SymPy (autore: Vittorio Albertoni)

Premessa

La differenza tra il calcolo numerico e il calcolo simbolico sta nel fatto che il calcolo numerico tratta solo numeri ed esprime sempre un risultato numerico, molto spesso approssimato, mentre il calcolo simbolico tratta numeri e simboli ed esprime un risultato numerico solo se si tratta di un numero esatto e non approssimato, altrimenti anche il risultato è espresso in simboli.

Così, se lavorando con Python ed avendo importato il modulo `math` per il calcolo numerico, calcoliamo la radice quadrata di 12,25 otteniamo il risultato 3,5. Sempre lavorando con Python ma avendo importato il modulo `sympy` per il calcolo simbolico, dal momento che 3,5 è esattamente la radice quadrata di 12,25, otteniamo ancora il risultato 3,5.

Se però calcoliamo, per esempio, la radice quadrata di 8, con il calcolo numerico arriviamo al risultato approssimato 2.8284271247461903 ma con il calcolo simbolico arriviamo al risultato 2 per la radice quadrata di 2 (dalla scomposizione $\sqrt{8} = \sqrt{4}\sqrt{2} = 2\sqrt{2}$): tutto quanto si può calcolare numericamente in maniera esatta ($\sqrt{4}$) viene calcolato e il resto ($\sqrt{2}$) rimane espresso in simboli.

Se, come nel caso della radice di 8, il simbolo è numerico ($\sqrt{2}$) abbiamo modo di forzare, se serve, la sua traduzione in numero approssimato; se il simbolo è un vero e proprio simbolo, come x, a o β, esso rimane tale e quale.

Come avviene, per esempio, nel calcolo del prodotto notevole $(\sqrt{2} + a)(\sqrt{2} - a)$ che porta al risultato $2 - a^2$, dove il prodotto tra i due simboli di numeri irrazionali diventa un numero intero (2) e rimane in simbolo la parte simbolica vera e propria (a^2).

Il modulo SymPy ci dà modo di inserire il calcolo simbolico in script Python fatti da noi in modo da poter eseguire elaborazioni per le quali dovremmo altrimenti ricorrere a software dedicati al calcolo simbolico.

Obiettivo di questo manualetto è innanzi tutto illustrare le più ricorrenti funzioni di elaborazione che ci mette a disposizione SymPy e poi far vedere come si può fare questo inserimento.

La descrizione completa di tutto ciò che si può fare con SymPy la troviamo all'indirizzo *https://www.sympy.org* nella guida completa a SymPy, disponibile anche in formato PDF, purtroppo solo in lingua inglese.

Indice

1 Installazione **3**

2 Importazione **3**

3 Operazioni di base **4**
3.1 Dichiarazione dei simboli 4
3.2 Costruzione delle espressioni 4
3.3 Sostituzione di simboli 4
3.4 Dall'espressione numerica al numero 5
3.5 Visualizzazione 5

4 Elaborazione delle espressioni **6**
4.1 Semplificazioni 6
4.2 Espansioni polinomiali 7
4.3 Fattorizzazioni polinomiali 7
4.4 Operazioni sui polinomi 7

5 Calcolo **7**
5.1 Derivate 8
5.2 Integrali 9
5.3 Limiti 9

6 Risolutori **10**

7 Matrici **11**

8 Integrazione con Python **12**
8.1 Risoluzione di equazioni 12
8.2 Strumenti di analisi 17

9 Avvertenza conclusiva **19**

1 Installazione

La casa di SymPy si trova all'indirizzo *https://www.sympy.org* e vi troviamo ottima documentazione per poterlo utilizzare oltre a quanto necessario per installarlo in maniera ruspante.

All'indirizzo *https://live.sympy.org* troviamo anche una shell per la sua sperimentazione online.

Chi voglia avventurarsi nell'installazione ruspante da tarball del source sappia che le nuove versioni di SymPy (quelle dalla 1.0) richiedono che sia installato anche il pacchetto `mpmath`.

Per una installazione a portata di dilettante è meglio ricorrere a Conda o a pip[1]in modo che automaticamente siano installate tutte le dipendenze.

Chi ha la fortuna di lavorare con sistema operativo Linux può ricorrere al gestore di pacchetti Synaptic, dove sicuramente trova SymPy come pacchetto, dipendenze incluse.

2 Importazione

Per utilizzare il modulo SymPy non basta averlo installato ma occorre anche importarlo.

Come avviene per tutti i moduli Python abbiamo praticamente due modi per farlo:
. la formula `from sympy import *` ci mette a disposizione tutto il contenuto del modulo e possiamo chiamarne le funzioni scrivendole direttamente nello script;
esempio:

```
from sympy import *
sqrt(9)
```

Con questa formula potremmo importare solo alcune funzioni contenute nel modulo indicandone il nome al posto dell'asterisco.
Se, per esempio, nel nostro script ci bastasse disporre della sola funzione `sqrt()` potremmo agire così

```
from sympy import sqrt
sqrt(9)
```

. la formula `import sympy` ci dà modo di importare il modulo come oggetto: in questo caso possiamo chiamare le funzioni che ci servono come sue funzioni membro;
esempio:

```
import sympy
sympy.sqrt(9)
```

Con questa formula possiamo battezzare l'oggetto con un nome più breve in modo da avere meno da scrivere:

```
import sympy as s
s.sqrt(9)
```

Ciascuno si può regolare come gli viene meglio.

La seconda formula è molto comoda per la code completion: se lavoriamo con un editor che la contempla, come l'IDLE, non appena abbiamo aggiunto il punto al nome dell'oggetto ci viene proposto l'elenco delle funzioni membro e al suo interno possiamo scegliere quella che ci interessa.

La seconda formula, infine, è d'obbligo se nello script dovremo utilizzare alternativamente funzioni denominate allo stesso modo nei moduli importati, come avviene, per esempio, per la funzione `sqrt()`, che ha lo stesso nome nel modulo `math` e nel modulo `sympy`.

```
import sympy
import math
sympy.sqrt(9)
math.sqrt(9)
```

[1]A chi voglia saperne di più consiglio la lettura dell'allegato «python_anaconda» al mio articolo «Software libero per data scientists» dell'aprile 2019, archiviato nella categoria Software libero del mio blog *www.vittal.it* e/o dell'allegato «mondo_python» al mio articolo «Python per tutti» del febbraio 2017, archiviato nella categoria Programmazione dello stesso blog.

3 Operazioni di base

3.1 Dichiarazione dei simboli

I simboli non numerici che intendiamo usare nelle nostre elaborazioni vanno preventivamente dichiarati.

