definidos como símbolos los argumentos de la función, lo que devuelve `return` en la función `f_py(x, y)` siguiente es una expresión válida de dichos símbolos:

Celda 11

```
def f_py(x, y):
    return (x + y) ** 3

print(type(f_py))
f_py(x, y)
```

```
<class 'function'>
```

$(x + y)^3$

En este ejemplo, `f_py` devuelve una expresión simbólica dado que los argumentos de esta función ya fueron definiddos como símbolos x y y. No obstante, podemos también evaluar numéricamente a `f_py` si le pasamos como argumentos dos números de un tipo de datos intrínseco del lenguaje, que en el siguiente ejemplo son del tipo `int`:

Celda 12

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

```
27
```

mientras que si los argumentos son símbolos, podemos realizar algún tipo de manipulación simbólica sobre `f_py`:

Celda 13

```
sym.expand(f_py(x, 3))
```

$x^3 + 9x^2 + 27x + 27$

Versión Preliminar

Por último, SymPy permite definir funciones anónimas, o funciones lambda, que no tienen nombre pero implementan el cuerpo de la función de forma que permitan su evaluación. Estas funciones son instancias de `sympy.Lambda` y reciben como primer argumento una tupla con los símbolos que representan las variables independientes, y como segundo argumento la expresión que representa el cuerpo de la función:

CELDA 14

```
h = sym.Lambda((x, y), (x + y) ** 2)
h
```

$$\left((x,\ y) \mapsto (x+y)^2\right)$$

CELDA 15

```
h(3, y)
```

$$(y+3)^2$$

1.4.2. Límites

SymPy es capaz de calcular simbólicamente el límite de una expresión. Para obtener

$$\lim_{x \to x_0} f(x)$$

se utiliza la función `sympy.limit()`, pasando como primer argumento la expresión a la que se le calcula el límite, como segundo argumento la variable, y por último el valor al que se aproxima la variable, tal como muestra la celda siguiente:

CELDA 16

```
sym.limit(sym.sin(x) / x, x, 0)
```

$$1$$

Naturalmente, en situaciones en que el límite dé origen a una expresión indefinida, no es válido evaluar este límite reemplazando el valor de la variable independiente por el valor al que tiende. Por ejemplo, en la celda siguiente queremos calcular el límite al que tiende $x^3 - 3x$ cuando $x \to \infty$ (que se representa en SymPy como `sym.oo`):

CELDA 17

```
f = x ** 3 - 3 * x
f.subs(x, sym.oo)
```

NaN

Dado que esta expresión resulta indeterminada, obtenemos un `NaN` (*Not-a-Number*) como respuesta. La forma correcta de realizar el cálculo es:

CELDA 18

```
sym.limit(f, x, sym.oo)
```

∞

Es posible también calcular límites simbólicos definidos sobre funciones abstractas, por ejemplo para el cálculo de derivadas por definición, que luego al utilizar funciones específicas obtenemos como resultado la derivada en cuestión:

CELDA 19

```
f = sym.Function('f')
x, h = sym.symbols('x h')
cociente_incremental = (f(x + h) - f(x)) / h
sym.limit(cociente_incremental.subs(f, sym.sin), h, 0)
```

$\cos(x)$

CELDA 20

```
sym.limit(cociente_incremental.subs(f, sym.exp), h, 0)
```

e^x

Es posible generar una expresión que contenga el límite sin evaluar, que permite una evaluación posterior de la expresión obtenida. Estos límites sin evaluar también son utilizados por SymPy cuando no puede obtener un resultado, por ejemplo, cuando la expresión contiene una función abstracta. Para dejar sin evaluar un límite se utiliza la función `sympy.Limit` (con «L» mayúscula) con idéntica sintaxis que `sympy.limit()`:

