Matematyczny Python

Obliczenia naukowe i analiza danych z użyciem NumPy, SciPy i Matplotlib

—

Robert Johansson

Apress

Tytuł oryginału: Numerical Python: Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib

Tłumaczenie: Filip Kamiński

ISBN: 978-83-283-7150-7

First published in English under the title Numerical Python: Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib by Robert Johansson, edition: 2

Copyright © 2019 by Robert Johansson

This edition has been translated and published under licence from APress Media, LLC, part of Springer Nature.

APress Media, LLC, part of Springer Nature takes no responsibility and shall not be made liable for the accuracy of the translation.

Polish edition copyright © 2021 by Helion SA

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/pytobl
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/pytobl.zip

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

O autorze 13

O korektorach merytorycznych 15

Wprowadzenie 19

Rozdział 1. Wprowadzenie do obliczeń w Pythonie 23

Środowiska obliczeniowe w Pythonie 26
Python 27
- Interpreter 27

Konsola IPython 28
- Buforowanie wejścia i wyjścia 29
- Autouzupełnianie i introspekcja obiektów 30
- Dokumentacja 30
- Interakcja z powłoką systemową 31
- Rozszerzenia IPythona 31

Jupyter 36
- Jupyter QtConsole 37
- Jupyter Notebook 37
- Jupyter Lab 39
- Rodzaje komórek 40
- Edycja komórek 41
- Komórki typu Markdown 42
- Możliwości prezentacji danych 42
- nbconvert 46

Zintegrowane środowisko programistyczne Spyder 48
- Edytor kodu źródłowego 50
- Konsola w Spyderze 50
- Inspektor obiektów 51

Podsumowanie 51
Materiały dodatkowe 52
Bibliografia 52

Kup ksi k

Pole ksi k

Rozdział 2. Wektory, macierze i tablice wielowymiarowe ... 53
Importowanie modułów ... 54
Typ tablicowy NumPy ... 54
Typy danych ... 55
Reprezentacja danych tablicowych w pamięci ... 57
Tworzenie tablic ... 58
Tablice utworzone na podstawie list i innych obiektów tablicopodobnych ... 60
Tablice wypełnione stałymi wartościami ... 60
Tablice wypełnione rosnącymi wartościami ... 61
Tablice z wartościami rozmieszczonymi logarytmicznie ... 62
Tablice z siatkami współrzędnych ... 62
Tworzenie niezainicjalizowanych tablic ... 63
Tworzenie tablic o cechach innych tablic ... 63
Tworzenie macierzy ... 63
Indeksowanie i zakresy ... 64
Tablice jednowymiarowe ... 64
Tablice wielowymiarowe ... 66
Widoki ... 67
Indeksowanie logiczne i fancy indexing ... 68
Zmiany kształtu i rozmiaru ... 69
Wyrażenia zwektoryzowane ... 74
Operacje arytmetyczne ... 74
Funkcje działające na elementach ... 77
Funkcje agregujące ... 79
Wyrażenia warunkowe i tablice wartości logicznych ... 81
Operacje na zbiorach ... 84
Operacje na tablicach ... 85
Operacje macierzowe i wektorowe ... 86
Podsumowanie ... 91
Materiały dodatkowe ... 92
Bibliografia ... 92

Rozdział 3. Obliczenia symboliczne ... 93
Importowanie modułów ... 94
Symbole ... 95
Liczby ... 97
Wyrażenia ... 102
Manipulowanie wyrażeniami ... 103
Upraszczanie wyrażeń ... 103
Rozwijanie wyrażeń ... 105
Funkcje factor, collect i combine ... 105
Funkcje Apart, Together i Cancel ... 106
Podstawienia ... 107
Ewaluacja wyrażeń ... 108

Kup ksi k Pole ksi k

Rachunek różniczkowy ... 109
Pochodne ... 109
Całki ... 111
Szeregi ... 112
Granice ... 114
Sumy i iloczyny uogólnione ... 115
Równania ... 115
Algebra liniowa ... 117
Podsumowanie ... 120
Materiały dodatkowe ... 121
Bibliografia ... 121

Rozdział 4. Wykresy i wizualizacje ... 123
Importowanie modułów ... 124
Pierwsze kroki ... 125
Tryb interaktywny i nieinteraktywny ... 128
Klasa Figure ... 130
Klasa Axes ... 131
Typy wykresów ... 132
Parametry linii ... 133
Legendy ... 136
Formatowanie tekstu i adnotacje ... 138
Właściwości osi ... 140
Złożone układy obiektów Axes ... 149
Wstawki ... 149
plt.subplots ... 150
subplot2grid ... 152
GridSpec ... 152
Wykresy typu colormap ... 153
Wykresy 3D ... 156
Podsumowanie ... 158
Materiały dodatkowe ... 158
Bibliografia ... 158

Rozdział 5. Rozwiązywanie równań ... 159
Importowanie modułów ... 160
Układy równań liniowych ... 160
Układy z macierzą kwadratową ... 161
Układy równań z macierzą prostokątną ... 166
Problem wartości własnych ... 169
Równania nieliniowe ... 171
Równania jednowymiarowe ... 171
Układy równań nieliniowych ... 177
Podsumowanie ... 181
Materiały dodatkowe ... 181
Bibliografia ... 181

Kup ksi k   Pole ksi k

Rozdział 6. Optymalizacja ... 183
Importowanie modułów ... 184
Klasyfikacja problemów optymalizacyjnych ... 184
Optymalizacja jednowymiarowa ... 187
Optymalizacja wielowymiarowa bez ograniczeń ... 190
Nieliniowy problem najmniejszych kwadratów ... 196
Optymalizacja z ograniczeniami ... 198
Programowanie liniowe ... 202
Podsumowanie ... 204
Materiały dodatkowe ... 205
Bibliografia ... 205

Rozdział 7. Interpolacja ... 207
Importowanie modułów ... 208
Interpolacja ... 208
Wielomiany ... 209
Interpolacja wielomianowa ... 212
Interpolacja funkcjami sklejanymi ... 216
Interpolacja funkcji wielu zmiennych ... 218
Podsumowanie ... 224
Materiały dodatkowe ... 224
Bibliografia ... 224

Rozdział 8. Całkowanie ... 225
Importowanie modułów ... 226
Metody całkowania numerycznego ... 226
Całkowanie numeryczne z użyciem SciPy ... 230
Całki z funkcji w postaci tablicowej ... 233
Całki wielokrotne ... 235
Całkowanie symboliczne i całkowanie z dowolną precyzją ... 239
Całki krzywoliniowe ... 241
Transformaty całkowe ... 241
Podsumowanie ... 244
Materiały dodatkowe ... 245
Bibliografia ... 245

Rozdział 9. Równanie różniczkowe zwyczajne ... 247
Importowanie modułów ... 248
Równania różniczkowe zwyczajne ... 248
Rozwiązania symboliczne ... 250
Pola kierunków ... 255
Rozwiązywanie równań z użyciem transformaty Laplace'a ... 258
Numeryczne metody rozwiązywania równań różniczkowych ... 261
Numeryczne rozwiązywanie równań różniczkowych z użyciem SymPy ... 264
Podsumowanie ... 275
Materiały dodatkowe ... 276
Bibliografia ... 276

Kup ksi k Pole ksi k

Rozdział 10. Macierze rzadkie i grafy ... 277
Importowanie modułów ... 278
Macierze rzadkie w SciPy ... 278
Funkcje do tworzenia macierzy rzadkich ... 283
Algebra liniowa macierzy rzadkich ... 285
Układy równań liniowych ... 285
Grafy i sieci ... 291
Podsumowanie ... 297
Materiały dodatkowe ... 297
Bibliografia ... 297

Rozdział 11. Równania różniczkowe cząstkowe ... 299
Importowanie modułów ... 300
Równania różniczkowe cząstkowe ... 301
Metoda różnic skończonych ... 302
Metoda elementów skończonych ... 307
Przegląd frameworków MES ... 310
Rozwiązywanie równań różniczkowych cząstkowych z użyciem FEniCS-a ... 311
Podsumowanie ... 330
Materiały dodatkowe ... 330
Bibliografia ... 331

Rozdział 12. Przetwarzanie i analiza danych ... 333
Importowanie modułów ... 334
Wprowadzenie do Pandas ... 334
Typ Series ... 335
Typ DataFrame ... 337
Szeregi czasowe ... 344
Biblioteka Seaborn ... 353
Podsumowanie ... 358
Materiały dodatkowe ... 358
Bibliografia ... 359

Rozdział 13. Statystyka ... 361
Importowanie modułów ... 362
Statystyka i prawdopodobieństwo ... 362
Liczby losowe ... 364
Zmienne losowe i rozkłady ... 367
Testowanie hipotez ... 374
Metody nieparametryczne ... 378
Podsumowanie ... 381
Materiały dodatkowe ... 381
Bibliografia ... 381

Kup ksi k

Pole ksi k

Rozdział 14. Modelowanie statystyczne ... 383
Importowanie modułów ... 384
Wprowadzenie do modelowania statystycznego ... 385
Definiowanie modeli statystycznych w Patsy ... 386
Regresja liniowa ... 393
Przykładowe zbiory danych ... 400
Regresja dyskretna ... 401
Regresja logistyczna ... 402
Model Poissona ... 406
Szeregi czasowe ... 409
Podsumowanie ... 413
Materiały dodatkowe ... 413
Bibliografia ... 413

Rozdział 15. Uczenie maszynowe ... 415
Importowanie modułów ... 416
Krótki przegląd uczenia maszynowego ... 417
Regresja ... 419
Klasyfikacja ... 428
Klasteryzacja ... 431
Podsumowanie ... 436
Materiały dodatkowe ... 436
Bibliografia ... 436

Rozdział 16. Statystyka bayesowska ... 437
Importowanie modułów ... 438
Wprowadzenie do statystyki bayesowskiej ... 439
Definiowanie modelu ... 441
Próbkowanie rozkładów a posteriori ... 445
Regresja liniowa ... 448
Podsumowanie ... 458
Materiały dodatkowe ... 459
Bibliografia ... 459

Rozdział 17. Przetwarzanie sygnałów ... 461
Importowanie modułów ... 462
Analiza spektralna ... 462
Transformata Fouriera ... 462
Okna czasowe ... 467
Spektrogramy ... 471
Filtrowanie sygnałów ... 474
Filtry konwolucyjne ... 474
Filtry o skończonej i nieskończonej odpowiedzi impulsowej ... 476
Podsumowanie ... 481
Materiały dodatkowe ... 481
Bibliografia ... 481

Kup książkę Poleć książkę

Rozdział 18. Wprowadzanie i wyprowadzanie danych 483
Importowanie modułów 484
Format CSV 485
HDF5 489
h5py 490
PyTables 500
HDFStore z Pandas 503
JSON 505
Serializacja 509
Podsumowanie 511
Materiały dodatkowe 511
Bibliografia 512

Rozdział 19. Optymalizacja kodu 513
Importowanie modułów 515
Numba 516
Cython 522
Podsumowanie 531
Materiały dodatkowe 532
Bibliografia 532

Dodatek Instalacja i konfiguracja środowiska 533
Miniconda i conda 534
Pełne środowisko 540
Podsumowanie 543
Materiały dodatkowe 543

Kup ksi k Pole ksi k

Kup ksi k

Pole ksi k

ROZDZIAŁ 3

Obliczenia symboliczne

Obliczenia symboliczne opierają się na zupełnie innym paradygmacie niż obliczenia oparte na tablicach liczb, przedstawione w poprzednim rozdziale. W oprogramowaniu komputerowym do przetwarzania symbolicznego, znanym również pod nazwą systemów algebry komputerowej (lub komputerowych systemów obliczeń symbolicznych — ang. *computer algebra systems*, CAS), reprezentacje obiektów matematycznych podlegają przekształceniom i manipulacjom analitycznym. Obliczenia symboliczne polegają w dużej mierze na wykorzystaniu komputerów do automatyzacji obliczeń analitycznych, które można by w zasadzie wykonać ręcznie za pomocą długopisu i kartek papieru. Dzięki zautomatyzowaniu rachunków i manipulowaniu wyrażeniami matematycznymi za pomocą oprogramowania możliwe jest wykonywanie bardziej złożonych obliczeń niż w przypadku ręcznych zapisów. Obliczenia symboliczne są doskonałym narzędziem do sprawdzania i debugowania obliczeń analitycznych wykonywanych ręcznie, ale też, co ważniejsze, umożliwiają przeprowadzanie analizy od strony symbolicznej, która w innym przypadku nie byłaby możliwa.