La sintassi per farlo è:

`<simbolo>, <simbolo>, ... = symbols('<simbolo> <simbolo> ...')`

dove <simbolo> è una lettera dell'alfabeto latino direttamente prodotta dalla tastiera o una lettera dell'alfabeto greco prodotta con le dizioni `alpha, beta, gamma, delta, epsilon, zeta, eta, theta, iota, kappa, lamda, mu, nu, xi, omicron, pi, rho, sigma, tau, upsilon, phi, chi, psi, omega`.

Esempio:

```
a, beta, x, w = symbols('a beta x w')
```

rende disponibili per le nostre elaborazioni i simboli a, β, x e w.

I simboli di SymPy sono oggetti dotati di funzioni membro: se nell'IDLE scriviamo un simbolo precedentemente dichiarato e lo facciamo seguire da un punto (`.`) vediamo comparire l'elenco delle sue funzioni membro.

3.2 Costruzione delle espressioni

Le espressioni matematiche si compongono utilizzando i simboli dichiarati, funzioni riconosciute da SymPy e numeri, combinati con gli operatori aritmetici del linguaggio Python (`+ - * / **`).

L'espressione può essere inserita in una variabile, creando così un oggetto espressione dotato di funzioni membro.

Esempio:

```
e = sin(x)**2 + cos(x)**2 - x + 2*x*y
```

crea l'oggetto `e` contenente l'espressione matematica $sin^2(x) + cos^2(x) - x + 2xy$

Possiamo creare un'espressione da una stringa utilizzando la sintassi:

```
<nome_espressione> = sympify('<stringa>')
<nome_espressione> = sympify(<nome_stringa>)
```

La stringa può essere un nome o una frase qualsiasi ma, ovviamente, a noi servirà che sia un'espressione matematica.

Nel primo caso scriviamo la stringa tra le parentesi e tra apici: cosa che potevamo fare senza disturbare stringhe e sympify.

Nel secondo caso importiamo una variabile stringa già esistente con un suo nome.

Attenzione a non confondere la funzione `sympify` con la funzione `simplify` che, come vedremo, fa tutt'altre cose.

Questa funzione ci dà modo di acquisire come espressione matematica elaborabile in SymPy un'espressione generata al di fuori di SymPy, per esempio introdotta dall'utente da tastiera come risposta all'invito di input da uno script Python.

Esempi:

```
e = sympify('4*x**2-5')
```

crea l'oggetto `e` contenente l'espressione matematica $4x^2 - 5$.

```
e = sympify(formula)
```

crea l'oggetto `e` che contiene l'espressione contenuta in una stringa denominata formula.

3.3 Sostituzione di simboli

Le espressioni matematiche di SymPy sono immutabili.

E' tuttavia possibile generare altre espressioni matematiche sostituendo un simbolo di una espressione matematica con un altro simbolo o valore numerico con la sintassi

`<espressione>.subs(<simbolo_da_sostituire>, <simbolo_o_valore_in_sostituzione>).`

Esempio:
`e.subs(x, y)`
applicato all'espressione `e` dell'ultimo esempio del paragrafo precedente genera l'espressione $4y^2 - 5$.
`e.subs(x, 2)`
sempre applicato all'espressione `e`, genera l'espressione matematica $4 * 2^2 - 5$, cioè il numero 11.
In ogni caso l'espressione `e` rimane $4x^2 - 5$.
Se vogliamo salvare l'espressione modificata dobbiamo assegnare il risultato della sostituzione ad una nuova espressione, ad esempio
`new_e = e.subs(x, y)`
crea una nuova espressione, chiamata `new_e`, $4y^2 - 5$.

3.4 Dall'espressione numerica al numero

Quando SymPy compie calcoli su numeri fornisce il risultato in una espressione numerica.

Se questo risultato è un numero che esiste, l'espressione numerica coincide con il numero stesso e non ci accorgiamo nemmeno che è un'espressione numerica.

Se il risultato è un numero che non esiste e che non si può calcolare, come avviene per i numeri irrazionali, l'espressione numerica è, in realtà, un simbolo.

Tutto ciò lo abbiamo visto nella premessa, esemplificando il calcolo della radice quadrata di 8.

La funzione `sqrt(8)` di Sympy fornisce il risultato $2\sqrt{2}$.

L'oggetto che contiene questo risultato ha una funzione membro, che si chiama `evalf()`, che forza la traduzione dell'espressione numerica simbolica in numero approssimato. Tra le parentesi la funzione accetta un numero intero corrispondente al numero di cifre decimali, virgola compresa, che si vogliono vedere. Se non si indica questo parametro, per default vengono forniti la virgola e 14 cifre decimali.
Esempio:
`a = sqrt(8)` attribuisce alla variabile `a` il risultato $2\sqrt{2}$
`a.evalf()` fornisce il risultato 2.82842712474619
`a.evalf(4)` fornisce il risultato 2.828

Ogni espressione ha la funzione membro evalf(); per esempio
`(sqrt(2)*x).evalf()` fornisce il risultato $1.4142135623731x$.

Agli stessi risultati si perviene utilizzando la funzione N() i cui argomenti sono l'espressione da valorizzare e, eventualmente, il numero di decimali richiesti.
L'esempio appena fatto potrebbe essere scritto `N(sqrt(8)*x)` e il precedente `N(a,4)`.

Bene sapere che in SymPy esiste anche la funzione `eval()` che fornisce il valore di una espressione passata come argomento in forma di stringa.
Praticamente `eval()` fa ciò che fa `sympify()`, che abbiamo visto nel paragrafo 3.2.
Sia `eval()` che `sympify()` hanno la funzione membro `evalf()` per eventualmente forzare i risultati numerici.
Esempio:
`eval('sqrt(8)').evalf(16)` fornisce il risultato 2.828427124746190
`sympify('sqrt(8)').evalf()` fornisce il risultato 2.82842712474619

3.5 Visualizzazione

Per visualizzare il contenuto di variabili e i risultati che via via otteniamo dalle nostre elaborazioni, dal momento che lavoriamo in Python, abbiamo ovviamente a disposizione la funzione `print()`, con la quale, tuttavia, otteniamo la visualizzazione delle espressioni così come si inseriscono da tastiera.

SymPy è però dotato della funzione di pretty printing `pprint()`, con la quale otteniamo visualizzazioni più adeguate.

La differenza la vediamo in questo esempio

```
>>> i = Integral(sqrt(1/sin(x)**2), x)
>>> i
Integral(sqrt(sin(x)**(-2)), x)
>>> print(i)
Integral(sqrt(sin(x)**(-2)), x)
>>> pprint(i)
⌠
⎮      ⎽⎽⎽⎽⎽⎽⎽⎽
⎮     ╱    1
⎮    ╱  ──────  dx
⎮   ╱        2
⎮ ╲╱    sin (x)
⌡
```

Abbiamo inserito nella variabile `i` l'espressione per scrivere un integrale e vediamo come l'ha acquisita SymPy: la frazione con la potenza al denominatore è sostituita dal solo denominatore con potenza negativa.
Se stampiamo il contenuto della variabile `i` con la funzione `print()` otteniamo tale e quale questo contenuto, scritto secondo la sintassi Python.
Se lo stampiamo con la funzione `pprint()` otteniamo la visualizzazione del contenuto secondo la ricorrente simbologia matematica.