CELDA 21

```
derivada = sym.Limit(cociente_incremental, h, 0)
derivada
```

$$\lim_{h\to 0^+}\left(\frac{-f(x)+f(h+x)}{h}\right)$$

y para realizar la evaluación sobre una función definida, podemos utilizar primero el método `subs` que reemplaza la función abstracta f por una específica (en la celda siguiente, `sym.cos`), y realizamos la evaluación con `doit()`:

CELDA 22

```
derivada.subs(f, sym.cos).doit()
```

$-\sin(x)$

Para finalizar esta sección, mostraremos que también es posible calcular límites laterales, estableciando como cuarto parámetro el sentido desde el que nos aproximamos al valor límite del argumento: `'+'` para límite por la derecha, y `'-'` para el límite por izquierda:

Versión Preliminar

Celda 23

```
l_sup = sym.limit(1 / (x - 2), x, 2, '+')
l_inf = sym.limit(1 / (x - 2), x, 2, '-')
l_sup, l_inf
```

```
(oo, -oo)
```

1.4.3. Derivadas

Vimos en la sección anterior que podemos calcular la derivada de una función mediante su definición utilizando límites. SymPy permite calcular la derivada usando directamente la función `diff()`:

Celda 24

```
sym.diff(sym.cos(x) ** 2, x)
```

$$-2\sin\left(x\right)\cos\left(x\right)$$

o utilizando el método `.diff()` de una expresión:

Celda 25

```
expr = sym.cos(x) ** 2
expr.diff()
```

$$-2\sin\left(x\right)\cos\left(x\right)$$

Para realizar múltiples derivadas se puede repetir la variable las veces que sean necesarias:

Celda 26

```
sym.diff(sym.exp(x ** 2), x, x, x)
```

$$4x\left(2x^2+3\right)e^{x^2}$$

o simplemente indicar el orden de la derivación con un número entero:

Celda 27

```
sym.diff(sym.exp(x ** 2), x, 3)
```

$$4x\left(2x^2+3\right)e^{x^2}$$

También es posible realizar el cálculo de derivadas parciales de múltiples variables. Por ejemplo, para calcular

$$\frac{\partial^7}{\partial x^2 \partial y^2 \partial z^3} e^{xyz}$$

podemos escribir:

Versión Preliminar

Celda 28

```
sym.diff(sym.exp(x * y * z), x, 2, y, y, z, 4)
```

$$x^2y^2\left(x^4y^4z^4 + 20x^3y^3z^3 + 122x^2y^2z^2 + 256xyz + 144\right)e^{xyz}$$

Dado que el cálculo de derivadas es una tarea relativamente simple, es posible evaluar las derivadas de la mayoría de las funciones matemáticas definidas en SymPy, incluso las que involucran expresiones complicadas:

Celda 29

```
expresion = sym.hermite(x, 0)
expresion.diff(x)
```

$$\frac{2^x\sqrt{\pi}\operatorname{polygamma}\left(0, \frac{1}{2} - \frac{x}{2}\right)}{2\Gamma\left(\frac{1}{2} - \frac{x}{2}\right)} + \frac{2^x\sqrt{\pi}\log\left(2\right)}{\Gamma\left(\frac{1}{2} - \frac{x}{2}\right)}$$

Del mismo modo que para el caso de los límites, podemos crear una expresión que contenga una derivada sin evaluar:

Celda 30

```
derivada = sym.Derivative(sym.exp(x * y * z), x, 2, y, y, z, 4)
derivada
```

$$\frac{\partial^8}{\partial z^4\partial y^2\partial x^2}e^{xyz}$$

Cuando queremos efectivamente evaluar la derivada, invocamos el método `doit()`

Celda 31

```
derivada.doit()
```

$$x^2y^2\left(x^4y^4z^4 + 20x^3y^3z^3 + 122x^2y^2z^2 + 256xyz + 144\right)e^{xyz}$$

Desde luego, podemos utilizar la función o el método `diff()` sobre funciones abstractas:

Celda 32

```
f, g = sym.symbols('f g', cls=sym.Function)
f(x).diff()
```

$$\frac{d}{dx}f(x)$$

Celda 33

```
g(x, y, z).diff(x, x, y, 3)
```

$$\frac{\partial^5}{\partial y^3 \partial x^2} g(x, y, z)$$

Celda 34

```
f(x).diff(y)
```

$$0$$

Las derivadas de expresiones abstractas son útiles porque permiten utilizarlas para representar, por ejemplo, ecuaciones diferenciales. Para ilustrar un caso de uso, podemos resolver la ecuación diferencial

$$f''(x) - f(x) = \text{sen}(x)$$

Definimos entonces una expresión que representa esta ecuación:

Celda 35

```
ecdif = sym.Eq(f(x).diff(x, 2) - f(x), sym.sin(x))
ecdif
```

$$-f(x) + \frac{d^2}{dx^2} f(x) = \sin(x)$$

y pasamos esta expresión como argumento de la función `dsolve()`:

Celda 36

```
sym.dsolve(ecdif, f(x))
```

$$f(x) = C_1 e^{-x} + C_2 e^{x} - \frac{\sin(x)}{2}$$

Observamos que `dsolve()` devuelve una instancia de `Eq` debido a que, por lo general, las soluciones de una ecuación diferencial no se pueden resolver explícitamente por medio de una función. Como es usual, las constantes C_1 y C_2 deben determinarse a partir de las condiciones de contorno de la ecuación.

1.4.4. Integrales

El cálculo de integrales se realiza mediante la función `sympy.integrate()`, mientras que la representación formal de integrales se realiza mediante `sympy.Integral()` (que al igual que para los límites y las derivadas, pueden evaluarse explícitamente mediante el método `doit()`). De esta forma es posible calcular integrales definidas o indefinidas de una o más variables utilizando para ello algoritmos heurísticos para el reconocimiento de patrones, el algoritmo de Risch-Norman y consultas en tablas. De esta manera se pueden obtener las integrales de funciones algebraicas elementales y trascendentales, y una variedad de funciones especiales.

Versión Preliminar

Para el caso de las integrales indefinidas, es necesario pasar como argumento de `integrate()` la función integrando y un símbolo que representa la variable de integración:

Celda 37

```
sym.integrate(sym.cos(x), x)
```

$$\sin(x)$$

Debe observarse que el resultado no contiene la constante de integración. Para calcular integrales definidas, como segundo argumento hay que pasar una tupla con el símbolo correspondiente a la variable de integración, y los límites inferior y superior de la integral definida:

Celda 38

```
a, b = sym.symbols('a b')
sym.integrate(sym.cos(x), (x, a, b))
```

$$-\sin(a) + \sin(b)$$

Si `integrate()` es incapaz de determinar la primitiva del integrando, devolverá un objeto `Integral` no evaluado:

Celda 39

```
sym.integrate(x ** x, x)
```

$$\int x^x \, dx$$

Tanto para el caso de integrales definidas o indefinidas, es posible pasar como argumento de `integrate()` múltiples tuplas para evaluar una integral múltiple. En el ejemplo siguiente definimos primero una integral múltiple que dejamos sin evaluar, y a continuación invocamos el método mipdoit() para obtener el resultado:

Celda 40

```
integral = sym.Integral(sym.exp(-x ** 2 - y **2), (x, 0, sym.oo), (y, 0, sym.oo))
integral
```

$$\int_0^\infty \int_0^\infty e^{-x^2-y^2} \, dx \, dy$$

Celda 41

```
integral.doit()
```

$$\frac{\pi}{4}$$

SymPy provee rutinas para la evaluación numérica de integrales, que son especialmente útiles

Versión Preliminar

cuando no es posible determinar el resultado en forma analítica. Por ejemplo:

Celda 42

```
integral_numérica = sym.integrate(sym.exp(-x ** 2 * abs(x-1)), (x, 0, sym.oo))
integral_numérica
```

$$\int_0^\infty e^{-x^2|x-1|}\,dx$$

Celda 43

```
integral_numérica.evalf()
```

$$1{,}322$$

Por último, mencionamos también que SymPy es capaz de resolver integrales de línea mediante la función `line_integrate()`. En el ejemplo siguiente, definimos una curva paramétrica:

$$x(t) = e^t + 1,\ y(t) = e^t - 1$$

para t en el intervalo $[0, \ln 2]$, y posteriormente integramos la función $f(x, y) = x^2 + y^2$ a lo largo de esa curva:

Celda 44

```
x, y, t = sym.symbols('x y t', real=True)
C = sym.Curve([sym.E ** t + 1, sym.E ** t - 1], (t, 0, sym.ln(2)))
sym.line_integrate(x **2 + y ** 2, C, [x, y])
```

$$\frac{20\sqrt{2}}{3}$$

1.4.5. Sumas, productos y series

Es posible representar simbólicamente sumas y productos con SymPy a través de instancias de las clases `Sum` y `Product`. Los constructores de ambas clases reciben como primer argumento una expresión, y una tupla `(n, n1, n2)` como segundo argumento, siendo `n` un símbolo y `n1` y `n2` los límites inferior y superior para la suma o producto, respectivamente. Después de instanciar estos objetos, la evaluación de las sumas o productos se realiza con el método `doit()`. Por ejemplo, para el caso de sumas:

Celda 45

```
n = sym.symbols('n', integer=True)
S1 = sym.Sum((-1) ** n / (n + 1), (n, 0, sym.oo))
S1
```

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{n+1}$$

Versión Preliminar

Celda 46

```
S1.doit()
```

$$\log(2)$$

Celda 47

```
S2 = sym.Sum(1 / n ** 2, (n, 1, sym.oo))
S2
```

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

Celda 48

```
S2.doit()
```

$$\frac{\pi^2}{6}$$

Celda 49

```
S3 = sym.Sum(1 / n ** 4, (n, 1, 10))
S3
```

$$\sum_{n=1}^{10} \frac{1}{n^4}$$

Celda 50

```
S3.doit()
```

$$\frac{43635917056897}{40327580160000}$$

Podemos notar que en las celdas 45 y 46 utilizamos ∞ (`sym.oo`) como límite superior, y claramente la operación no puede evaluarse sumando explícitamente. SymPy realiza el cálculo en forma analítica, incluso en casos en los que el sumando contiene variables simbólicas además del índice de suma, como en el ejemplo a continuación:

Celda 51

```
S4 = sym.Sum(x ** n / sym.factorial(n), (n, 1, sym.oo))
S4.doit().simplify()
```

$$e^x - 1$$

Para el caso de los productos, es posible evaluar expresiones con valores acotados del índice:

Celda 52

```
P = sym.Product(1 / n, (n, 1, 10))
P
```

$$\prod_{n=1}^{10} \frac{1}{n}$$

Celda 53

```
P.doit()
```

$$\frac{1}{3628800}$$

Sin embargo, para productos de infinitos factores resulta muy difícil obtener en forma directa una expresión cerrada. Si consideramos el producto de Wallis:

Celda 54

```
W = sym.Product(2 * n/(2 * n - 1) * 2 * n / (2 * n + 1), (n, 1, sym.oo))
W.doit()
```

$$\prod_{n=1}^{\infty} \frac{4n^2}{(2n-1)(2n+1)}$$

vemos que el cálculo directo falla y devuelve la expresión sin evaluar. Podemos recorrer un camino indirecto si definimos el producto de Wallis con un número finito de factores

Celda 55

```
m = sym.Symbol('m', integer=True, positive=True)
wallis = sym.Product(2 * n / (2 * n - 1) * 2 * n / (2 * n + 1), (n, 1, m))
WE = wallis.doit()
WE
```

$$\frac{2^{-2m}4^m m!^2}{\left(\frac{1}{2}\right)^{(m)}\left(\frac{3}{2}\right)^{(m)}}$$

y luego evaluamos el límite cuando $m \to \infty$:

Celda 56

```
sym.limit(WE, m, sym.oo)
```

$$\frac{\pi}{2}$$

lo que nos produce el resultado correcto.