Obliczenia analityczne i symboliczne są kluczową częścią środowiska obliczeń naukowych i technicznych. Umożliwiają przesunięcie granicy tego, co można osiągnąć w sposób analityczny przed przejściem do obliczeń numerycznych, nawet w przypadku problemów dających się rozwiązać tylko numerycznie (jest to dość powszechne zjawisko, ponieważ dla wielu praktycznych problemów nie istnieją metody analityczne). Zastosowanie obliczeń symbolicznych może na przykład zmniejszyć złożoność lub rozmiar problemu numerycznego, który finalnie i tak będzie musiał zostać rozwiązany w sposób numeryczny. Innymi słowy, zamiast poszukiwania numerycznego rozwiązania problemu w jego oryginalnej formie można na początek uprościć go z wykorzystaniem metod analitycznych.

Kup ksi k Pole ksi k

W naukowym środowisku Python głównym modułem obliczeń symbolicznych jest SymPy (Symbolic Python). Moduł ten został napisany w całości w Pythonie i oferuje narzędzia do rozwiązywania szerokiej gamy problemów analitycznych i symbolicznych. W tym rozdziale szczegółowo przyjrzymy się temu, w jaki sposób można wykorzystać bibliotekę SymPy do prowadzenia obliczeń symbolicznych w Pythonie.

■ **SymPy** Celem biblioteki Symbolic Python (SymPy) jest zapewnienie w pełni funkcjonalnego systemu algebry komputerowej (CAS). W przeciwieństwie do wielu innych narzędzi tego typu SymPy to przede wszystkim biblioteka, a nie pełne środowisko obliczeniowe. Dzięki temu doskonale nadaje się do integracji z rozwiązaniami i obliczeniami wykorzystującymi również inne biblioteki Pythona. W chwili gdy piszę te słowa, najnowsza wersja SymPy to 1.6.1. Więcej informacji o tej bibliotece znajdziesz na *www.sympy.org* i *https://github.com/sympy/sympy/wiki/Faq*.

Importowanie modułów

Projekt SymPy zapewnia moduł Pythona o nazwie `sympy`. Podczas pracy z tą biblioteką często importuje się wszystkie symbole z modułu `sympy`, z użyciem instrukcji `from sympy import *`. W celu zachowania przejrzystości i uniknięcia konfliktów pomiędzy funkcjami i zmiennymi z przestrzeni nazw SymPy oraz z innych pakietów, takich jak NumPy i SciPy (patrz dalsze rozdziały), bibliotekę tę zaimportuję w całości jako `sympy`. W dalszej części tej książki zakładam, że biblioteka SymPy jest importowana w dokładnie ten sposób.

```
In [1]: import sympy
In [2]: sympy.init_printing()
```

Po zaimportowaniu biblioteki wywołana została również funkcja `sympy.init_printing`, której wywołanie powoduje, że system SymPy służący do wyświetlania danych na ekranie prezentuje wyrażenia matematyczne w elegancki sposób. Będziesz mógł się o tym przekonać w dalszej części tego rozdziału. W Jupyter Notebook wywołanie to spowoduje, że biblioteka JavaScript MathJax stanie się odpowiedzialna za renderowanie wyrażeń SymPy, które będą wyświetlane bezpośrednio w przeglądarce, w oknie, w którym otwarty jest notatnik.

Dla wygody i w celu zwiększenia czytelności kodów w tym rozdziale załóż także, że następujące, często używane symbole, są importowane z SymPy do lokalnej przestrzeni nazw:

```
In [3]: from sympy import I, pi, oo
```

■ **Uwaga** Biblioteki NumPy i SymPy, jak też wiele innych, zawierają liczne funkcje i zmienne o tej samej nazwie. Jednak symbole te rzadko można stosować wymiennie, przykładowo `numpy.pi` to numeryczne przybliżenie wartości , a `sympy.pi` to jej symboliczna reprezentacja. Ważne jest, aby nie mieszać ze sobą tych symboli, na przykład podczas wykonywania obliczeń symbolicznych w miejscu `sympy.pi` nie powinno się stosować `numpy.pi` (i odwrotnie). Ta sama reguła dotyczy wielu podstawowych funkcji matematycznych, takich jak `numpy.sin` i `sympy.sin`. Dlatego też przy korzystaniu z więcej niż jednego pakietu obliczeniowego w Pythonie ważne jest konsekwentne wykorzystywanie przestrzeni nazw.

Kup ksi k Pole ksi k

Symbole

Podstawową funkcjonalnością biblioteki SymPy jest możliwość reprezentowania symboli matematycznych w postaci obiektów Pythona. W bibliotece SymPy można do tego wykorzystać na przykład klasę `sympy.Symbol`. Instancja klasy `Symbol` zawiera nazwę i zestaw atrybutów opisujących jej właściwości wraz z metodami pozwalającymi uzyskać do nich dostęp i metodami pozwalającymi prowadzić na nich działania. Sam symbol nie ma większego zastosowania praktycznego, ale symbole są wykorzystywane w roli węzłów w drzewach reprezentujących wyrażenia algebraiczne (patrz następny podrozdział). Jednym z pierwszych kroków podczas analizowania problemu z wykorzystaniem SymPy jest stworzenie symboli dla różnych zmiennych matematycznych i wielkości wymaganych do jego opisania.

Nazwa symbolu jest łańcuchem znaków, który może zawierać dodatkowe znaczniki podobne do LaTeX-a, pozwalające na eleganckie zaprezentowanie treści, na przykład w bogatym systemie wyświetlania IPython. Nazwa obiektu typu `Symbol` jest ustalana podczas jego tworzenia. Symbole można tworzyć na kilka różnych sposobów, między innymi przy użyciu instrukcji `sympy.Symbol`, `sympy.symbols` i `sympy.var`. Zazwyczaj pożądane jest powiązanie symboli SymPy ze zmiennymi Python o tej samej nazwie (lub nazwie, która jest do nich bardzo zbliżona). Na przykład, aby utworzyć symbol o nazwie `x` i powiązać go ze zmienną Pythona o tej samej nazwie, możesz do konstruktora klasy `Symbol` przekazać łańcuch znaków zawierający nazwę symbolu (pierwszy argument):

```
In [4]: x = sympy.Symbol("x")
```

Zmienna `x` reprezentuje teraz abstrakcyjny symbol matematyczny x, o którym nie wiadomo zbyt wiele. W tym momencie `x` może reprezentować na przykład liczbę rzeczywistą, liczbę całkowitą, liczbę zespoloną, funkcję, a także wiele innych obiektów. Chociaż w wielu przypadkach przedstawienie symbolu matematycznego w postaci abstrakcyjnego i nieokreślonego obiektu klasy `Symbol` jest wystarczające, czasem konieczne jest dostarczenie bibliotece dodatkowych wskazówek na temat tego, jaki rodzaj symbolu reprezentuje dany obiekt. Informacje te mogą pomóc w bardziej wydajnym manipulowaniu symbolem lub w upraszczaniu wyrażeń analitycznych. Z wykorzystaniem opcjonalnych argumentów funkcji tworzących symbole (takich jak funkcja `Symbol`) można przekazać bibliotece dodatkowe założenia pozwalające ograniczyć właściwości danego symbolu. Tabela 3.1 zawiera wybór często stosowanych założeń, które można powiązać z instancją klasy `Symbol`. Na przykład, jeżeli dysponujesz zmienną matematyczną y, o której wiadomo, że jest liczbą rzeczywistą, to podczas tworzenia odpowiadającej jej instancji możesz ograniczyć jej zakres przez przekazanie do funkcji argumentu `real=True`. Do sprawdzenia, czy SymPy rzeczywiście rozpoznaje, że dany symbol jest liczbą rzeczywistą, można użyć atrybutu `is_real`:

```
In [5]: y = sympy.Symbol("y", real=True)
In [6]: y.is_real
Out[6]: True
```


Kup ksi k

Pole ksi k

Tabela 3.1. *Wybrane założenia i odpowiadające im argumenty przekazywane podczas tworzenia obiektów klasy Symbol. Pełna lista atrybutów jest dostępna w dokumentacji sympy.Symbol*

Nazwa argumentu	Atrybut	Opis
`real`, `imaginary`	`is_real`, `is_imaginary`	Symbol reprezentuje liczbę rzeczywistą/urojoną.
`positive`, `negative`	`is_positive`, `is_negative`	Symbol jest dodatni/ujemny.
`integer`	`is_integer`	Symbol jest liczbą całkowitą.
`odd`, `even`	`is_odd`, `is_even`	Symbol reprezentuje nieparzystą/parzystą liczbę całkowitą.
`prime`	`is_prime`	Symbol jest liczbą pierwszą, a zatem także liczbą całkowitą.
`finite`, `infinite`	`is_finite`, `is_infinite`	Symbol reprezentuje ilość, która jest skończona/nieskończona.

Z drugiej strony skorzystanie z metody `is_real` do sprawdzenia, czy zdefiniowany wcześniej, bez jawnego określenia typu wartości, a zatem mogący reprezentować zarówno liczby rzeczywiste, jak i urojone, symbol `x` może zawierać liczby rzeczywiste, spowoduje zwrócenie wartości `None`:

```
In [7]: x.is_real is None
Out[7]: True
```

Zauważ, że metoda `is_real` zwraca `True`, tylko jeżeli wiadomo, że symbol jest liczbą rzeczywistą, `False` — jeżeli wiadomo, że symbol jest liczbą urojoną, i `None` — gdy nie wiadomo, czy symbol jest rzeczywisty czy nie. Inne atrybuty (patrz tabela 3.1) pozwalające na sprawdzenie założeń dotyczących obiektów klasy `Symbol` działają w ten sam sposób. Poniżej jest pokazany przykład symbolu, dla którego atrybut `is_real` ma wartość `False`:

```
In [8]: sympy.Symbol("z", imaginary=True).is_real
Out[8]: False
```

Najważniejsze z założeń pokazanych w tabeli 3.1, które warto jawnie określić podczas tworzenia nowych symboli, to `real` i `positive`. W odpowiednich przypadkach dodanie tych założeń może pomóc bibliotece w uproszczeniu różnych wyrażeń w stopniu większym niż w razie ich braku. Rozważ następujący prosty przykład:

```
In [9]: x = sympy.Symbol("x")
In [10]: y = sympy.Symbol("y", positive=True)
In [11]: sympy.sqrt(x ** 2)
```
Out[11]: $\sqrt{x^2}$
```
In [12]: sympy.sqrt(y ** 2)
Out[12]: y
```

W powyższym przykładzie utworzono dwa symbole, `x` i `y`, oraz obliczono z nich pierwiastki kwadratowe za pomocą funkcji `sympy.sqrt`. W przypadku gdy brakuje informacji o wykorzystywanym w obliczeniach symbolu, wyrażenia nie da się uprościć. Z drugiej strony, gdy wiadomo, że symbol reprezentuje liczbę dodatnią, a tym samym wiadomo, że $\sqrt{y^2} = y$, SymPy poprawnie rozpoznaje tę sytuację i dokonuje uproszczenia.

Kup ksi k Pole ksi k

Podczas pracy z symbolami matematycznymi, które reprezentują liczby całkowite, warto, jeżeli to możliwe, zamiast określać je jako liczby rzeczywiste, opisać je jawnie na przykład za pomocą argumentów takich jak `integer=True`, `even=True` lub `odd=True`. Dzięki temu SymPy może analitycznie uprościć niektóre wyrażenia i obliczenia wartości funkcji, tak jak pokazałem w poniższym przykładzie:

```
In [13]: n1 = sympy.Symbol("n")
    ...: n2 = sympy.Symbol("n", integer=True)
    ...: n3 = sympy.Symbol("n", odd=True)
In [14]: sympy.cos(n1 * pi)
Out[14]: cos( n)
In [15]: sympy.cos(n2 * pi)
Out[15]: (–1)n
In [16]: sympy.cos(n3 * pi)
Out[16]: –1
```

By opisać nietrywialny problem matematyczny, często trzeba zdefiniować dużą liczbę symboli. Używanie konstruktora `Symbol` do tworzenia każdego z osobna może być trochę monotonne, dlatego też SymPy zawiera również wygodną funkcję `sympy.symbols`, umożliwiającą tworzenie wielu symboli w jednym wywołaniu. Funkcja ta pobiera rozdzielony przecinkami ciąg nazw symboli oraz zestaw argumentów (które odnoszą się do wszystkich tworzonych symboli) i zwraca krotkę nowo utworzonych symboli. Wykorzystanie składni rozpakowywania krotek w połączeniu z wywołaniem `sympy.symbols` to wygodny sposób na tworzenie nowych symboli:

```
In [17]: a, b, c = sympy.symbols("a, b, c", negative=True)
In [18]: d, e, f = sympy.symbols("d, e, f", positive=True)
```

Liczby

Celem reprezentowania symboli matematycznych w postaci obiektów Pythona jest ich użycie w drzewach wyrażeń, które reprezentują wyrażenia matematyczne. Aby z nich skorzystać poza zmiennymi matematycznymi w postaci obiektów, musisz przedstawić również pozostałe byty, takie jak liczby, funkcje i stałe. W tym punkcie przyjrzymy się klasom służącym do reprezentowania obiektów liczbowych. Wszystkie te klasy współdzielą wiele metod i atrybutów z obiektami klasy `Symbol`. Dzięki temu liczby i symbole mogą być traktowane w wyrażeniach w ten sam sposób.