Se lavoriamo in SymPy, inoltre, sia la funzione `print()` sia la funzione `pprint()` possono produrre il codice Latex per la scrittura dell'espressione con la sintassi
`print(latex(<espressione>))`
`pprint(latex(<espressione>))`
Tornando alla nostra espressione `i` possiamo ottenere il codice Latex per scriverla come si deve

```
>>> print(latex(i))
\int \sqrt{\frac{1}{\sin^{2}{\left(x \right)}}}\, dx
```

codice che, inserito in un editor Latex tra due simboli `$`, produce questo risultato

$$\int \sqrt{\frac{1}{\sin^2(x)}}\, dx$$

4 Elaborazione delle espressioni

4.1 Semplificazioni

Quando creiamo un'espressione dalla tastiera o importando una stringa, SymPy la acquisisce apportando già le più immediate semplificazioni.

Per esempio, se digitiamo la seguente istruzione
`e = sin(x)**2 + cos(x)**2 - x + 2*x*y + 4*x - (x*y)/2`
l'espressione che viene creata da SymPy è
$sin^2(x) + cos^2(x) + 3x + \frac{3}{2}xy$
nella quale abbiamo già $3x$ al posto di $-x + 4x$ e $\frac{3}{2}xy$ al posto di $2xy - \frac{xy}{2}$

Ma la nostra espressione contiene un'altra possibile semplificazione, meno immediata: la somma dei quadrati di $sin(x)$ e $cos(x)$, infatti, è uguale a 1, qualunque sia il valore di x.

Per eseguire le semplificazioni meno immediate SymPy ci offre la funzione `simplify()` il cui argomento può essere un'espressione creata al momento o il nome di un'espressione esistente.
Esempi:

`simplify(sin(x)**2 + cos(x)**2)`
fornisce il risultato 1;
`simplify(e)` con riferimento all'espressione `e` del precedente esempio
fornisce il risultato
$1 + 3x + \frac{3}{2}xy$.

4.2 Espansioni polinomiali

Per espandere un'espressione polinomiale abbiamo la funzione `expand()`, il cui argomento è un'espressione inserita direttamente o richiamata.
`expand((a+b)**3)` produce l'espressione $a^3 + 3a^2b + 3ab^2 + b^3$
Se abbiamo l'espressione `e = (x - 3)*(x + 1)`,
`expand(e)` produce l'espressione $x^2 - 2x - 3$
`expand((a+b)*(a-b))` produce l'espressione $a^2 - b^2$

4.3 Fattorizzazioni polinomiali

La fattorizzazione è l'opposto dell'espansione e la realizziamo con la funzione `factor()`, il cui argomento è un'espressione inserita direttamente o richiamata.
factor(a**2 + 2*a*b + b**2) produce l'espressione $(a + b)^2$
Se abbiamo l'espressione `e = x**2 - 2*x -3`,
factor(e) produce l'espressione $(x - 3)(x + 1)$

4.4 Operazioni sui polinomi

Per sommare, sottrarre, moltiplicare ed elevare a potenza possiamo usare i soliti operatori `+ - * e **` di Python.
Esempi:
Siano A e B i due polinomi
`A =` $x^4 - 2x^3 - 8x + 16$
`B =` $2x^4 + 8x$
`A + B` ritorna $3x^4 - 2x^3 + 16$
`A - B` ritorna $-x^4 - 2x^3 - 16x + 16$
`A * B`, assoggettando il risultato a expand(), ritorna $2x^8 - 4x^7 - 8x^5 + 16x^4 - 64x^2 + 128x$
`B ** 2`, assoggettando il risultato a expand(), ritorna $4x^8 + 32x^5 + 64x^2$

Per la divisione abbiamo a disposizione la funzione `div()`, i cui argomenti, separati da virgola, sono il polinomio dividendo e il polinomio divisore, che ritorna una tupla contenente il quoziente e il resto.
Esempio:
Siano DD e D i due polinomi
`DD =` $x^6 + 3x^4 - 4x + 2$
`D =` $x^2 - x + 3$
`div(DD,D)` ritorna $\left(x^4 + x^3 + x^2 - 2x - 5, 17 - 3x\right)$
per cui il quoziente è $x^4 + x^3 + x^2 - 2x - 5$
e il resto è $17 - 3x$

5 Calcolo

SymPy conosce tutte le funzioni matematiche che possiamo immaginare: per rendersene conto basta consultare la documentazione che troviamo all'indirizzo *https://live.sympy.org*, specificamente il capitolo dedicato al Functions Module.

Al mio lettore dilettante con una preparazione matematica basica ricordo le più ricorrenti:
. tutte le funzioni trigonometriche,
. tutte le funzioni iperboliche,

. `Abs()` per il valore assoluto,
. la funzione esponenziale `exp()`,
. `log()` oppure `ln()` per il logaritmo naturale,
. `log(x, b)` oppure `ln(x, b)` per il logaritmo in una qualsiasi base `b`,
. `sqrt()` per la radice quadrata,
. `root(x, n)` per la radice ennesima,
. `factorial()` per il fattoriale.

SymPy conosce pure le due costanti numeriche fondamentali:
. `pi` che è la costante π,
. `E` che è la base dei logaritmi naturali e.

Attenzione a parte meritano poi i seguenti strumenti di calcolo simbolico. Anche per questi mi limito ai più ricorrenti.

5.1 Derivate

Per calcolare la derivata usiamo la funzione `diff()` che accetta due argomenti:
. la funzione da derivare, scritta tale e quale o richiamata attraverso una variabile che la contiene,
. la variabile rispetto alla quale calcolare la derivata (se la funzione contiene un solo simbolo possiamo omettere questo argomento).
Esempi:
`diff(log(x))` ritorna $\frac{1}{x}$
`diff(a*log(x), x)` ritorna $\frac{a}{x}$
`diff(a*log(x), a)` ritorna $log(x)$
se abbiamo la variabile `v` che contiene l'espressione $log(x)sin^2(x)$
diff(v, x) ritorna $2\log(x)\sin(x)\cos(x) + \frac{\sin^2(x)}{x}$

La funzione `diff()` è funzione membro dell'oggetto espressione e può essere richiamata come tale: in questo caso accetta il solo argomento relativo alla variabile di derivazione.
Esempi:
I quattro esempi che abbiamo appena visto possono essere risolti anche con questa sintassi:
```
log(x).diff()
(a*log(x)).diff(x)
(a*log(x)).diff(a)
```
`v.diff(x)` o, essendoci un solo simbolo variabile, semplicemente `v.diff()`.
Attenzione a racchiudere sempre tra parentesi eventuali espressioni complesse, ad evitare che la funzione venga richiamata solo in relazione all'ultimo pezzo inserito.