Dada una función, se puede obtener su expansión en serie de potencias mediante el método `series()` o invocando a la función `series()`, indicando en ambos casos el valor alrededor del cual

Versión Preliminar

se expande la serie, y el orden hasta el que se obtienen los términos:

Celda 57

```
f = sym.cos(x)
f.series(n=10)
```

$$1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \frac{x^8}{40320} + O\left(x^{10}\right)$$

Celda 58

```
sym.series(f, x0=sym.pi)
```

$$-1 + \frac{(x-\pi)^2}{2} - \frac{(x-\pi)^4}{24} + O\left((x-\pi)^6 ; x \to \pi\right)$$

Al omitir el valor de `x0` se asume que $x_0 = 0$, mientras que si omitimos el valor para `n`, toma por defecto el valor 6. El término $O(x^{10})$ representa todos los términos de la suma con potencia mayor o igual a x^{10}. Esta notación también se utiliza para otros puntos límite arbitrarios (por ejemplo $x \to \pi$ en la celda 58). Para evitar incorporar ese término a la expresión se utiliza el método `removeO()`:

Celda 59

```
f.series(n=10).removeO()
```

$$\frac{x^8}{40320} - \frac{x^6}{720} + \frac{x^4}{24} - \frac{x^2}{2} + 1$$

El método (o la función) `series()` devuelve una expresión. Pero si especificamos `n=None`, lo que obtenemos como resultado es un generador, que nos permite colectar, por ejemplo, un número determinado de términos de la serie:

Celda 60

```
serie = f.series(n=None)
[next(serie) for i in range(7)]
```

```
[1, -x**2/2, x**4/24, -x**6/720, x**8/40320, -x**10/3628800, x**12/479001600]
```

Para finalizar esta sección, mostramos que SymPy es capaz de generar expansiones en series de Fourier. Si $f(x)$ está definida sobre el intervalo (a, b), la serie de Fourier correspondiente es:

$$\frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n \cos\left(\frac{2n\pi x}{L}\right) + b_n \operatorname{sen}\left(\frac{2n\pi x}{L}\right)\right]$$

Versión Preliminar

donde:

$$
\begin{aligned}
L &= b - a \\
a_0 &= \frac{2}{L}\int_a^b f(x)\,dx \\
a_n &= \frac{2}{L}\int_a^b f(x)\,\cos\left(\frac{2n\pi x}{L}\right)\,dx \\
b_n &= \frac{2}{L}\int_a^b f(x)\,\mathrm{sen}\left(\frac{2n\pi x}{L}\right)\,dx
\end{aligned}
$$

Dada entonces una función, el método `fourier_series()` realiza la expansión. Podemos visualizar los términos hasta un cierto orden por medio del método `truncate()`:

Celda 61

```
f = x ** 2
g = x
Ff = sym.fourier_series(f, (x, -sym.pi, sym.pi))
Fg = sym.fourier_series(g, (x, -sym.pi, sym.pi))
Ff.truncate(n=5)
```

$$-4\cos(x) + \cos(2x) - \frac{4\cos(3x)}{9} + \frac{\cos(4x)}{4} + \frac{\pi^2}{3}$$

Celda 62

```
Fg.truncate(n=4)
```

$$2\sin(x) - \sin(2x) + \frac{2\sin(3x)}{3} - \frac{\sin(4x)}{2}$$

1.5. Evaluaciones numéricas y gráficos

Hemos visto en las secciones precedentes una introducción muy superficial de las capacidades de SymPy para realizar cálculo simbólico. Las posibilidades que ofrece este módulo son mucho más extensas y profundas de lo que podemos abarcar en este capítulo, por lo que invitamos a quienes estén interesados en conocer más a leer la documentación.