W poprzedniej sekcji wspomniałem, że obiekty klasy `Symbol` zawierają atrybuty pozwalające na sprawdzenie ich właściwości, na przykład `is_real`. Podczas manipulowania wyrażeniami symbolicznymi musisz mieć możliwość stosowania tych samych atrybutów dla wszystkich typów obiektów, również dla liczb całkowitych i zmiennoprzecinkowych. Z tego powodu w wyrażeniach nie możesz korzystać z wbudowanych w Pythona typów, na przykład liczb całkowitych typu `int` czy liczb zmiennoprzecinkowych typu `float`. Zamiast tego w środowisku SymPy dostępne są odpowiadające im klasy, takie jak `sympy.Integer` i `sympy.Float`. Warto o tym pamiętać podczas pracy z SymPy, ale na szczęście obiektów typu `sympy.Integer` i `sympy.Float` nie trzeba tworzyć zbyt często, ponieważ SymPy automatycznie promuje wartości liczbowe Pythona do klas występujących w wyrażeniach symbolicznych. Jednakże aby zademonstrować różnicę pomiędzy wbudowanymi w Pythona typami liczbowymi a odpowiadającymi im typami z biblioteki SymPy, w poniższym przykładzie jawnie utworzono instancje `sympy.Integer` oraz `sympy.Float` i wykorzystano niektóre z ich atrybutów do ustalenia ich własności:

Kup ksi k Pole ksi k

```
In [19]: i = sympy.Integer(19)
In [20]: type(i)
Out[20]: sympy.core.numbers.Integer
In [21]: i.is_Integer, i.is_real, i.is_odd
Out[21]: (True, True, True)
In [22]: f = sympy.Float(2.3)
In [23]: type(f)
Out[23]: sympy.core.numbers.Float
In [24]: f.is_Integer, f.is_real, f.is_odd
Out[24]: (False, True, False)
```

■ **Wskazówka** Obiekty typów `sympy.Integer` i `sympy.Float` można rzutować z powrotem na wbudowane typy Pythona za pomocą standardowego rzutowania z użyciem `int(i)` i `float(f)`.

Do utworzenia reprezentacji liczby lub dowolnego wyrażenia w SymPy można również zastosować funkcję `sympy.sympify`. Funkcja ta przyjmuje szeroki zakres argumentów i zwraca wyrażenie kompatybilne ze składnią SymPy, bez potrzeby jawnego określania typu tworzonych obiektów. W przypadku danych liczbowych wystarczy skorzystać z poniższego wywołania:

```
In [25]: i, f = sympy.sympify(19), sympy.sympify(2.3)
In [26]: type(i), type(f)
Out[26]: (sympy.core.numbers.Integer, sympy.core.numbers.Float)
```

Liczby całkowite

W poprzednim punkcie do reprezentowania liczb całkowitych użyto klasy `Integer`. Warto zauważyć, że istnieje różnica pomiędzy instancją klasy `Symbol` utworzoną z argumentem `integer=True` a obiektem klasy `Integer`. Podczas gdy `Symbol` z argumentem `integer=True` reprezentuje pewną liczbę całkowitą, obiekt klasy `Integer` reprezentuje określoną wartość. W obu przypadkach atrybut `is_integer` ma wartość `True`, ale poza nim istnieje również atrybut `is_Integer` (zwróć uwagę na duże „I” w nazwie), który jest prawdziwy tylko dla obiektów typu `Integer`. Zasadniczo atrybuty postaci `is_Name` wskazują, czy dany obiekt jest obiektem typu `Name`, a atrybuty `is_name` — czy obiekt spełnia warunek `name`. Zatem istnieje także atrybut `is_Symbol`, który jest prawdziwy dla obiektów należących do klasy `Symbol`.

```
In [27]: n = sympy.Symbol("n", integer=True)
In [28]: n.is_integer, n.is_Integer, n.is_positive, n.is_Symbol
Out[28]: (True, False, None, True)
In [29]: i = sympy.Integer(19)
In [30]: i.is_integer, i.is_Integer, i.is_positive, i.is_Symbol
Out[30]: (True, True, True, False)
```

Liczby całkowite w SymPy charakteryzują się dowolną precyzją (są to tak zwane duże liczby całkowite). Oznacza to, że takie liczby nie mają ustalonych dolnych i górnych granic, tak jak liczby całkowite o określonym rozmiarze bitowym (na przykład te znane z biblioteki NumPy). Dzięki temu w SymPy możliwa jest praca z bardzo dużymi liczbami. Poniżej zamieszczono kilka przykładów:

Kup ksi k   Pole ksi k

```
In [31]: i ** 50
Out[31]: 8663234049605954426644038200675212212900743262211018069459689001
In [32]: sympy.factorial(100)
Out[32]: 933262154439441526816992388562667004907159682643816214685929638952 1759999322991
➥5608941463976156518286253697920827223758251185210916864000000000 000000000000000
```

Liczby zmiennoprzecinkowe

W poprzedniej części tego rozdziału spotkałeś się już z typem `sympy.Float`. W przeciwieństwie do wbudowanego w Pythona typu `float` i typów zmiennoprzecinkowych z NumPy, typ `Float` z biblioteki SymPy, podobnie jak typ `Integer`, cechuje dowolna precyzja. Oznacza to, że obiekt typu `Float` może reprezentować liczbę zmiennoprzecinkową z dowolną liczbą miejsc po przecinku. Konstruktor tego typu przyjmuje dwa argumenty: pierwszy z nich to wartość zmiennoprzecinkowa typu `float` (standardowy typ z Pythona) lub łańcuch znaków reprezentujący liczbę zmiennoprzecinkową, a drugi argument (opcjonalny) to precyzja, z jaką wartość ta ma być reprezentowana (liczba znaczących cyfr dziesiętnych). Powszechnie wiadomo, że na przykład wartość 0,3 nie może być dokładnie reprezentowana w postaci liczby zmiennoprzecinkowej o stałym rozmiarze bitowym. Podczas próby wyświetlenia tej wartości z dokładnością do 20 cyfr znaczących na ekranie zobaczysz wartość `0.299999999999999988888977698`. Obiekt typu `Float` z biblioteki SymPy może reprezentować tę liczbę bez ograniczeń związanych z klasycznymi liczbami zmiennoprzecinkowych:

```
In [33]: "%.25f" % 0.3 # Tworzenie łańcucha znaków reprezentującego wartość z dokładnością do 25 miejsc po przecinku
Out[33]: '0.2999999999999999888977698'
In [34]: sympy.Float(0.3, 25)
Out[34]: 0.2999999999999999888977698
In [35]: sympy.Float('0.3', 25)
Out[35]: 0.3
```

Zauważ, że aby poprawnie przedstawić wartość 0,3 w postaci obiektu `Float`, trzeba go zainicjalizować z użyciem łańcucha `"0.3"`, a nie wartości typu `float` z Pythona, która od początku swojego istnienia zawiera błąd związany z reprezentacją zmiennoprzecinkową.

Liczby wymierne

Liczba wymierna to ułamek $\frac{p}{q}$ składający się z dwóch liczb całkowitych, licznika p i mianownika q. SymPy reprezentuje ten typ liczb za pomocą klasy `sympy.Rational`. Liczby wymierne można tworzyć jawnie, z wykorzystaniem argumentów konstruktora `sympy.Rational` reprezentujących licznik i mianownik:

```
In [36]: sympy.Rational(11, 13)
```

Out[36]: $\frac{11}{13}$

Ułamki mogą być również wynikiem uproszczeń prowadzonych przez bibliotekę. W obu przypadkach operacje arytmetyczne obejmujące liczby wymierne i całkowite dają w wyniku liczby wymierne.

```
In [37]: r1 = sympy.Rational(2, 3)
In [38]: r2 = sympy.Rational(4, 5)
In [39]: r1 * r2
```

Out[39]: $\frac{8}{15}$

```
In [40]: r1 / r2
```

Out[40]: $\frac{5}{6}$

Stałe i symbole specjalne

SymPy zawiera predefiniowane symbole reprezentujące stałe matematyczne i obiekty specjalne, takie jak jednostka urojona (i) i nieskończoność. Symbole te przedstawiono w tabeli 3.2 wraz z odpowiadającymi im symbolami z SymPy. Zwróć uwagę, że jednostka urojona została oznaczona w SymPy dużą literą `I`.

Tabela 3.2. *Wybrane stałe matematyczne i symbole specjalne wraz z odpowiadającymi im symbolami z biblioteki SymPy*

Symbol matematyczny	Symbol w SymPy	Opis
π	`sympy.pi`	Stosunek obwodu koła do jego średnicy.
e	`sympy.E`	Podstawa logarytmu naturalnego; $e = \ln(1)$.
Y	`sympy.EulerGamma`	Stała Eulera.
i	`sympy.I`	Jednostka urojona.
∞	`sympy.oo`	Nieskończoność.

Funkcje

W SymPy obiekty reprezentujące funkcje można tworzyć za pomocą `sympy.Function`. Podobnie jak obiekt klasy `Symbol` obiekt typu `Function` przyjmuje jako pierwszy argument nazwę. SymPy rozróżnia funkcje zdefiniowane (*defined*) i niezdefiniowane (*undefined*), a także funkcje wywołane (*applied*) i niewywołane (*unapplied*). Utworzenie funkcji za pomocą konstruktora `Function` powoduje powstanie nazwanej niezdefiniowanej (abstrakcyjnej) i niewywołanej funkcji, której ewaluacja, ze względu na brak ciała lub opisującego ją wyrażenia, jest niemożliwa. Ponieważ funkcja nie została jeszcze wywołana wraz z konkretnymi symbolami lub zmiennymi wejściowymi, reprezentuje ona dowolną funkcję o dowolnej liczbie argumentów. Niewywołana funkcja może zostać wywołana z zestawem symboli wejściowych reprezentujących jej dziedzinę przez wywołanie instancji obiektu z symbolami przekazanymi do niego w postaci argumentów[1]. Rezultat wywołania nadal jest funkcją niezdefiniowaną, ale taką, która została wywołana z określonym zestawem argumentów, a zatem dysponującą zbiorem zmiennych zależnych. Do zilustrowania tej koncepcji posłuży mi następujący kod, w którym tworzę

[1] W tym miejscu należy pamiętać o rozróżnieniu pomiędzy funkcją Pythona lub obiektem wywoływalnym (*callable object*), takim jak `sympy.Function`, a funkcją symboliczną, reprezentowaną przez instancję klasy `sympy.Function`.

Kup ksi k   Pole ksi k

niezdefiniowaną funkcję `f` wywoływaną z argumentem (symbolem) `x` oraz inną funkcję, `g`, wywoływaną bezpośrednio ze zbiorem argumentów (symboli) `x`, `y`, `z`:

```
In [41]: x, y, z = sympy.symbols("x, y, z")
In [42]: f = sympy.Function("f")
In [43]: type(f)
Out[43]: sympy.core.function.UndefinedFunction
In [44]:f(x)
Out[44]: f(x)
In [45]: g = sympy.Function("g")(x, y, z)
In [46]: g
Out[46]: g(x,y,z)
In [47]: g.free_symbols
Out[47]: {x,y,z}
```

W powyższym przykładzie do pokazania, że zastosowana funkcja rzeczywiście jest powiązana z określonym zestawem symboli wejściowych, skorzystałem z właściwości `free_symbols`, która zwraca zestaw unikatowych symboli zawartych w danym wyrażeniu (w tym przypadku wywołanej, ale nie zdefiniowanej funkcji `g`). Związek ten będzie istotny w dalszej części tego rozdziału, na przykład podczas rozważań na temat pochodnych funkcji abstrakcyjnych. Jednym z ważnych zastosowań funkcji niezdefiniowanych są równania różniczkowe, w których znane jest wyrażenie uwzględniające funkcję, ale nie jest znana jej postać.