Tutto ciò che abbiamo visto finora calcola la derivata prima.

Le derivate di ordine successivo possiamo calcolarle derivando via via la derivata precedente, ma SymPy ci offre di meglio. Infatti possiamo calcolare direttamente una derivata di un ordine qualsiasi inserendo più volte la variabile di derivazione, separando con virgole, come argomento di `diff()` oppure inserendo il numero corrispondente al grado di derivazione dopo la variabile di derivazione, separato da virgola.
Esempi:
`diff(x**3, x, x)` ritorna $6x$ che è la derivata seconda di x^3
`diff(x**3, x, 3)` ritorna 6 che è la derivata terza di x^3

Nel caso di funzioni di più variabili possiamo ottenere le derivate parziali inserendo come argomento della funzione `diff()` solo la variabile di derivazione e possiamo ottenere la derivata totale inserendo come argomenti della funzione `diff()` tutte le variabili.
Esempi:
`diff(cos(x)*sin(y), x)` ritorna $-sin(x)sin(y)$ che è la derivata parziale rispetto a x
`diff(cos(x)*sin(y), y)` ritorna $cos(x)cos(y)$ che è la derivata parziale rispetto a y
`diff(cos(x)*sin(y), x, y)` ritorna $-sin(x)cos(y)$ che è la derivata totale.

5.2 Integrali

Per il calcolo dell'integrale usiamo la funzione `integrate()`.

Integrale indefinito

L'integrale indefinito è l'antiderivata e ci fornisce la funzione primitiva.

Per calcolarlo passiamo alla funzione `integrate()` due argomenti:
. la funzione da integrare, direttamente o richiamando la variabile che la contiene,
. la variabile rispetto alla quale integrare (se la funzione contiene un solo simbolo possiamo omettere questo argomento).
Esempi:
`integrate(1/x)` ritorna $log(x)$
`integrate(a/x, x)` ritorna $alog(x)$
`integrate(log(x), a)` ritorna $alog(x)$
Se abbiamo la variabile v che contiene l'espressione $2\log(x)\sin(x)\cos(x) + \frac{\sin^2(x)}{x}$
`integrate(v, x)` ritorna $\frac{-log(x)cos(2x)}{2} + \frac{log(x)}{2}$ semplificabile in $log(x)sin^2(x)$.
A seconda della potenza del nostro processore il calcolo di questo integrale può richiedere anche alcuni minuti.

La funzione `integrate()` è funzione membro dell'oggetto espressione e può essere richiamata come tale: in questo caso accetta il solo argomento relativo alla variabile di derivazione.
Esempi:
I quattro esempi che abbiamo appena visto possono essere risolti anche con questa sintassi:

```
(1/x).integrate()
(a/x).integrate(x)
log(x).integrate(a)
v.integrate(x)
```

Integrale definito

Per calcolare l'integrale definito dobbiamo passare alla funzione `integrate()`, oltre ai due argomenti necessari per il calcolo dell'integrale indefinito, altri due argomenti: il limite inferiore e il limite superiore, secondo la sintassi:

```
integrate(<espressione>, (<variabile>, <limite_inferiore>, <limite superiore>))
```

Così l'integrale definito tra 1 e 3 di $cos(x)$ si calcola con l'istruzione

```
integrate(cos(x), (x, 1, 3))
```

il cui risultato è $-sin(1) + sin(3)$ che, assoggettato a `evalf()`, è il numero -0.700350976748029.

5.3 Limiti

La funzione per il calcolo dei limiti è `limit()` che richiede tre argomenti:
. la funzione di cui calcolare il limite, inserita direttamente o richiamando la variabile che la contiene,
. la variabile in relazione alla quale calcolare il limite,
. il valore cui far tendere la variabile per calcolare il limite.

Così, il limite di cos(x) per x tendente a 0 si calcola con l'istruzione
`limit(cos(x), x, 0)` che ritorna 1.
Se abbiamo la variabile v che contiene l'espressione 1/x, possiamo calcolarne il limite per x tendente a 0 con l'istruzione
`limit(v, x, 0)` che ritorna infinito, evidenziato in SymPy con il simbolo oo (doppia o minuscola).
`limit(v, x, oo)` ritorna 0.

La funzione `limit()` è funzione membro dell'oggetto espressione e può essere richiamata come tale: in questo caso accetta i soli argomenti relativi alla variabile in riferimento alla quale calcolare il limite e al valore cui essa debba tendere.
Esempio:
il limite di $sin(x)$ per x tendente a 1 lo possiamo calcolare con l'istruzione
`sin(x).limit(x, 1)` con risultato $sin(1)$, che assoggettato a `evalf(4)`, è il numero 0.8414.
Allo stesso risultato perverremmo con l'istruzione `(sin(x).limit(x,1)).evalf(4)` che dimostra quale varietà di combinazioni di istruzioni ci offre SymPy.

Per valutare il limite da una parte sola possiamo passare anche, tra virgolette, l'argomento + (limite a destra) o - (limite a sinistra) dopo quello relativo al valore cui debba tendere la variabile.
Esempio:
`limit(1/x, x, 0, '-')` ritorna - oo ($-\infty$) che è il limite di 1/x per x tendente a zero a sinistra
`limit(1/x, x, 0, '+')` ritorna oo (∞) che è il limite di 1/x per x tendente a zero a destra.

6 Risolutori

Il più semplice risolutore che ci offre SymPy e che ritengo più che sufficiente per noi dilettanti è la funzione `solve()` e risolve le equazioni algebriche, comprese polinomiali e trascendenti.

Gli argomenti da passare alla funzione sono:
. l'equazione da risolvere implicitamente uguagliata a zero: ciò significa, per esempio, che se l'equazione è $x^2 - x = 2$ l'argomento da inserire è $x^2 - x - 2$,
. la variabile in relazione alla quale si intende trovare la soluzione.

Sicché l'equazione indicata si risolve con l'istruzione
`solve(x**2-x-2, x)` che ritorna la lista delle due soluzioni possibili [-1, 2]

L'equazione da risolvere può essere inserita direttamente, come abbiamo fatto qui, o richiamando la variabile ove è inserita.

Se, per esempio, abbiamo la variabile `v` che contiene l'equazione $x^2 - x - 3$, possiamo risolverla con
`solve(v, x)` e, in questo caso, la lista delle soluzioni possibili è $\left[\frac{1}{2} - \frac{\sqrt{13}}{2}, \frac{1}{2} + \frac{\sqrt{13}}{2}\right]$. Per approssimare numericamente i risultati dobbiamo applicare `evalf()` a ciascuno di essi o utilizzare per ciascuno di essi la funzione `N()` (purtroppo tutto ciò non è applicabile a liste ma solo a oggetti singoli).