Sin embargo, es muy común que en alguna instancia del cálculo necesitemos ver números, o gráficos, generados por las expresiones simbólicas. Para realizar evaluaciones numéricas de expresiones podemos usar el método `evalf()` (que mencionamos al final de la primera sección para evaluar constantes) o la función `N()`.

Celda 63

```
f = sym.sin(x) + sym.pi
f_numérico = f.evalf(subs={x:1})
print(f"f(1) = {f_numérico}")
```

```
f(1) = 3.98306363839769
```


Versión Preliminar

Se puede especificar la precisión de las evaluaciones numéricas (por defecto establecido en 15 dígitos decimales) pasando como primer argumento un entero positivo:

CELDA 64

```
f_preciso = f.evalf(50, subs={x:1})
print(f"f(1) = {f_preciso}")
```

```
f(1) = 3.9830636383976897451151457049098018838197324601735
```

CELDA 65

```
sym.N(f.subs({x:1}), 50)
```

3,9830636383976897451151457049098018838197324601735

De esta forma se pueden evaluar expresiones en determinados valores de algún símbolo que contenga la expresión que resulta reemplazado mediante `subs()`. Esto no resulta práctico cuando necesitamos realizar evaluaciones numéricas masivas. Para ello es conveniente utilizar la función `lambdify()`, que permite realizar cálculos vectorizados veloces por medio del módulo NumPy. Entre otras aplicaciones, estas evaluaciones masivas permiten convertir expresiones simbólicas en arrays numéricos para su visualización con Matplotlib (aunque SymPy tiene su propia función `plot()`, no resulta tan versátil como el popular módulo de visualización). Aprovecharemos las expansiones en series de Fourier de la sección anterior para realizar evaluaciones numéricas vectorizadas y visualizar estos resultados.

En la celda 66 desarrollamos un pequeño script para ver cómo se aproximan a la función $f(x) = x$ las sucesivas sumas parciales de la expansión en serie de Fourier con n términos. Primero importamos los módulos Matplotlib y NumPy (para realizar el gráfico y tener soporte para el uso de arrays, respectivamente).

A continuación asignamos a `Fx` la transformada de Fourier, entre $-\pi$ y π, de la función f, y construimos un array de 500 puntos equiespaciados entre estos límites, asignándolo a `x_array`. Inmediatamente después graficamos este array en función de si mismo (lo que representa $f(x) = x$).

Para generar las sumas parciales incoporando un número creciente de términos, recorremos un bucle asignando a la variable `n` los valores de la lista `[1, 3, 5, 7]`. Dentro de este bucle, la primera instrucción consiste en truncar la expansión de Fourier en el número de términos indicados por `n`. Este paso convierte un objeto del tipo `FourierSeries` (devuelto por `fourier_series()`) en una instancia de `Expr`, ya que `lambdify()` acepta expresiones que representan funciones como argumento, pero no un objeto del tipo `FourierSeries`.