W przeciwieństwie do funkcji niezdefiniowanych funkcja zdefiniowana to taka, która ma określoną implementację i której wartość jest możliwa do wyliczenia dla wszystkich poprawnych argumentów. Funkcję tego typu można zdefiniować na przykład przez utworzenie klasy dziedziczącej po `sympy.Function`. W większości przypadków wystarczy jednak skorzystać z funkcji matematycznych zapewnianych przez SymPy. Biblioteka zawiera wiele standardowych funkcji matematycznych, które są dostępne w globalnej przestrzeni nazw SymPy (jeżeli chcesz zapoznać się z ich listą, skorzystaj z funkcji `help` i zajrzyj do dokumentacji modułów `sympy.functions.elementary`, `sympy.functions.combinatorial` i `sympy.functions.special` oraz powiązanych z nimi podmodułów). Na przykład funkcja SymPy odpowiadająca sinusowi jest dostępna jako `sympy.sin` (zgodnie z obowiązującą w tej książce konwencją importu). Zauważ, że nie jest to funkcja w pythonowym znaczeniu tego słowa. W rzeczywistości jest to klasa dziedzicząca po `sympy.Function`, reprezentująca funkcję `sinus`, którą można zastosować z argumentem będącym wartością liczbową, symbolem lub wyrażeniem.

```
In [48]: sympy.sin
Out[48]: sympy.functions.elementary.trigonometric.sin
In [49]: sympy.sin(x)
Out[49]: sin(x)
In [50]: sympy.sin(pi * 1.5)
Out[50]: –1
```

Wywołana z symbolem abstrakcyjnym, takim jak `x`, funkcja `sin` pozostaje niezewaluowana. Ewaluacja następuje, gdy ustalenie wartości liczbowej jest możliwe, na przykład po wywołaniu z argumentem liczbowym lub w specjalnych przypadkach, w których wyrażenie w argumencie ma jakieś specjalne własności, tak jak w poniższym przykładzie:

```
In [51]: n = sympy.Symbol("n", integer=True)
In [52]: sympy.sin(pi * n)
Out[52]: 0
```


Kup ksi k   Pole ksi k

Trzecim rodzajem funkcji dostępnych w SymPy są funkcje lambda (funkcje anonimowe). Funkcje te nie są związane z żadną nazwą, ale mają określone ciało, które można poddać ewaluacji. Funkcje lambda można tworzyć za pomocą `sympy.Lambda`:

```
In [53]: h = sympy.Lambda(x, x**2)
In [54]: h
Out[54]: (x ↦ x²)
In [55]: h(5)
Out[55]: 25
In [56]: h(1 + x)
Out[56]: (1 + x)²
```

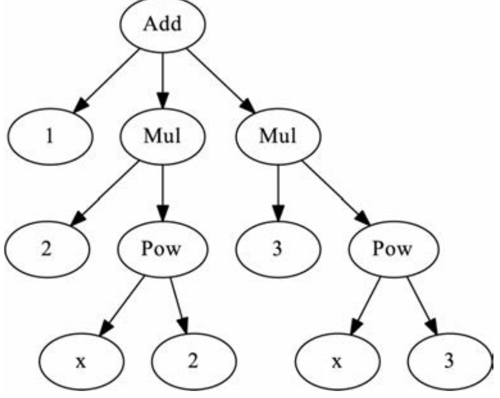
Wyrażenia

Różne symbole wprowadzone w poprzednim podrozdziale to podstawowe elementy budulcowe niezbędne do tworzenia wyrażeń matematycznych. W SymPy wyrażenia matematyczne są reprezentowane w postaci drzew, których liście stanowią symbole, a węzły — instancje klas reprezentujące operacje matematyczne. Przykładami takich klas są `Add`, `Mul` i `Pow`, odpowiadające podstawowym operacjom arytmetycznym, oraz `Sum`, `Product`, `Integral` i `Derivative`, reprezentujące działania analityczne. Ponadto istnieje wiele innych klas operacji matematycznych, które będziesz miał okazję zobaczyć w dalszej części tego rozdziału.

Dla przykładu weźmy wyrażenie $1 + 2x^2 + 3x^3$. Aby zapisać je w SymPy, wystarczy utworzyć symbol `x` i przedstawić formułę w postaci kodu Pythona:

```
In [57]: x = sympy.Symbol("x")
In [58]: expr = 1 + 2 * x**2 + 3 * x**3
In [59]: expr
Out[59]: 3x³ + 2x² + 1
```

W tym przykładzie zmienna `expr` jest instancją klasy `Add`, która składa się z trzech podwyrażeń: `1`, `2*x**2` i `3*x**3`. Na rysunku 3.1 jest pokazane całe drzewo reprezentujące wyrażenie `expr`. Zauważ, że nie jest konieczne jawne utworzenie drzewa, które jest budowane automatycznie na podstawie wyrażenia, symboli i operatorów. Niemniej jednak znajomość sposobu reprezentacji formuł matematycznych pomaga w zrozumieniu sposobu, w jaki działa biblioteka SymPy.



Rysunek 3.1. *Drzewo reprezentujące formułę 1 + 2 * x**2 + 3 * x**3*

Kup ksi k   Pole ksi k

Dzięki atrybutom args, zapewnianym przez wszystkie operacje i symbole SymPy, drzewo wyrażenia można przejść w sposób jawny. W przypadku operatora atrybut args jest krotką podwyrażeń połączonych ze sobą regułą implementowaną przez klasę operatora. W przypadku symboli atrybut args jest pustą krotką oznaczającą, że dany symbol jest liściem. Poniższy przykład pokazuje, w jaki sposób można uzyskać bezpośredni dostęp do drzewa wyrażenia:

```
In [60]: expr.args
Out[60]: (1,2x²,3x³)
In [61]: expr.args[1]
Out[61]: 2x²
In [62]: expr.args[1].args[1]
Out[62]: x²
In [63]: expr.args[1].args[1].args[0]
Out[63]: x
In [64]: expr.args[1].args[1].args[0].args
Out[64]: ()
```

W podstawowych zastosowaniach rzadko trzeba przeprowadzać operację na drzewie w jawny sposób, ale gdy metody manipulowania wyrażeniami przedstawione poniżej są niewystarczające, możliwość implementacji własnych funkcji, które przechodzą przez drzewo i wykonują na nim operacje z wykorzystaniem atrybutu args, może być przydatna.

Manipulowanie wyrażeniami

Operacje na drzewach wyrażeń stanowią jedno z głównych zadań biblioteki SymPy, która oferuje wiele funkcji pozwalających na realizację wielu typów przekształceń. Ogólna idea zakłada, że drzewa wyrażeń można przekształcać za pomocą funkcji uproszczeń i przekształceń w sposób zachowujący ich matematyczną równoważność. Zasadniczo funkcje te nie modyfikują wyrażeń przekazywanych jako argumenty, ale raczej tworzą nowe wyrażenie zawierające wprowadzone modyfikacje. Wyrażenia w SymPy należy zatem traktować jako obiekty niezmienne (ang. *immutable* — obiekty, których wartości nie można zmienić). Wszystkie opisane w tym podrozdziale funkcje traktują wyrażenia właśnie w ten sposób i, zamiast modyfikować drzewa w miejscu, zwracają ich nowe instancje.

Upraszczanie wyrażeń

W trakcie manipulowania formułami matematycznymi najbardziej pożądana jest możliwość uproszczenia wyrażenia. Ponieważ ustalenie w sposób algorytmiczny, czy jedno wyrażenie wydaje się człowiekowi prostsze niż drugie, oraz tego, którą metodę należy zastosować, aby je otrzymać, nie jest łatwe, operacja ta może być również najbardziej niejednoznaczna. Niemniej jednak „czarna skrzynka” upraszczająca wyrażenia jest ważną częścią każdego oprogramowania CAS. SymPy zawiera funkcję sympy.simplify, która próbuje uprościć wyrażenie przy użyciu różnych metod i podejść. Funkcję tę można również wywołać za pomocą metody simplify, w sposób pokazany poniżej:

```
In [65]: expr = 2 * (x**2 - x) - x * (x + 1)
In [66]: expr
Out[66]: 2x² − x(x+1)−2x
```


Kup ksi k   Pole ksi k

```
In [67]: sympy.simplify(expr)
Out[67]: x(x−3)
In [68]: expr.simplify()
Out[68]: x(x−3)
In [69]: expr
Out[69]: 2x^2 − x(x+1)−2x
```

Zauważ, że zarówno `sympy.simplify(expr)`, jak i `expr.simplify()` zwracają nowe drzewa wyrażeń i, jak wspomniano wcześniej, pozostawiają wyrażenie `expr` w niezmienionej formie. W tym przykładzie wyrażenie `expr` można uprościć przez pomnożenie ze sobą składników środkowego wyrazu, dodanie do siebie wyrazów podobnych, a następnie ponowną faktoryzację wyrażenia. Ogólnie funkcja `sympy.simplify` podejmie kilka różnych prób uproszczenia z zastosowaniem różnych strategii. Jak pokazałem poniżej, funkcja ta potrafi także upraszczać wyrażenia zawierające na przykład funkcje trygonometryczne i wyrażenia potęgowe:

```
In [70]: expr = 2 * sympy.cos(x) * sympy.sin(x)
In [71]: expr
Out[71]: 2 sin(x)cos(x)
In [72]: sympy.simplify(expr)
Out[72]: sin(2x)
```

i

```
In [73]: expr = sympy.exp(x) * sympy.exp(y)
In [74]: expr
Out[74]: exp(x)exp(y)
In [75]: sympy.simplify(expr)
Out[75]: exp(x+y)
```

Każdy konkretny rodzaj uproszczenia można również przeprowadzić za pomocą bardziej specjalistycznych funkcji, takich jak `sympy.trigsimp` i `sympy.powsimp`, które upraszczają, odpowiednio, wyrażenia trygonometryczne i potęgowe. Funkcje te wykonują jedynie uproszczenie, na które wskazują ich nazwy, pozostawiając inne części wyrażeń w oryginalnej formie. Przegląd funkcji upraszczających znajdziesz w tabeli 3.3. W większości przypadków, w których kolejne kroki niezbędne do uproszczenia wyrażenia są znane, warto korzystać z bardziej szczegółowych funkcji upraszczających. Działanie takich funkcji jest lepiej zdefiniowane, a prawdopodobieństwo wystąpienia zmian ich działania w przyszłych wersjach biblioteki SymPy jest mniejsze. Działanie funkcji `sympy.simplify` opiera się na podejściach heurystycznych, które mogą się zmienić w przyszłości, a w konsekwencji dawać inne wyniki dla określonych wyrażeń wejściowych.

Tabela 3.3. *Wybrane funkcje służące do upraszczania wyrażeń w SymPy*

Funkcja	Opis
`sympy.simplify`	Próba uproszczenia wyrażenia z wykorzystaniem różnych metod i podejść.
`sympy.trigsimp`	Próba uproszczenia wyrażenia z wykorzystaniem tożsamości trygonometrycznych.
`sympy.powsimp`	Próba uproszczenia wyrażenia z wykorzystaniem zasad działania na potęgach.
`sympy.compsimp`	Upraszczanie wyrażeń kombinatorycznych.
`sympy.ratsimp`	Upraszczanie wyrażenia przez sprowadzenie do wspólnego mianownika.

Kup ksi k   Pole ksi k

Rozwijanie wyrażeń

Gdy uproszczenie otrzymane z wywołania `sympy.simplify` nie daje satysfakcjonujących rezultatów, poprawę może przynieść ręczne wskazanie bibliotece właściwego kierunku przy użyciu bardziej szczegółowych operacji algebraicznych. Ważnym narzędziem w tym procesie są różne metody rozwijania wyrażeń. Funkcja `sympy.expand`, w zależności od przekazanych jej opcjonalnych argumentów, rozszerza wyrażenia na różne sposoby. Domyślnie funkcja ta wymnaża iloczyn i zwraca w pełni rozwiniętą sumę. Na przykład iloczyn typu $(x + 1)(x + 2)$ może zostać przekształcony do postaci $x^2 + 3x + 2$ za pomocą poniższego wywołania:

```
In [76]: expr = (x + 1) * (x + 2)
In [77]: sympy.expand(expr)
Out[77]: x^2 + 3x + 2
```

Niektóre z dostępnych argumentów to `mul=True`, powodujący wymnażanie iloczynów (jak w poprzednim przykładzie), `trig=True`, rozwijający wyrażenia trygonometryczne:

```
In [78]: sympy.sin(x + y).expand(trig=True)
Out[78]: sin(x)cos(y) + sin(y)cos(x)
```

`log=True`, rozwijający logarytmy:

```
In [79]: a, b = sympy.symbols("a, b", positive=True)
In [80]: sympy.log(a * b).expand(log=True)
Out[80]: log(a) + log(b)
```

`complex=True`, rozdzielający część rzeczywistą od urojonej:

```
In [81]: sympy.exp(I*a + b).expand(complex=True)
Out[81]: ieb sin(a) + eb cos(a)
```

i `power_base=True` oraz `power_exp=True` do rozwijania podstaw i wykładników potęg w wyrażeniach potęgowych:

```
In [82]: sympy.expand((a * b)**x, power_base=True)
Out[82]: a^x b^x
In [83]: sympy.exp((a-b)*x).expand(power_exp=True)
Out[83]: e^iax e^-ibx
```

Wywołanie funkcji `sympy.expand` z kolejnymi argumentami ustawionymi na `True` odpowiada wywołaniu bardziej szczegółowych funkcji, odpowiednio: `sympy.expand_mul`, `sympy.expand_trig`, `sympy.expand_log`, `sympy.expand_complex`, `sympy.expand_power_base` i `sympy.expand_power_exp`. Zaletą `sympy.expand` jest możliwość skorzystania z kilku bardziej szczegółowych funkcji w jednym wywołaniu.