Altro esempio la risoluzione dell'equazione $ax - 3x^2 = 0$ che possiamo risolvere rispetto a `x` o ad `a` ottenendo i seguenti risultati:
`solve(a*x-3*x**2, x)` ritorna le possibili soluzioni [0, $\frac{a}{3}$],
`solve(a*x-3*x**2, a)` ritorna la soluzione $3x$.

Se l'equazione non ammette soluzioni reali vengono indicate le soluzioni complesse, come avviene, per esempio, cercando le soluzioni di $x^2 + x + 2 = 0$ con
`solve(x**2+x+2, x)` che ci dà la lista delle soluzioni [-1/2 - sqrt(7)*I/2, -1/2 + sqrt(7)*I/2], cioè $-0.5 - 1.3228756555323i$ e $-0.5 + 1.3228756555323i$ come possiamo ricavare sottoponendo le due espressioni a `evalf()`.

A volte la lista delle soluzioni ne comprende di reali e complesse e spesso le radici sono espresse in termini molto grezzi e da semilavorato rispetto all'aspettativa di vederle espresse con un numero, con la complicazione che il tutto si trova in liste alle quali, come ho già detto, non sono direttamente applicabili il metodo `evalf()` e la funzione `N()`.

In alcuni casi, infine, pure in presenza di equazioni che hanno più di una soluzione, ne viene indicata solo una.
Per esempio l'equazione trascendente $e^x - 5x + 1$ ha due soluzioni: 0,544880... e 2,396138....
Se la passiamo alla funzione `solve()` ci viene indicata solo la prima delle due.

La funzione `solve()` di SymPy è comunque il meglio che ci sia nel mondo Python per risolvere equazioni di qualsiasi tipo: tornerò su questo argomento in un esercizio nel capitolo finale.

7 Matrici

Per costruire una matrice usiamo la funzione `Matrix()` il cui argomento è la lista dei vettori riga che la compongono.

`Matrix([[3, 4], [2, 5]])`
costruisce la matrice $\begin{vmatrix} 3 & 4 \\ 2 & 5 \end{vmatrix}$

`Matrix([[2, 5]])`
costruisce il vettore riga $\begin{vmatrix} 2 & 5 \end{vmatrix}$

`Matrix([2, 5])`
costruisce il vettore colonna $\begin{vmatrix} 2 \\ 5 \end{vmatrix}$

Ovviamente, essendo in SymPy, possiamo utilizzare anche simboli:

`Matrix([[a, 4], [2, b]])`
costruisce la matrice $\begin{vmatrix} a & 4 \\ 2 & b \end{vmatrix}$

Per conoscere le dimensioni di una matrice abbiamo a disposizione il metodo `shape`.

Se M è la matrice $\begin{vmatrix} a & 2 & b \\ 2 & b & 3 \end{vmatrix}$
`M.shape` fornisce il risultato (2, 3), essendo 2 il numero delle righe e 3 il numero delle colonne.

Con gli operatori `+`, `-`, `*`, `/` e `**` possiamo fare la somma, la differenza, la moltiplicazione, la divisione e l'elevamento a potenza di matrici.

Per calcolare la matrice inversa dobbiamo elevare la matrice quadrata ad esponente `-1` (se compiamo questa operazione su una matrice non quadrata generiamo un errore).

L'oggetto matrice quadrata ha la funzione membro `det()` per calcolarne il determinante (se richiamiamo questa funzione in presenza di una matrice non quadrata generiamo un errore).
Esempi:

Siano A = $\begin{vmatrix} x & 2 \\ 1 & 3 \end{vmatrix}$ e B = $\begin{vmatrix} 3 & 2 \\ 4 & 1 \end{vmatrix}$

`A + B` = $\begin{vmatrix} x+3 & 4 \\ 5 & 4 \end{vmatrix}$

`A * 3` = $\begin{vmatrix} 3x & 6 \\ 3 & 9 \end{vmatrix}$

`A * B` = $\begin{vmatrix} 3x+8 & 2x+2 \\ 15 & 5 \end{vmatrix}$

`A ** 2` = $\begin{vmatrix} x^2+2 & 2x+6 \\ x+3 & 11 \end{vmatrix}$

`A ** -1` = $\begin{vmatrix} \frac{-3}{2-3x} & \frac{2}{2-3x} \\ \frac{1}{2-3x} & \frac{-x}{2-3x} \end{vmatrix}$

`B ** -1` = $\begin{vmatrix} -\frac{1}{5} & \frac{2}{5} \\ \frac{4}{5} & -\frac{3}{5} \end{vmatrix}$ `(B ** -1).evalf()` = $\begin{vmatrix} -0.2 & 0.4 \\ 0.8 & -0.6 \end{vmatrix}$

`A.det()` = $3x-2$

`B.det()` = -5

Matrice aumentata

Comunemente, per risolvere un sistema di equazioni lineari, si moltiplica l'inversa della matrice dei coefficienti per il vettore dei termini noti e si ottiene il vettore dei valori delle incognite.

Dato il sistema di equazioni

$$\begin{cases} 3x+4y=25 \\ 2x+y=20 \end{cases}$$

possiamo creare la matrice dei coefficienti
`A = Matrix([[3,4],[2,1]])`
il vettore colonna dei termini noti
`b = Matrix([25,20])`
ed eseguire
`A**-1 * b`
ottenendo il vettore $\begin{vmatrix} 11 \\ -2 \end{vmatrix}$ della soluzione: x = 11 e y = -2

Se usiamo un foglio di calcolo o altri moduli Python come NumPy e SciPy è questo il modo con cui procediamo alla soluzione dei sistemi di equazioni lineari.

SymPy ci offre un modo originale, avvalendosi di una funzione che utilizza la matrice aumentata `M = [A|b]` del sistema di equazioni, una matrice che contiene sia la matrice dei coefficienti sia il vettore dei termini noti.
Nel caso del sistema sopra ipotizzato costruiamo questa matrice aumentata così
`A = Matrix([[3,4,25],[2,1,20]]),`
ottenendo A = $\begin{vmatrix} 3 & 4 & 25 \\ 2 & 1 & 20 \end{vmatrix}$

La funzione prevista per calcolare la soluzione utilizzando la matrice aumentata è
`solve_linear_system()`
i cui argomenti sono, separati da virgole, il nome della matrice aumentata e il simbolo delle incognite.
Nel nostro caso `solve_linear_system(A, x, y)`
che produce il risultato nel dizionario `{x: 11, y: -2}`

8 Integrazione con Python

In questo capitolo vedremo alcuni esempi di come il modulo SymPy si possa utilmente integrare in script Python.