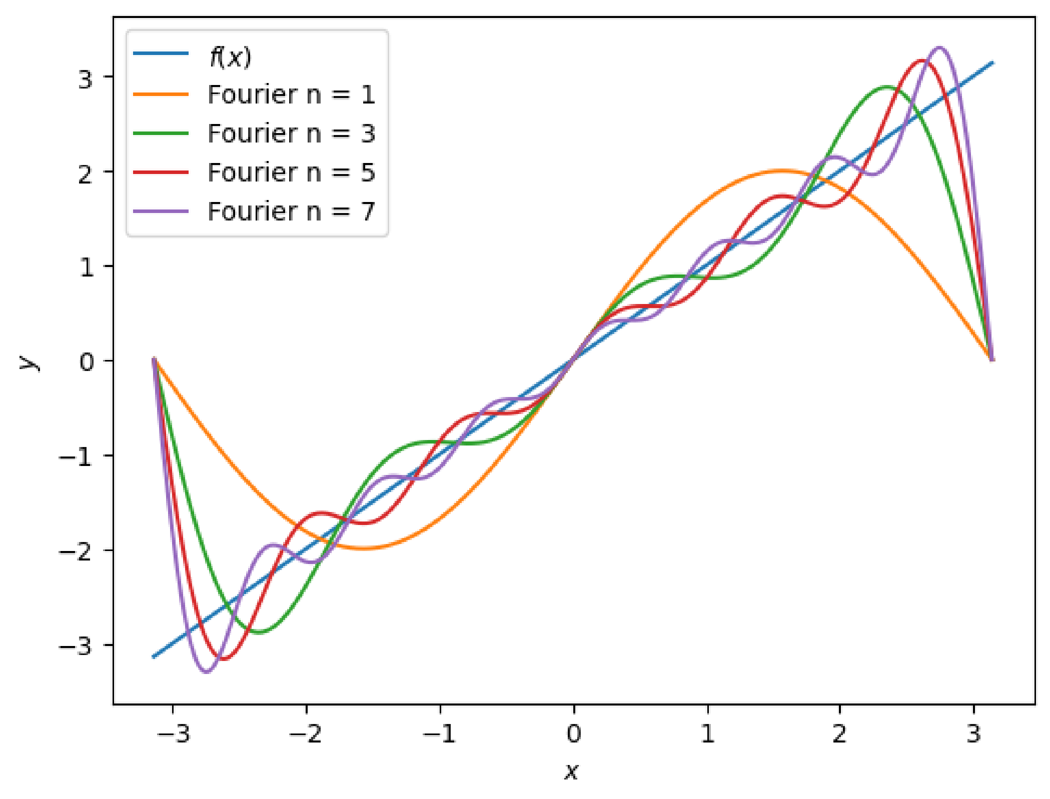
Una vez truncada la serie, le pasamos a `lambdify()` el símbolo que representa la variable que evaluaremos numéricamente (`x`), la expresión que representa la función a evaluar (`F_trunc`) y el módulo que se utilizará para realizar la evaluación numérica de la expresión simbólica. En este caso usaremos el módulo `'numpy'`, pero son posibles otras opciones como `'math'`, que es el módulo por defecto de Python, o `'mpmath'` para cálculos de precisión arbitraria. Asignamos a la variable `F_numerica` la función que devuelve `lambdify`.

La última instrucción dentro del bucle construye la representación gráfica, evaluando en forma vectorial `F_numérica` para cada valor del array `x_array`. Luego completamos el gráfico agregando las etiquetas de los ejes y la leyenda que informa los colores asignados a cada `n`.

Celda 66

```
import matplotlib.pyplot as plt
import numpy as np
Fx = sym.fourier_series(x, (x, -sym.pi, sym.pi))
x_array = np.linspace(-np.pi, np.pi, 500)
plt.plot(x_array, x_array, label=r'$f(x)$')
for n in [1, 3, 5, 7]:
    F_trunc = Fx.truncate(n=n)
    F_numérica = sym.lambdify(x, F_trunc, modules='numpy')
    plt.plot(x_array, F_numérica(x_array), label=f"Fourier n = {n}")

plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.legend()
plt.show()
```



Parte IV
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [2].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!

Bibliografía

[1] Aaron Meurer et al. «SymPy: symbolic computing in Python». En: *PeerJ Computer Science* 3 (ene. de 2017), e103. DOI: 10.7717/peerj-cs.103. URL: https://doi.org/10.7717/peerj-cs.103.

[2] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: https://www.python.org/dev/peps/pep-0020/.