Funkcje factor, collect i combine

W pracy z SymPy często najpierw wykorzystuje się funkcję `sympy.expand` do rozwinięcia wyrażenia, następnie pozwala się bibliotece zredukować wspólne czynniki, a na koniec ponownie rozkłada się lub łączy elementy otrzymanego wyrażenia. Funkcja `sympy.factor` próbuje rozłożyć wyrażenie na najprostsze możliwe czynniki. Jej działanie jest w pewnym sensie przeciwieństwem wywołania `sympy.expand` z argumentem `mul=True`. Funkcję tę można zastosować do faktoryzacji wyrażeń algebraicznych:

Kup ksi k Pole ksi k

```
In [84]: sympy.factor(x**2 - 1)
Out[84]: (x − 1)(x + 1)
In [85]: sympy.factor(x * sympy.cos(y) + sympy.sin(z) * x)
Out[85]: x(sin(x) + cos(y))
```

Odwrotności działania innych pokazanych w poprzednim punkcie funkcji można otrzymać z użyciem `sympy.trigsimp`, `sympy.powsimp` i `sympy.logcombine`. Na przykład:

```
In [86]: sympy.logcombine(sympy.log(a) - sympy.log(b))
```

Out[86]: $\log\left(\frac{a}{b}\right)$

W pracy z formułami matematycznymi często konieczna jest precyzyjna kontrola procesu faktoryzacji. Funkcja `sympy.collect` wyłącza przed nawias czynniki przekazane jej jako argumenty (w postaci pojedynczej wartości lub listy). Na przykład wyrażenie $x + y + xyz$ nie może być w całości rozłożone na czynniki, ale można przeprowadzić jego częściową faktoryzację względem zmiennej x lub y:

```
In [87]: expr = x + y + x * y * z
In [88]: expr.collect(x)
Out[88]: x(yz + 1) + y
In [89]: expr.collect(y)
Out[89]: x + y(xz + 1)
```

Przekazanie listy symboli lub wyrażeń do funkcji `sympy.collect` lub powiązanej z nią metody `collect` pozwala wyłączyć wiele symboli w pojedynczym wywołaniu. Ponadto podczas korzystania z metody `collect` zwracającej nowe wyrażenie możliwe jest łańcuchowe łączenie wywołań w następujący sposób:

```
In [90]: expr = sympy.cos(x + y) + sympy.sin(x - y)
In [91]: expr.expand(trig=True).collect([sympy.cos(x),
    ...:                                 sympy.sin(x)]).collect(sympy.cos(y) -
    ...:                                 sympy.sin(y))
Out[91]: (sin(x) + cos(x))(−sin(y) + cos(y))
```

Funkcje Apart, Together i Cancel

Ostatnim rozważanym w tym podrozdziale rodzajem uproszczeń będą przekształcenia ułamków. Funkcja `sympy.apart` rozkłada ułamek na ułamki proste, a funkcja `sympy.together` łączy ułamki proste w jeden ułamek złożony. Z funkcji tych można skorzystać w następujący sposób:

```
In [92]: sympy.apart(1/(x**2 + 3*x + 2), x)
```

Out[92]: $-\frac{1}{x+2}+\frac{1}{x+1}$

```
In [93]: sympy.together(1 / (y * x + y) + 1 / (1+x))
```

Out[93]: $\frac{y+1}{y(x+1)}$

```
In [94]: sympy.cancel(y / (y * x + y))
```

Out[94]: $\frac{1}{x+1}$

Kup ksi k Pole ksi k

W pierwszym przykładzie użyłem `sympy.apart` do przedstawienia wyrażenia $(x^2 + 3x + 2)^{-1}$ w postaci sumy ułamków prostych $-\frac{1}{x+2}+\frac{1}{x+1}$, w drugim zastosowałem funkcję `sympy.together` do sprowadzenia sumy $\frac{1}{yx+y}+\frac{1}{x+1}$ do wspólnego mianownika. W tym przykładzie skorzystałem również z funkcji `sympy.cancel` do wyeliminowania wspólnych czynników występujących w liczniku i mianowniku ułamka $\frac{y}{yx+y}$.

Podstawienia

Poprzednie punkty dotyczyły przekształcania wyrażeń przy użyciu różnych tożsamości matematycznych. Inną często używaną formą manipulowania nimi jest podstawianie symboli lub wyrażenia w obrębie badanej formuły. Pracując z danym wyrażeniem, możesz na przykład chcieć podstawić zmienną x pod zmienną y lub zastąpić symbol jakimś innym wyrażeniem. W SymPy istnieją dwie metody podstawień: `subs` i `replace`. Zazwyczaj najlepiej skorzystać z `subs`, ale w niektórych przypadkach warto użyć potężnego `replace`, które potrafi na przykład wykonać podstawienie tylko w miejscach określonych wyrażeniem regularnym (szczegółowe informacje można znaleźć w dokumentacji funkcji `sympy.Symbol.replace`).

W najprostszym przypadku metoda `subs` jest wywoływana na wyrażeniu. Jej pierwszym argumentem jest symbol/wyrażenie, które ma zostać zastąpione (`x`), a drugim zastępujący je symbol/wyrażenie (`y`). W rezultacie wszystkie wystąpienia wyrażenia `x` są zastępowane przez `y`:

```
In [95]: (x + y).subs(x, y)
Out[95]: 2y
In [96]: sympy.sin(x * sympy.exp(x)).subs(x, y)
Out[96]: sin(yeʸ)
```

Zamiast wywoływać metodę `subs` kilka razy, można jej, w sytuacjach, w których wymagane jest wykonanie wielu podstawień, przekazać jako jedyny argument słownik zawierający pary symboli i wartości, jakimi należy je zastąpić:

```
In [97]: sympy.sin(x * z).subs({z: sympy.exp(y), x: y, sympy.sin: sympy.cos})
Out[97]: cos(yeʸ)
```

Typowym zastosowaniem metody `subs` jest zastępowanie symboli wartościami liczbowymi w celu obliczenia wartości wyrażenia (więcej szczegółów znajduje się w następnym podrozdziale). Wygodnym sposobem przeprowadzania ewaluacji jest zdefiniowanie słownika tłumaczącego symbole na wartości liczbowe i przekazanie go do metody `subs`. Na przykład:

```
In [98]: expr = x * y + z**2 *x
In [99]: values = {x: 1.25, y: 0.4, z: 3.2}
In [100]: expr.subs(values)
Out[100]: 13.3
```


Kup ksi k	Pole ksi k

Ewaluacja wyrażeń

W trakcie każdych, nawet tych symbolicznych, obliczeń właściwie zawsze, wcześniej lub później, pojawia się konieczność ustalenia wartości liczbowej jakiegoś wyrażenia, na przykład przy tworzeniu wykresów lub opracowywania wyników. Wyrażenie SymPy można ewaulować za pomocą funkcji `sympy.N` lub metody `evalf` dostępnej w obiektach reprezentujących wyrażenie:

```
In [101]: sympy.N(1 + pi)
Out[101]: 4.14159265358979
In [102]: sympy.N(pi, 50)
Out[102]: 3.1415926535897932384626433832795028841971693993751
In [103]: (x + 1/pi).evalf(10)
Out[103]: x + 0.3183098862
```

Zarówno `sympy.N`, jak i metoda `evalf` przyjmują opcjonalny argument — liczbę cyfr znaczących — określający dokładność, z jaką ma nastąpić ewaluacja. Przykład użycia tego argumentu jest pokazany na powyższym listingu, w którym wykorzystano możliwości SymPy pozwalające na użycie reprezentacji z dowolną precyzją i dokonano ewaluacji wartości π z dokładnością do 50 cyfry znaczących.

Jeżeli chcesz ustalić wartość liczbową wyrażenia dla wartości z jakiegoś zakresu, możesz umieścić kolejne wywołania metody `evalf` w pętli, na przykład w poniższy sposób:

```
In [104]: expr = sympy.sin(pi * x * sympy.exp(x))
In [105]: [expr.subs(x, xx).evalf(3) for xx in range(0, 10)]
Out[105]: [0,0.774,0.642,0.722,0.944,0.205,0.974,0.977,-0.870,-0.695]
```

Rozwiązanie to jest raczej powolne i lepiej jest wykorzystać do tego celu dostępną w SymPy funkcję `sympy.lambdify`. Funkcja ta zwraca funkcję pozwalającą na wydajną ewaluację wyrażenia, które wraz z zestawem wolnych symboli ta pierwsza przyjmuje jako argumenty. Zwrócona funkcja przyjmuje tyle argumentów, ile zostało przekazanych wolnych symboli (w pierwszym argumencie) do funkcji `sympy.lambdify`.

```
In [106]: expr_func = sympy.lambdify(x, expr)
In [107]: expr_func(1.0)
Out[107]: 0.773942685266709
```

Zauważ, że funkcja `expr_func` oczekuje na wejściu tylko wartości liczbowych (skalarnych) i nie można do niej przekazać na przykład symbolu. Jej działanie jest ograniczone jedynie do obliczenia wartości powiązanego wyrażenia. Funkcja `expr_func` utworzona powyżej jest funkcją skalarną, która nie pozwala na użycie tablic NumPy jako jej argumentu w sposób omówiony w rozdziale 2. Biblioteka SymPy umożliwia również generowanie funkcji współpracujących z tablicami NumPy. Funkcje takie można otrzymać w wyniku przekazania funkcji `sympy.lambdify` jako trzeciego, opcjonalnego, argumentu `'numpy'`. SymPy utworzy wtedy zwektoryzowaną funkcję, która będzie przyjmowała na wejściu tablicę. Ogólnie rzecz biorąc, jest to wydajny sposób ewaluacji wyrażeń symbolicznych powiązanych z dużą liczbą parametrów wejściowych[2].

---

[2] Zobacz też funkcje `ufuncify` z modułu `sympy.utilities.autowrap` i `theano_function` z modułu `sympy.printing.theanocode`. Zapewniają one te same funkcjonalności, co `sympy.lambdify`, ale wykorzystują do tego inne backendy obliczeniowe.

Kup ksi k Pole ksi k

Poniższy kod ilustruje, w jaki sposób wyrażenie `expr` jest konwertowane na zwektoryzowaną funkcję przyjmującą tablicę NumPy, która umożliwia wydajne przeprowadzenie jego ewaluacji:

```
In [108]: expr_func = sympy.lambdify(x, expr, 'numpy')
In [109]: import numpy as np
In [110]: xvalues = np.arange(0, 10)
In [111]: expr_func(xvalues)
Out[111]: array([0.        , 0.77394269, 0.64198244,  0.72163867,  0.94361635,
                 0.20523391, 0.97398794, 0.97734066, -0.87034418, -0.69512687])
```

Ta metoda generowania danych z wyrażeń SymPy przydaje się podczas tworzenia wykresów i w innych zorientowanych na dane zastosowaniach.

Rachunek różniczkowy

Dotychczas pokazałem sposoby reprezentacji wyrażeń matematycznych w SymPy i podstawowe sposoby ich przekształcania i upraszczania. Wyposażeni w tę wiedzę, jesteśmy gotowi na spojrzenie na symboliczny rachunek różniczkowy, lub inaczej: na analizę matematyczną, która stanowi podstawowe narzędzie matematyki stosowanej i ma wiele zastosowań w nauce i inżynierii. Główne koncepcje rachunku różniczkowego stanowią: zmiana wartości funkcji związana ze zmianą wartości jej parametrów (opisywana za pomocą pochodnych i różniczek) oraz kumulacje wartości w danych zakresach opisywane przez całki. W tym podrozdziale dowiesz się, jak obliczać pochodne i całki w SymPy.