8.1 Risoluzione di equazioni

Nel capitolo 6, parlando della funzione `solve()` di SymPy, ho detto che questo formidabile strumento, per chi sia interessato a vedere le soluzioni di una equazione espresse in bei numeri, ha il difetto di essere molto grezzo nel presentare le soluzioni che trova. Peraltro non bisogna dimenticare che SymPy è innanzi tutto uno strumento di calcolo simbolico e i risultati che ritornano le sue funzioni devono essere utilizzabili per ulteriori elaborazioni, meglio facendolo in maniera diretta senza inutilmente passare attraverso intermedie conversioni numeriche ottenute con approssimazioni. Ed è per questo che `solve()` non si preoccupa di ottenere sempre e comunque risultati numerici, anche quando ha a che fare con soli numeri.

Questi sono esempi di come `solve()` si esprime.
Per l'equazione $x^2 - 2x - 3 = 0$, `solve(x**2-2*x-3, x)`, dal momento che le radici sono numeri belli e buoni, ritorna la lista di soluzioni [-1, 3];
per l'equazione $3x - 2 = 0$, `solve(3*x-2, x)`, dal momento che la radice è una frazione che richiede un'approssimazione, ritorna la soluzione [2/3];
per l'equazione $3^{x-1} - 4 = 0$, `solve(3**(x-1)-4, x)` ritorna la soluzione [log(12)/log(3)];
per l'equazione $e^x + 5a + 1$, `solve(exp(x)+5*a +1, x)` ritorna la soluzione [log(-5*a - 1)];
per l'equazione $x + log(x)$, `solve(x+log(x), x)` ritorna la soluzione [LambertW(1)], la più ermetica di tutte.

Se siamo interessati a soluzioni espresse in numeri, anche per quelle strane equazioni che hanno radici non immediatamente esprimibili in numero salvo approssimazioni di calcolo, possiamo costruirci uno script come questo, che utilizza la funzione `solve()` di SymPy:

```
from sympy import *
def avvio():
    print('Inserisci la parte sinistra di una equazione uguagliata a zero')
    print("(usa la sintassi Python e nomina con x l'incognita)")
    global espressione
    espressione = input()
    risolvi()
def risolvi():
    x = symbols('x')
    e = sympify(espressione)
    l = solve(e, x)
    n = len(l)
    for i in range(n):
        s = N(l[i-1])
        print('soluzione', i+1, ':  ')
        pprint(s)
    print("Vuoi risolvere un'altra equazione?  (si/no)")
    r = input()
    if r == 'si' or r == 'SI' or r == 'Si':
        avvio()
    else:
        input('Premi INVIO per terminare')
avvio()
```

In rosso è evidenziato il codice che utilizza il modulo SymPy, in nero è evidenziato il normale codice Python.
Lo script, dopo aver chiesto all'utente di inserire l'equazione da risolvere, la risolve utilizzando la funzione `solve()` di SymPy e poi applica la funzione `N()` di SymPy a ciascuna delle soluzioni.In questo modo potremo vedere le soluzioni espresse in formato numerico, pur conservando l'eventuale presenza di simboli non numerici.
Per le equazioni viste prima avremo questi risultati:
per l'equazione $x^2 - 2x - 3 = 0$
soluzione 1 : 3.00000000000000
soluzione 2 : -1.00000000000000
per l'equazione $3x - 2 = 0$
soluzione 1 : 0.666666666666667
per l'equazione $3^{x-1} - 4 = 0$
soluzione 1 : 2.26185950714292
per l'equazione $e^x + 5a + 1 = 0$
soluzione 1 : log(-5*a - 1)
per l'equazione $x + log(x) = 0$
soluzione 1 : 0.567143290409784
e, per altre, a volontà:
per l'equazione $(1 - sin(x))^2 + cos(x) = 0$
soluzione 1 : 0.785398163397448 + 1.38432969165679*I
soluzione 2 : 1.57079632679490
soluzione 3 : 0.785398163397448 - 1.38432969165679*I
per l'equazione $(1 + x)^3 - 2 = 0$
soluzione 1 : -1.62996052494744 + 1.09112363597172*I
soluzione 2 : 0.259921049894873
soluzione 3 : -1.62996052494744 - 1.09112363597172*I
per l'equazione $x^7 - 3x^4 + 4x^2 - x = 0$
soluzione 1 : 1.09802749885925 + 0.346632626761893*I
soluzione 2 : 0

soluzione 3 : -1.07697753213803
soluzione 4 : 0.263663043733714
soluzione 5 : -0.691370254657094 - 1.47587345054391*I
soluzione 6 : -0.691370254657094 + 1.47587345054391*I
soluzione 7 : 1.09802749885925 - 0.346632626761893*I

per l'equazione $ax^2 + bx + c = 0$

soluzione 1: $\frac{-0{,}5(b+(-4ac+b^2)^{0{,}5})}{a}$

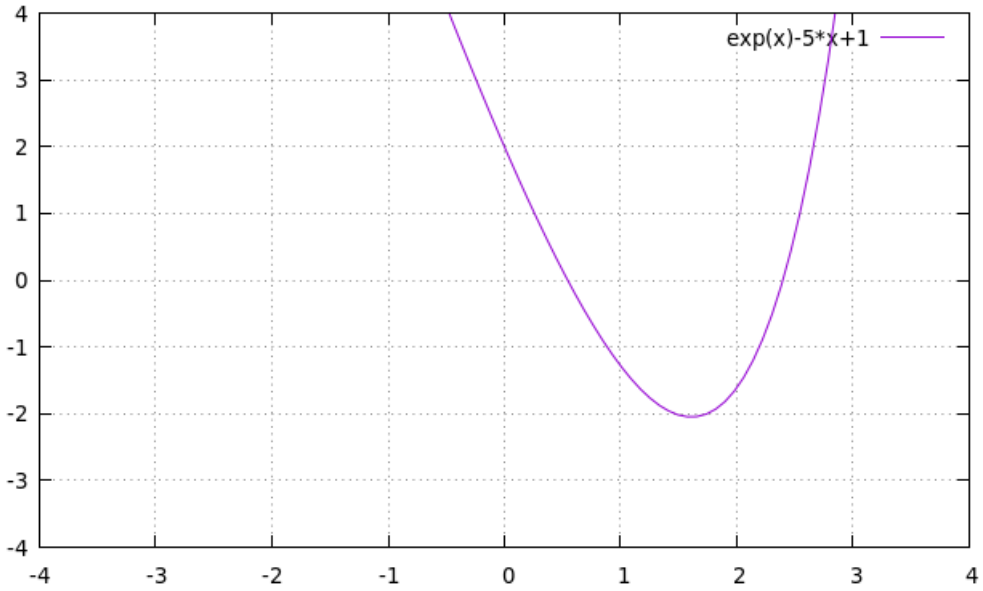
soluzione 2: $\frac{0{,}5(-b+(-4ac+b^2)^{0{,}5})}{a}$

che sono un altro modo di scrivere la nota formula risolutiva $\frac{-b\pm\sqrt{b^2-4ac}}{2a}$

per l'equazione $e^x - 5x + 1 = 0$

soluzione 1 : 0.544880440159982

Come si vede, se non ci sono di mezzo altri simboli, ora le soluzioni sono espresse come numeri. Purtroppo in molti casi abbiamo soluzioni reali mischiate con soluzioni complesse e questo disturba un tantino.
La cosa peggiore è che per l'ultima equazione ($e^x - 5x + 1 = 0$) ci viene indicata una sola radice, quando le radici sono due, come si può vedere dal seguente grafico gnuplot:



una tra 0 e 1, quella trovata da `solve()`, l'altra tra 2 e 3, che non è stata evidenziata.