Pochodne

Pochodna funkcji opisuje szybkość zmiany jej wartości w danym punkcie. W SymPy możemy ją obliczyć za pomocą `sympy.diff` lub metody `diff` dostępnej w obiekcie reprezentującym dane wyrażenie. Argumentem obu tych funkcji jest symbol lub pewna liczba symboli, względem których należy ustalić wartość pochodnej. Pochodną pierwszego rzędu funkcji abstrakcyjnej $f(x)$ względem zmiennej x możemy zapisać w następujący sposób:

```
In [112]: f = sympy.Function('f')(x)
In [113]: sympy.diff(f, x) # Odpowiednik wywołania f.diff(x)
```

Out[113]: $\frac{d}{dx}f(x)$

Aby przedstawić pochodne wyższych rzędów, wystarczy powtórzyć symbol `x` na liście argumentów funkcji `sympy.diff` lub po przekazaniu symbolu zmiennej określić rząd pochodnej względem tej zmiennej przy użyciu liczby całkowitej:

```
In [114]: sympy.diff(f, x, x)
```

Out[114]: $\frac{d^2}{dx^2}f(x)$

```
In [115]: sympy.diff(f, x, 3)
```

Out[115]: $\frac{d^3}{dx^3}f(x)$ *# Odpowiednik wywołania sympy.diff(f, x, x, x)*

Kup ksi k Pole ksi k

Metodę tę można łatwo rozszerzyć na funkcje wielu zmiennych:

```
In [116]: g = sympy.Function('g')(x, y)
In [117]: g.diff(x, y) # Odpowiednik wywołania sympy.diff(g, x, y)
```

Out[117]: $\frac{\partial^2}{\partial x \partial y} g(x, y)$

```
In [118]: g.diff(x, 3, y, 2) # Odpowiednik wywołania sympy.diff(g, x, x, x, y, y)
```

Out[118]: $\frac{\partial^5}{\partial x^3 \partial y^2} g(x, y)$

Powyższe przykłady pokazują jedynie formalne pochodne nieokreślonych funkcji. Oczywiście w SymPy można również obliczać pochodne zdefiniowanych funkcji i wyrażeń. W efekcie powstają nowe wyrażenia reprezentujące pochodne. Na przykład z wykorzystaniem `sympy.diff` możemy łatwo obliczyć pochodne dowolnych wyrażeń matematycznych, takich jak wielomiany:

```
In [119]: expr = x**4 + x**3 + x**2 + x + 1
In [120]: expr.diff(x)
Out[120]: 4x³ + 3x² + 2x + 1
In [121]: expr.diff(x, x)
Out[121]: 2(6x² + 3x + 1)
In [122]: expr = (x + 1)**3 * y ** 2 * (z - 1)
In [123]: expr.diff(x, y, z)
Out[123]: 6y(x + 1)²
```

jak również funkcje trygonometryczne i bardziej złożone wyrażenia:

```
In [124]: expr = sympy.sin(x * y) * sympy.cos(x / 2)
In [125]: expr.diff(x)
```

Out[125]: $y \cos\left(\frac{x}{2}\right)\cos(xy) - \frac{1}{2}\sin\left(\frac{x}{2}\right)\sin(xy)$

```
In [126]: expr = sympy.functions.special.polynomials.hermite(x, 0)
In [127]: expr.diff(x).doit()
```

Out[127]: $\frac{2^x \sqrt{\pi}\, polygamma\left(0, -\frac{x}{2} + \frac{1}{2}\right)}{2\Gamma\left(-\frac{x}{2} + \frac{1}{2}\right)} + \frac{2^x \sqrt{\pi} \log(2)}{\Gamma\left(-\frac{x}{2} + \frac{1}{2}\right)}$

Pochodne zazwyczaj są stosunkowo łatwe do obliczenia, a funkcja `sympy.diff` powinna być w stanie obliczyć pochodną większości standardowych funkcji matematycznych zdefiniowanych w SymPy.

Zauważ, że w powyższych przykładach wywołanie `sympy.diff` skutkuje powstaniem nowego wyrażenia. Jeżeli chciałbyś przedstawić pochodną określonego wyrażenia w sposób symboliczny, możesz skorzystać z instancji klasy `sympy.Derivative`. Pierwszy argument jej konstruktora to wyrażenie, a dalsze to symbole odnoszące się do pochodnej, która ma być obliczona:

```
In [128]: d = sympy.Derivative(sympy.exp(sympy.cos(x)), x)
In [129]: d
```

Out[129]: $\frac{d}{dx} e^{\cos(x)}$

Kup ksi k Pole ksi k

Następnie przedstawiona w ten sposób pochodna może zostać obliczona z użyciem metody `doit` dostępnej w obiekcie klasy `sympy.Deriviative`:

```
In [130]: d.doit()
Out[130]: −e^cos(x) sin(x)
```

Ten wzorzec opóźnionej ewaluacji występuje w całej bibliotece SymPy, a możliwość pełnej kontroli nad tym, kiedy następuje ewaluacja wyrażenia, przydaje się w wielu sytuacjach, w szczególności w pracy z wyrażeniami, które można uprościć lub przekształcić jedynie przed obliczeniem ich wartości.

Całki

W SymPy do obliczania całek służy funkcja `sympy.integrate`, a wyrażenia zawierające formalne całki mogą być reprezentowane za pomocą obiektów klasy `sympy.Integral` (które, podobnie jak obiekty `sympy.Deriviative`, można jawnie obliczyć przez wywołanie `doit`). Istnieją dwa podstawowe rodzaje całek: całki oznaczone i całki nieoznaczone. Całka oznaczona ma określone granice całkowania i może być interpretowana jako obszar lub objętość, natomiast całka nieoznaczona nie ma granic i oznacza funkcję pierwotną (odwrotność pochodnej). Oba rodzaje całek są obsługiwane przez funkcję `sympy.integrate`.

Jeżeli funkcja `sympy.integrate` zostanie wywołana tylko z jednym argumentem będącym wyrażeniem matematycznym, obliczona zostanie całka nieoznaczona. W przypadku całek oznaczonych do funkcji `sympy.integrate` należy dodatkowo przekazać krotkę postaci `(x, a, b)` (gdzie `x` oznacza zmienną, względem której następuje całkowanie, a `a` i `b` to granice tej operacji). Dla funkcji jednej zmiennej $f(x)$ całka nieoznaczona i oznaczona obliczane są w następujący sposób:

```
In [131]: a, b, x, y = sympy.symbols("a, b, x, y")
     ...: f = sympy.Function("f")(x)
In [132]: sympy.integrate(f)
Out[132]: ∫f(x)dx
In [133]: sympy.integrate(f, (x, a, b))
Out[133]: ∫_a^b f(x)dx
```

W przypadku zastosowania tych metod do funkcji jawnych następuje obliczenie całek:

```
In [134]: sympy.integrate(sympy.sin(x))
Out[134]: −cos(x)
In [135]: sympy.integrate(sympy.sin(x), (x, a, b))
Out[135]: cos(a) − cos(b)
```

Całki oznaczone mogą również zawierać granice, których wartości rozciągają się od minus nieskończoności do plus nieskończoności. Do ich zapisania należy użyć symbolu nieskończoności reprezentowanego w SymPy przez `oo`:

```
In [136]: sympy.integrate(sympy.exp(-x**2), (x, 0, oo))
Out[136]: √π/2
In [137]: a, b, c = sympy.symbols("a, b, c", positive=True)
```


Kup ksi k   Pole ksi k

```
In [138]: sympy.integrate(a * sympy.exp(-((x-b)/c)**2), (x, -oo, oo))
```
Out[138]: $\sqrt{\pi} a c$

Rozwiązywanie całek symbolicznie jest ogólnie trudnym problemem, a SymPy nie będzie w stanie zwrócić symbolicznych wyników w zasadzie dla żadnej całki, którą możesz wymyślić. Gdy SymPy nie potrafi symbolicznie rozwiązać całki, zwracany jest obiekt klasy `sympy.Integral`, reprezentujący ją w formalny sposób.

```
In [139]: sympy.integrate(sympy.sin(x * sympy.cos(x)))
```
Out[139]: $\int \sin\left(x \cos(x)\right) dx$

`sympy.integrate` umie również obliczać całki wyrażeń zawierających wiele zmiennych. W przypadku całki nieoznaczonej z wyrażenia wielu zmiennych należy wyraźnie zaznaczyć, która ze zmiennych jest zmienną całkowania:

```
In [140]: expr = sympy.sin(x*sympy.exp(y))
In [141]: sympy.integrate(expr, x)
```
Out[141]: $-e^{-y} \cos(x e^{y})$

```
In [142]: expr = (x + y)**2
In [143]: sympy.integrate(expr, x)
```
Out[143]: $\frac{x^3}{3} + x^2 y + x y^2$

Całki wielokrotne można obliczać przez przekazanie funkcji `sympy.integrate` więcej niż jednego symbolu lub wielu krotek zawierających symbole i związane z nimi granice całkowania:

```
In [144]: sympy.integrate(expr, x, y)
```
Out[144]: $\frac{x^3 y}{3} + \frac{x^2 y^2}{2} + \frac{x y^3}{3}$

```
In [145]: sympy.integrate(expr, (x, 0, 1), (y, 0, 1))
```
Out[145]: $\frac{7}{6}$

Szeregi

Rozwinięcie funkcji w szereg to ważne narzędzie w wielu dziedzinach obliczeniowych, pozwalające na przedstawienie dowolnej funkcji w postaci wielomianu o współczynnikach będących wartościami kolejnych pochodnych w punkcie, wokół którego przeprowadzane jest rozwinięcie. Obcięcie szeregu po n-tym wyrazie pozwala uzyskać aproksymację rozwijanej funkcji n-tego rzędu. W SymPy rozwinięcie funkcji lub wyrażenia w szereg można obliczyć za pomocą funkcji `sympy.series` lub metody `series` dostępnej w obiekcie reprezentującym wyrażenie SymPy. Pierwszym argumentem `sympy.series` jest funkcja lub wyrażenie, które mają zostać rozwinięte, po nich zaś należy podać symbol, w odniesieniu do którego ma nastąpić rozwinięcie (można go pominąć w przypadku wyrażeń i funkcji jednej zmiennej). Ponadto istnieje możliwość określenia: punktu, w którego otoczeniu nastąpi rozwinięcie (parametr `x0`, domyślnie `x0=0`), długości rozwinięcia (parametr `n`, domyślnie `n=6`) oraz kierunku, w którym następuje obliczanie wartości szeregu, tj. czy w otoczeniu punktu `x0` stosowana jest lewo- czy prawostronna granica (parametr `dir`, domyślnie granica prawostronna `dir="+"`).

Kup ksi k Pole ksi k

Dla nieokreślonej funkcji *f*(x) rozwinięcie w szereg ograniczony do sześciu wyrazów w otoczeniu punktu x0=0 oblicza się w następujący sposób:

```
In [146]: x, y = sympy.symbols("x, y")
In [147]: f = sympy.Function("f")(x)
In [148]: sympy.series(f, x)
```

Out[148]: $f(0)+x)\frac{d}{dx}f(x)\Big|_{x=0}+\frac{x^2}{2}\frac{d^2}{dx^2}f(x)\Big|_{x=0}+\frac{x^3}{6}\frac{d^3}{dx^3}f(x)\Big|_{x=0}+\frac{x^4}{24}\frac{d^4}{dx^4}f(x)\Big|_{x=0}+\frac{x^5}{120}\frac{d^5}{dx^5}f(x)\Big|_{x=0}+O\left(x^6\right)$

Aby zmienić punkt, w którego otoczeniu następuje rozwinięcie, należy określić wartość argumentu x0, tak jak w poniższym przykładzie:

```
In [149]: x0 = sympy.Symbol("{x_0}")
In [150]: f.series(x, x0, n=2)
```

Out[150]: $f(x_0)+(x-x_0)\frac{d}{d\xi_1}f(\xi_1)\Big|_{\xi_1=x_0}+O\left((x-x_0)^2; x\mapsto x_0\right)$

Aby ograniczyć długość rozwinięcia do dwóch składników, w powyższym przykładzie skorzystano z parametru n=2. Zwróć uwagę, że błąd spowodowany obcięciem szeregu jest reprezentowany przez obiekt wskazujący jego rząd $O(\ldots)$ (ang. *order object*). Obiekt ten przydaje się podczas śledzenia rzędu wyrażenia w trakcie operacji na szeregach, na przykład w trakcie dodawania lub mnożenia różnych rozwinięć. Jednak aby obliczyć konkretną wartość danego rozwinięcia, trzeba je usunąć. Można to zrobić z wykorzystaniem metody removeO:

```
In [151]: f.series(x, x0, n=2).removeO()
```

Out[151]: $f(x_0)+(x-x_0)\frac{d}{d\xi_1}f(\xi_1)\Big|_{\xi_1=x_0}$

Poza pokazanymi powyżej rozwinięciami nieokreślonej funkcji f(x) SymPy pozwala obliczać rozwinięcia dla określonych funkcji i wyrażeń, w wyniku których otrzymasz konkretne formuły matematyczne. Można w ten sposób obliczyć na przykład powszechnie znane rozwinięcia wielu standardowych funkcji:

```
In [152]: sympy.cos(x).series()
```

Out[152]: $1-\frac{x^2}{2}+\frac{x^4}{24}+O(x^6)$

```
In [153]: sympy.sin(x).series()
```

Out[153]: $1-\frac{x^3}{6}+\frac{x^5}{120}+O(x^6)$

```
In [154]: sympy.exp(x).series()
```

Out[154]: $1+x+\frac{x^2}{2}+\frac{x^3}{6}+\frac{x^4}{24}+\frac{x^5}{120}+O(x^6)$

```
In [155]: (1/(1+x)).series()
```

Out[155]: $1-x+x^2-x^3+x^4-x^5+O(x^6)$

a także dowolnych innych funkcji i wyrażeń zawierających symbole, które w ogólności mogą być również funkcjami wielu zmiennych:

```
In [156]: expr = sympy.cos(x) / (1 + sympy.sin(x * y))
In [157]: expr.series(x, n=4)
```


Kup ksi k Pole ksi k

Out[157]: $1-xy+x^2\left(y^2-\frac{1}{2}\right)+x^3\left(-\frac{5y^3}{6}+\frac{y}{2}\right)+O(x^4)$

```
In [158]: expr.series(y, n=4)
```

Out[158]: $\cos(x)-xy\cos(x)+x^2y^2\cos(x)-\frac{5x^3y^3\cos(x)}{6}+O(y^4)$

Granice

Granice są kolejnym ważnym narzędziem analizy matematycznej. Granica funkcji reprezentuje jej wartość w momencie, w którym jeden z jej argumentów zbliża się do określonej wartości lub plus/minus nieskończoności. Przykładem granicy jest jedna z definicji pochodnej:

$$\frac{d}{dx}f(x)=\lim_{h\to 0}\frac{f(x+h)-f(x)}{h}$$

Chociaż granice to narzędzie bardziej teoretyczne, które, w odróżnieniu od choćby rozwinięcia w szereg, nie ma zbyt wielu praktycznych zastosowań, możliwość ich obliczenia przy użyciu SymPy nadal może być przydatna. W SymPy granice można obliczać za pomocą funkcji `sympy.limit`, która przyjmuje następujące argumenty: wyrażenie, symbol, którego wartość się zmienia, i wartość, do której zmierza zadany symbol. Na przykład granicę funkcji $\frac{\sin x}{x}$ przy x dążącym do zera, tj. $\lim_{x\to 0}\frac{\sin x}{x}$, można obliczyć w następujący sposób:

```
In [159]: sympy.limit(sympy.sin(x) / x, x, 0)
Out[159]: 1
```

Jak można się było spodziewać, wartość tej granicy to 1. Funkcja `sympy.limit` może służyć również do obliczania granic symbolicznych, na przykład pochodnej przy użyciu pokazanej powyżej definicji (oczywiście bardziej efektywne byłoby zastosowanie funkcji `sympy.diff`):

```
In [160]: f = sympy.Function('f')
     ...: x, h = sympy.symbols("x, h")
In [161]: diff_limit = (f(x + h) - f(x))/h
In [162]: sympy.limit(diff_limit.subs(f, sympy.cos), h, 0)
Out[162]: -sin(x)
In [163]: sympy.limit(diff_limit.subs(f, sympy.sin), h, 0)
Out[163]: cos(x)
```

Bardziej praktycznym przykładem użycia granic jest poszukiwanie asymptoty ukośnej funkcji, gdy zmienna zależna dąży na przykład do nieskończoności. Dana jest funkcja $f(x)=\frac{x^2-3x}{2x-2}$. Powiedzmy, że interesuje Cię jej zachowanie dla dużych wartości x. Asymptota ukośna ma ogólną postać $f(x) = px + q$, a parametry p i q można obliczyć z wykorzystaniem `sympy.limit` w następujący sposób:

```
In [164]: expr = (x**2 - 3*x) / (2*x - 2)
In [165]: p = sympy.limit(expr/x, x, sympy.oo)
In [166]: q = sympy.limit(expr - p*x, x, sympy.oo)
In [167]: p, q
```

Out[167]: $\left(\frac{1}{2},-1\right)$

Zatem dla dużych wartości x wartości $f(x)$ dążą do wartości funkcji liniowej $f(x)=\frac{x}{2}-1$.

Kup ksi k Pole ksi k

Sumy i iloczyny uogólnione

Sumy i iloczyny uogólnione mogą być symbolicznie reprezentowane za pomocą klas `sympy.Sum` i `sympy.Product`. Oba konstruktory jako pierwszy argument przyjmują wyrażenie, drugi zaś argument to krotka postaci `(n, n1, n2)`, gdzie `n` jest symbolem, a `n1` i `n2` są dolną i górną granicą zmiany wartości `n`, odpowiednio w sumie i iloczynie. Po utworzeniu obiektów `sympy.Sum` i `sympy.Product` ich ewaluacja jest możliwa przy użyciu metody `doit`:

```
In [168]: n = sympy.symbols("n", integer=True)
In [169]: x = sympy.Sum(1/(n**2), (n, 1, oo))
In [170]: x
```

Out[170]: $\sum_{n=1}^{\infty} \frac{1}{n^2}$

```
In [171]: x.doit()
```

Out[171]: $\frac{\pi^2}{6}$

```
In [172]: x = sympy.Product(n, (n, 1, 7))
In [173]: x
```

Out[173]: $\prod_{1}^{7} n$

```
In [174]: x.doit()
Out[174]: 5040
```

Zauważ, że w powyższym przykładzie górną granicą sumy jest nieskończoność. Jest zatem jasne, że wartość tej sumy nie została obliczona przez dodawanie do siebie kolejnych liczb, lecz analitycznie. SymPy może ewaluować wiele sum tego typu, w tym sumy, których składniki zawierają zmienne symboliczne inne niż indeks sumowania, na przykład:

```
In [175]: x = sympy.Symbol("x")
In [176]: sympy.Sum((x)**n/(sympy.factorial(n)), (n, 1, oo)).doit().simplify()
Out[176]: e^x − 1
```

Równania

Rozwiązywanie równań to jedna z podstawowych i niezmiernie ważnych umiejętności matematycznych, która znajduje zastosowanie w prawie każdej gałęzi nauki i techniki. SymPy potrafi rozwiązywać symbolicznie różnorodne równania, jednak wiele równań po prostu nie ma rozwiązania analitycznego. Jeżeli jednak równanie lub układ równań da się rozwiązać w sposób analityczny, jest duża szansa, że SymPy będzie w stanie to zrobić. Jeżeli równanie nie jest rozwiązywalne analitycznie, jedyną opcją może być zastosowanie jednej z metod numerycznych.

W najprostszym przypadku rozwiązywanie równania sprowadza się do rozwiązania pojedynczego równania jednej zmiennej bez dodatkowych parametrów. Przykładem takiego problemu jest znalezienie wartości x spełniającej równanie wielomianowe stopnia drugiego $x^2 + 2x - 3 = 0$. To równanie jest oczywiście łatwe do rozwiązania, nawet na papierze, ale do znalezienia jego rozwiązania w SymPy można zastosować funkcję `sympy.solve`:

```
In [177]: x = sympy.Symbol("x")
In [178]: sympy.solve(x**2 + 2*x - 3)
Out[178]: [−3,1]
```


Kup ksi k   Pole ksi k

Funkcja zwróciła dwa rozwiązania $x = -3$ i $x = 1$. Argumentem funkcji sympy.solve jest wyrażenie, które zostanie rozwiązane przy założeniu, że jego wartość jest równa zero. Gdy wyrażenie zawiera więcej niż jeden symbol, zmienna, której wartość jest poszukiwana, musi zostać podana jako drugi argument. Na przykład:

```
In [179]: a, b, c = sympy.symbols("a, b, c")
In [180]: sympy.solve(a * x**2 + b * x + c, x)
```

Out[180]: $\left[\frac{1}{2a}\left(-b+\sqrt{b^2-4ac}\right), -\frac{1}{2a}\left(b+\sqrt{b^2-4ac}\right)\right]$

W tym przypadku otrzymane rozwiązanie to wyrażenie zależne od symboli reprezentujących parametry występujące w równaniu.

Funkcja sympy.solve może również rozwiązywać inne typy równań, w tym równania trygonometryczne:

```
In [181]: sympy.solve(sympy.sin(x) - sympy.cos(x), x)
```

Out[181]: $\left[-\frac{3\pi}{4}\right]$

oraz równania, których rozwiązanie można wyrazić w kategoriach funkcji specjalnych:

```
In [182]: sympy.solve(sympy.exp(x) + 2 * x, x)
```

Out[182]: $\left[-\mathrm{LambertW}\left(\frac{1}{2}\right)\right]$

Niestety, podczas rozwiązywania równań ogólnych, nawet jednej zmiennej, często spotyka się równania, które nie mają rozwiązań analitycznych, lub równania, których SymPy nie jest w stanie rozwiązać. W takich przypadkach SymPy zwróci rozwiązanie formalne, które w razie potrzeby można poddać ewaluacji numerycznej, lub zgłosi wyjątek, jeżeli w jego zasobach nie znajduje się żadna metoda, która potrafiłaby rozwiązać formalnie dane równanie:

```
In [183]: sympy.solve(x**5 - x**2 + 1, x)
Out[183]: [CRootOf(x5 – x2 + 1,0), CRootOf(x5 – x2 + 1,1), CRootOf(x5 – x2 + 1,2),
          CRootOf(x5 – x2 + 1,3),CRootOf(x5 – x2 + 1,4)]
In [184]: sympy.solve(sympy.tan(x) + x, x)
---------------------------------------------------------------------------
↪NotImplementedError                           Traceback (most recent call last)
...
NotImplementedError: multiple generators [x, tan(x)] No algorithms are implemented to
↪solve equation x + tan(x)
```

Rozwiązanie układu równań więcej niż jednej zmiennej w SymPy to proste uogólnienie procedury stosowanej do rozwiązywania równań jednej zmiennej. Aby rozwiązać układ równań, w pierwszym argumencie sympy.solve zamiast pojedynczego wyrażenia przekazuje się listę wyrażeń, a w drugim listę symboli, które są zmiennymi. Dwa poniższe przykłady pokazują, w jaki sposób można rozwiązać dwa — liniowy i nieliniowy — układy równań dwóch zmiennych x i y:

```
In [185]: eq1 = x + 2 * y - 1
     ...: eq2 = x - y + 1
In [186]: sympy.solve([eq1, eq2], [x, y], dict=True)
```


Kup książkę Poleć książkę

Out[186]: $\left[\left\{x: -\frac{1}{3}, y: \frac{2}{3}\right\}\right]$

```
In [187]: eq1 = x**2 - y
     ...: eq2 = y**2 - x
In [188]: sols = sympy.solve([eq1, eq2], [x, y], dict=True)
In [189]: sols
```

Out[189]: $\left[\{x:0, y:0\}, \{x:1, y:1\}, \left\{x: -\frac{1}{2}+\frac{\sqrt{3}i}{2}, y: -\frac{1}{2}-\frac{\sqrt{3}i}{2}\right\}, \left\{x: \frac{1}{4}, y: -\frac{\left(1-\sqrt{3}i\right)^2}{2}+\frac{\sqrt{3}i}{2}\right\}\right]$

Zauważ, że w obu przypadkach funkcja `sympy.solve` zwraca listę, której poszczególne elementy reprezentują rozwiązania układu. W wywołaniach zastosowałem również opcjonalny argument `dict=True`, powodujący, że każde zwrócone rozwiązania ma formę słownika, w którym dany symbol jest powiązany z odpowiadającą mu wartością. Słownik ten można wygodnie wykorzystać na przykład w wywołaniu metody `sub`, która w poniższym kodzie służy do sprawdzenia, czy każde znalezione rozwiązanie rzeczywiście spełnia obydwa równania:

```
In [190]: [eq1.subs(sol).simplify() == 0 and eq2.subs(sol).simplify() == 0 for sol in sols]
Out[190]: [True, True, True, True]
```