* * *

Dal momento che SymPy è bravo a calcolare le derivate, possiamo costruire noi un risolutore, sempre integrando il nostro script con SymPy, che, invece di sfruttare la funzione `solve()`, sfrutta quest'altra sua capacità.

Esiste infatti un metodo per risolvere equazioni di qualsiasi tipo, detto metodo delle tangenti, ideato da Newton[2] , per applicare il quale occorre calcolare la derivata dell'espressione che rappresenta l'equazione.

Lo script il cui codice vediamo nella pagina seguente è un esempio di applicazione di questo metodo.

E' evidenziato in rosso il codice che utilizza il modulo SymPy e in nero è evidenziato il normale codice Python.
Dopo aver importato il modulo chiediamo all'utente di inserire l'equazione e la acquisiamo come stringa Python (espressione).

[2]Un'ottima esposizione di questo metodo, detto metodo di Newton-Raphson, si trova su Wikipedia

```
from sympy import *
def avvio():
    print('Inserisci la parte sinistra di una equazione uguagliata a zero')
    print("(usa la sintassi Python e nomina con x l'incognita)")
    global espressione
    espressione = input()
    global s
    s = 0.1
    risolvi()
def risolvi():
    ss = 0
    test = 0
    contatore = 0
    while 1:
        x = symbols('x')
        global s
        e = sympify(espressione)
        de = diff(e, x)
        ss = s - N(e.subs(x,s))/N(de.subs(x,s))
        test = s
        s = ss
        contatore = contatore + 1
        try:
            if int(ss*1000000) == int(test*1000000):
                break
        except TypeError:
            str(ss)
            ss = 'Non ho trovato alcuna soluzione reale'
            break
        if contatore == 100:
            str(ss)
            ss = 'Non ho trovato alcuna soluzione reale'
            break
    print('soluzione:  x = ', ss)
    print()
    print("Vuoi cercare altre soluzioni per la stessa equazione?  (si/no)")
    r1 = input()
    if r1 == 'SI' or r1 == 'Si' or r1 == 'si':
        ripeti()
   elif r1 != 'SI' or r1 != 'Si' or r1 != 'si':
        print("Vuoi cercare soluzioni per un'altra equazione?  (si/no)")
        r2 = input()
        if r2 == 'SI' or r2 == 'Si' or r2 == 'si':
            avvio()
        else:
            input('Premi INVIO per terminare')
def ripeti():
    print('Inserisci una radice tentativo')
    global s
    s = float(input())
    risolvi()
avvio()
```

Indi definiamo una variabile globale `s`, di fondamentale importanza per trovare le radici della nostra equazione; la definiamo come variabile globale in quanto dovremo utilizzarla in più di una funzione.
Questa variabile rappresenta un tentativo di soluzione e viene utilizzata per avviare un ciclo nel quale, a partire da questo tentativo, si genera un'altra variabile tentativo (`ss`) e si continua così fino a quando tra un tentativo e l'altro non vi è miglioramento del risultato: ciò significa che abbiamo trovato la soluzione.
Ho inizializzato la variabile con il valore 0,1 che è molto adatto per la soluzione di equazioni semplici e per trovare una prima radice di quelle più complicate.

Seguono poi due funzioni, la prima, `risolvi()`, dedicata a trovare una prima soluzione, la seconda, `ripeti()`, richiamabile a volontà, dedicata a trovare altre soluzioni.
Nella funzione `risolvi()`, innanzi tutto inizializziamo le variabili che ci servono per le iterazioni che prevede il metodo di Newton: `ss` è la soluzione «tentativo» che, in alternanza con la soluzione tentativo `s` predefinita come variabile globale, alimenterà le iterazioni, test ci serve per valutare quando siamo arrivati con buona approssimazione ad una soluzione vera e contatore serve per contare le iterazioni e poter fermare il programma una volta raggiunto un numero di iterazioni tale da escludere che si possano trovare soluzioni ed evitare il loop per ricercarle inutilmente.
Indi avviamo il ciclo: dichiariamo il simbolo x, che rappresenta l'incognita, dichiariamo di voler utilizzare la variabile globale `s`, trasformiamo in espressione SymPy (`e`) la stringa Python che rappresenta l'equazione (passaggio necessario per poter utilizzare in seguito il metodo `subs()`) e calcoliamo la derivata (`de`) della funzione sottesa nell'espressione.
A questo punto comincia il ciclo vero e proprio, durante il quale si inseriscono nelle espressioni che rappresentano la funzione e la sua derivata, il cui rapporto è il cuore della formula di Newton, una via l'altra, le soluzioni «tentativo» e ci si ferma quando si trova una soluzione (`ss`) che non migliora più rispetto al test (che contiene la soluzione precedente `s`).
Assumiamo così `ss` come soluzione e la stampiamo.
Notare come il rapporto tra funzione e derivata sia effettuato sulle espressioni numeriche approssimate di queste attraverso la funzione `N()`, ad evitare che, trattandosi di espressioni SymPy, la loro valorizzazione senza la forzatura della traduzione numerica conservi un simbolo numerico: il che manderebbe in loop il nostro programma in quanto non si arriverebbe mai ad una soluzione.
Ho infine inserito un paio di controlli per evitare loop quando non esistono soluzioni reali dell'equazione.

Alla fine del ciclo possiamo scegliere se ricercare altre soluzioni: nel qual caso ricorreremo alla funzione `ripeti()`, che semplicemente ci consente di modificare la variabile `s` che rappresenta la soluzione tentativo per avviare nuovamente la ricerca con la funzione `risolvi()` da un'altra posizione.

Con questo script possiamo calcolare le radici di una qualsiasi equazione, purché sia numerica e non contenga altri simboli oltre quello dell'incognita `x`.

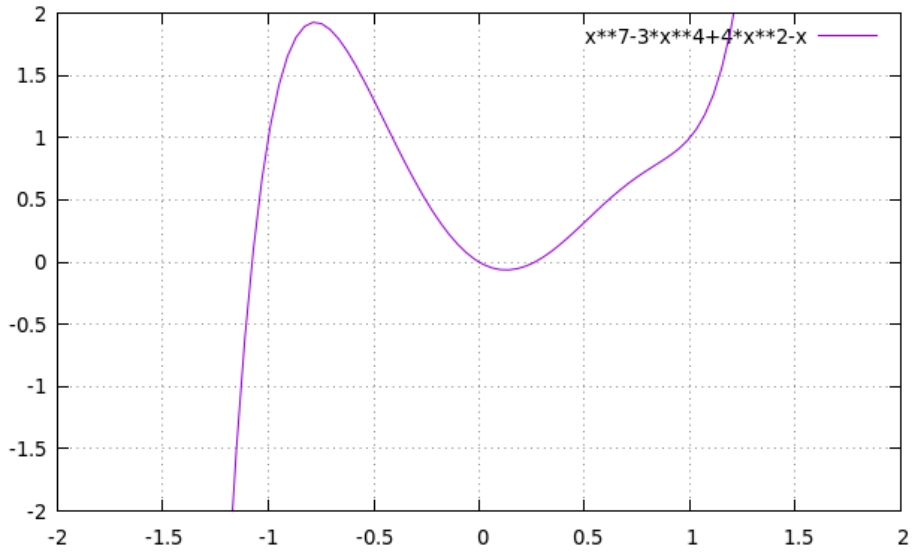
Rispetto allo script che abbiamo visto prima, oltre a questa limitazione (l'altro script risolve anche equazioni simboliche), questo script ci indica una sola soluzione, la prima che trova con l'iterazione a partire dalla soluzione tentativo, e per trovare le altre occorre ritentare con un'altra soluzione tentativo.

Se passiamo l'equazione $x^7 - 3x^4 + 4x^2 - x$ con la soluzione tentativo di default (variabile s uguale a 0,1) otteniamo la radice 0.
Se ritentiamo modificando la soluzione tentativo in 1, otteniamo la radice 0.263663043733714.
Se ritentiamo modificando la soluzione tentativo in -1, otteniamo la radice -1.07697753213803.

La soluzione tentativo, come si vede, deve essere il più vicino possibile alla soluzione vera.

Il problema è sapere prima quante e pressapoco dove si trovano le radici di un'equazione ricorrendo a un grafico: in ciò ci possiamo aiutare, per esempio, con gnuplot.

Nel caso dell'esempio appena visto, il grafico gnuplot



ci fa vedere che esistono tre radici reali, in quanto il grafico attraversa l'asse orizzontale 0 in tre punti, uno attorno all'ascissa -1, uno attorno all'ascissa zero e uno con ascissa tra 0 e 0,5.
Con il nostro script siamo arrivati a calcolare i valori numerici di queste radici.

Allo stesso modo possiamo calcolare quella radice che lo script precedente non evidenziava per l'equazione $e^x - 5x + 1 = 0$.
Se passiamo a questo nuovo script l'equazione con la soluzione tentativo di default 0,1 scoviamo la radice 0.544880440159981 che avevamo già trovato prima.
Se ritentiamo con la soluzione tentativo 3 scoviamo l'altra radice 2.39613850607607 che prima non era stata evidenziata.

8.2 Strumenti di analisi

Se vogliamo avere sottomano un agile strumento per calcolare derivate, integrali e limiti possiamo ricorrere a questo script:

```
from sympy import *
x = symbols('x')
def menu():
    print("Cosa ti interessa calcolare?")
    print(" digita 1 per la derivata")
    print(" digita 2 per l'integrale indefinito")
    print(" digita 3 per l'integrale definito")
    print(" digita 4 per il limite")
    global scelta
    scelta = int(input())
    if scelta == 0 or scelta > 4:
        print("Scelta sbagliata")
        menu()
    elif scelta == 1:
        derivata()
    elif scelta == 2:
        integrale_indefinito()
    elif scelta == 3:
        integrale_definito()
    elif scelta == 4:
        limite()
```

```
def derivata():
    print('Inserisci la funzione di x da derivare, con sintassi Python')
    espressione = input()
    f = sympify(espressione)
    d = diff(f, x)
    print('la derivata di:')
    pprint(f)
    print('è:')
    pprint(d)
    seguito()
def integrale_indefinito():
    print('Inserisci la funzione da integrare, con sintassi Python')
    espressione = input()
    f = sympify(espressione)
    i = integrate(f, x)
    print("l'integrale indefinito di:")
    pprint(f)
    print('è:')
    pprint(i)
    seguito()
def integrale_definito():
    print('Inserisci la funzione da integrare, con sintassi Python')
    espressione = input()
    print('Inserisci il limite inferiore (sinistro)')
    li = float(input())
    print('Inserisci il limite superiore (destro)')
    ls = float(input())
    f = sympify(espressione)
    i = integrate(f, (x, li, ls))
    print("l'integrale definito tra", li, 'e', ls, 'di:')
    pprint(f)
    print('è:')
    pprint(N(i))
    seguito()
def limite():
    print('Inserisci la funzione di cui trovare il limite, con sintassi Python')
    espressione = input()
    print('Inserisci a cosa deve tendere la variabile (oo per infinito)')
    t = input()
    f = sympify(espressione)
    l = limit(f, x, t)
    print('Il limite di:')
    pprint(f)
    print('per x tendente a', t, 'è:')
    pprint(N(l))
    seguito()
def seguito():
    print("")
    print("Vuoi fare altro?")
    print("Rispondi SI per proseguire")
    opzione = input()
    if opzione == "si" or opzione == "SI" or opzione == "Si":
        menu()
```

```
menu()
print("PREMI INVIO PER USCIRE DAL PROGRAMMA")
input()
```

In rosso abbiamo il codice per il modulo SymPy e in nero il codice Python generico.
E' un utile esercizio che si spiega da sé.

9 Avvertenza conclusiva

Giunto alla fine di questo manualetto illustrativo di SymPy sento il dovere di avvertire il lettore che SymPy non è solo quello che abbiamo visto qui.

Ciò che ho illustrato in questo manualetto è solo una parte delle moltissime cose che possiamo fare con SymPy.

La parte che ho selezionato ritengo sia quella che riguarda la matematica più comunemente praticata: trattamento delle espressioni, delle equazioni, qualche strumento di analisi.

Ed anche nel trattare la parte selezionata ho tralasciato alcuni aspetti e modalità che ho ritenuto complicanti per un semplice dilettante, sia pure evoluto. Non ho accennato, per esempio alle equazioni alle differenze finite ed alle equazioni differenziali, ecc.

Per chi voglia di più è disponibile la documentazione alla cui esistenza ho accennato fin dalla premessa: oltre ad approfondimenti su ciò che è stato visto in questo manualetto si potranno trovare strumenti per la matematica combinatoria, per la geometria, per il calcolo numerico e tanto altro.