Algebra liniowa

Algebra liniowa to kolejna z podstawowych gałęzi matematyki mająca zastosowanie w obliczeniach naukowych i technicznych. Przedmiotem zainteresowania algebry liniowej są wektory, przestrzenie wektorowe i odwzorowania liniowe pomiędzy nimi, które można przedstawić w postaci macierzy. W SymPy wektory i macierze, których elementy mogą być liczbami, symbolami, a nawet dowolnymi wyrażeniami symbolicznymi, mogą być reprezentowane za pomocą obiektów klasy `sympy.Matrix`. Aby utworzyć macierz zawierającą wartości liczbowe, możesz, podobnie jak w przypadku omawianych w rozdziale 2. tablic NumPy, przekazać listę wartości do konstruktora `sympy.Matrix`:

```
In [191]: sympy.Matrix([1, 2])
```

Out[191]: $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$

```
In [192]: sympy.Matrix([[1, 2]])
Out[192]: [1 2]
In [193]: sympy.Matrix([[1, 2], [3, 4]])
```

Out[193]: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

Jak pokazuje ten przykład, przekazanie pojedynczej listy powoduje powstanie wektora kolumnowego, a do utworzenia macierzy konieczna jest zagnieżdżona lista wartości. Zauważ, że w przeciwieństwie do omawianych w rozdziale 2. tablic wielowymiarowych NumPy obiekt `sympy.Matrix` jest przeznaczony tylko do reprezentacji tablic dwuwymiarowych, czyli macierzy i wektorów. Innym sposobem na utworzenie nowej instancji `sympy.Matrix` jest przekazanie do konstruktora informacji o liczbie wierszy i kolumn oraz funkcji przekształcającej indeksy kolumny i wiersza w wartość elementu:

Kup ksi k   Pole ksi k

```
In [194]: sympy.Matrix(3, 4, lambda m, n: 10 * m + n)
```

Out[194]: $$\begin{bmatrix} ax_1 + bx_2 \\ cx_1 + dx_2 \end{bmatrix}$$

Najpotężniejszą cechą obiektów macierzowych SymPy, która odróżnia je na przykład od tablic NumPy, jest możliwość umieszczenia w nich elementów będących wyrażeniami symbolicznymi. Na przykład dowolną macierz 2×2 można przedstawić za pomocą zmiennych symbolicznych odpowiadających wszystkim jej elementom:

```
In [195]: a, b, c, d = sympy.symbols("a, b, c, d")
In [196]: M = sympy.Matrix([[a, b], [c, d]])
In [197]: M
```

Out[197]: $$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Oczywiście macierze tego typu mogą być także stosowane w obliczeniach, których wyniki zostają następnie sparametryzowane za pomocą wartości elementów symbolicznych. Dla obiektów macierzowych zaimplementowano popularne operatory arytmetyczne, należy jednak zauważyć, że w przypadku operacji na macierzach SymPy operator mnożenia * oznacza mnożenie macierzy:

```
In [198]: M * M
```

Out[198]: $$\begin{bmatrix} a^2 + bc & ab + bd \\ ac + cd & bc + d^2 \end{bmatrix}$$

```
In [199]: x = sympy.Matrix(sympy.symbols("x_1, x_2"))
In [200]: M * x
```

Out[200]: $$\begin{bmatrix} ax_1 & bx_2 \\ cx_1 & dx_2 \end{bmatrix}$$

Oprócz operacji arytmetycznych zaimplementowanych jako funkcje SymPy i metody klasy `sympy.Matrix` w bibliotece zaimplementowano wiele znanych z algebry liniowej operacji wektorowych i macierzowych. Tabela 3.4 zawiera przegląd często używanych funkcji związanych z algebrą liniową (aby zapoznać się z pełną listą dostępnych możliwości, zajrzyj do dokumentacji `sympy.Matrix`). Macierze SymPy mogą być także używane w sposób zorientowany na elementy z wykorzystaniem operatorów indeksowania i zakresu. Działanie tych operatorów odpowiada działaniu znanemu z tablic NumPy, które omówiłem w rozdziale 2.

Jako przykład problemu niedającego się rozwiązać za pomocą czysto numerycznego podejścia, a możliwego do rozwiązania za pomocą symbolicznej algebry liniowej w SymPy, rozważ następujący sparametryzowany układ równań liniowych:

$$x + py = b_1,$$
$$qx + y = b_2,$$

który miałby zostać rozwiązany dla nieznanych zmiennych x i y. W układzie tym p, q, b_1 i b_2 są nieokreślonymi parametrami. W postaci macierzowej układ ten można zapisać jako:

$$\begin{pmatrix} 1 & p \\ q & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

Kup ksi k Pole ksi k

Tabela 3.4. Wybrane funkcje i metody wykorzystywane z macierzami w SymPy

Funkcja/metoda	Opis
`transpose/T`	Transpozycja macierzy.
`adjoint/H`	Macierz dołączona.
`trace`	Ślad macierzy (suma wyrazów na przekątnej).
`det`	Wyznacznik macierzy.
`inv`	Macierz odwrotna.
`LUdecomposition`	Rozkład LU macierzy.
`LUsolve`	Rozwiązanie układu równań (znalezienie wektora x) zapisanego w postaci $Mx = b$ z wykorzystaniem rozkładu LU.
`QRdecomposition`	Rozkład QR macierzy.
`QRsolve`	Rozwiązanie układu równań (znalezienie wektora x) zapisanego w postaci $Mx = b$ z wykorzystaniem rozkładu QR.
`diagonalize`	Diagonalizacja macierzy M, czyli rozkład macierzy do postaci $D = P^{-1}MP$, gdzie D jest macierzą diagonalną.
`norm`	Norma macierzy.
`nullspace`	Jądro (zbiór wektorów rozpinających przestrzeń zerową) macierzy.
`rank`	Rząd macierzy.
`singular_values`	Oblicza wartości osobliwe (singularne) macierzy.
`solve`	Rozwiązuje układ równań postaci $Mx = b$

Rozwiązując ten układ metodami czysto numerycznymi, przed przystąpieniem do rozwiązania z wykorzystaniem na przykład metody rozkładu LU (lub poszukiwania macierzy odwrotnej) macierzy po lewej stronie równania należałoby najpierw wybrać określone wartości parametrów p i q. Dzięki zastosowaniu podejścia symbolicznego możesz rozwiązać ten układ w taki sam sposób jak podczas ręcznych obliczeń. W SymPy możesz po prostu zdefiniować symbole nieznanych zmiennych i parametrów oraz utworzyć wymagane obiekty macierzowe:

```
In [201]: p, q = sympy.symbols("p, q")
In [202]: M = sympy.Matrix([[1, p], [q, 1]])
In [203]: M
```

Out[203]: $\begin{bmatrix} 1 & p \\ q & 1 \end{bmatrix}$

```
In [204]: b = sympy.Matrix(sympy.symbols("b_1, b_2"))
In [205]: b
Out[205]:[b1 b2]
```

Następnie do rozwiązania układu możesz zastosować na przykład metodę `LUsolve`:

```
In [206]: x = M.LUsolve(b)
In [207]: x
```


Kup ksi k Pole ksi k

Out[207]:
$$\begin{bmatrix} b_1 - \frac{p(-b_1 q + b_2)}{-pq+1} \\ \frac{-b_1 q + b_2}{-pq+1} \end{bmatrix}$$

Innym sposobem jest bezpośrednie obliczenie odwrotności macierzy `M` i przemnożenie jej przez wektor `b`:

```
In [208]: x = M.inv() * b
In [209]: x
```

Out[209]:
$$\begin{bmatrix} b_1\left(\frac{pq}{-pq+1}+1\right) - \frac{b_2 p}{-pq+1} \\ -\frac{b_1 q}{-pq+1} + \frac{b_2}{-pq+1} \end{bmatrix}$$

Jednakże znalezienie odwrotności macierzy to problem trudniejszy niż znalezienie rozkładu LU. Dlatego jeżeli jedynym celem prowadzonych obliczeń jest rozwiązanie równania $Mx = b$ (tak jak w powyższym przypadku), to wydajniejsze będzie wykorzystanie rozkładu LU. Różnice stają się szczególnie widoczne w przypadku większych układów równań. Użycie obu metod daje w efekcie wyrażenie symboliczne opisujące rozwiązanie, którego wartość liczbowa jest łatwa do ustalenia dla dowolnych parametrów, bez konieczności ponownego rozwiązania układu. Przykład ten pokazuje siłę obliczeń symbolicznych i potwierdza, że czasem mogą one być lepszym rozwiązaniem niż bezpośrednie obliczenia numeryczne. Rozważany tutaj przykład można oczywiście łatwo rozwiązać ręcznie, ale wraz ze wzrostem liczby równań i nieokreślonych parametrów ręcznie prowadzone obliczenia analityczne szybko stają się zbyt złożone i uciążliwe. Wykorzystanie systemów CAS, takich jak SymPy, umożliwia analityczne rozwiązywanie o wiele bardziej złożonych problemów, których rozwiązanie na papierze nie byłoby możliwe.

Podsumowanie

Rozdział ten zawiera wprowadzenie do wspomaganych komputerowo obliczeń symbolicznych z użyciem Pythona i biblioteki SymPy. Chociaż techniki analityczne i numeryczne często są rozpatrywane osobno, faktem jest, że metody analityczne leżą u podstaw wszystkich metod obliczeniowych i są niezbędne podczas opracowywania metod i algorytmów numerycznych. Niezależnie od tego, czy obliczenia analityczne są prowadzone ręcznie czy przy użyciu systemu CAS, takiego jak SymPy, stanowią one niezbędne narzędzie w pracy obliczeniowej. Zachęcam wszystkich czytelników do przyjęcia następującego podejścia. Ponieważ metody analityczne i numeryczne są ze sobą ściśle powiązane, często warto rozpocząć analizę problemu obliczeniowego od zastosowania metod analitycznych i symbolicznych. Dopiero kiedy takie metody zawiodą, należy skorzystać z metod numerycznych. Użycie metod numerycznych przed przeprowadzeniem analizy problemu od strony analitycznej spowoduje, że będziesz rozwiązywał bardziej skomplikowany problem obliczeniowy, niż jest to naprawdę konieczne.

Kup ksi k Pole ksi k

Materiały dodatkowe

Krótkie i szybkie wprowadzenie do SymPy znajdziesz na przykład w książce Lamy (2013). Oficjalna dokumentacja SymPy zawiera również świetny poradnik ułatwiający rozpoczęcie pracy z SymPy. Jest on dostępny na stronie *http://docs.sympy.org/latest/tutorial/index.html*.

Bibliografia

R. Lamy, *Instant SymPy Starter*, Packt, Mumbai 2013.

Kup ksi k

Pole ksi k

Kup ksi k

Pole ksi k

PROGRAM PARTNERSKI

GRUPY HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

Python: język, który naukowcy lubią najbardziej!

Nie tylko programiści lubią Pythona. Również naukowcy i analitycy danych coraz częściej korzystają z tego języka, zwłaszcza że przed praktykami obliczeniowymi otwierają się niespotykane możliwości. Rozwój sprzętu, oprogramowania i algorytmów pozwala śmiało wkraczać w nowe obszary zastosowania i tworzyć nowe branże. W dalszym ciągu jednak prowadzenie obliczeń pozostaje dziedziną interdyscyplinarną, wymagającą wiedzy matematycznej i myślenia naukowego. Jeśli chce się wykorzystać do obliczeń nowoczesne technologie, takie jak Python wraz z szerokim ekosystemem bibliotek i rozszerzeń, trzeba też posiadać praktyczne umiejętności programowania.

W tej książce wyczerpująco przedstawiono nowoczesne metody rozwiązywania problemów obliczeniowych z tak różnych dziedzin, jak badania naukowe, inżynieria, finanse czy analiza danych za pomocą Pythona i jego bibliotek. Omówiono również wiele technik, w tym obliczenia oparte na tablicach, obliczenia symboliczne, metody wizualizacji danych, numeryczne operacje wejścia-wyjścia, rozwiązywanie równań, optymalizację, interpolację czy całkowanie. Pokazano także, jak rozwiązywać problemy obliczeniowe charakterystyczne dla takich dziedzin jak rozwiązywanie równań różniczkowych, analiza danych, modelowanie statystyczne i uczenie maszynowe. Znalazło się tu też wiele studiów przypadków, ukazujących zastosowanie Pythona w analizie danych i statystyce.

W książce między innymi:

- wektory i macierze w NumPy
- wykresy i wizualizacje danych w Matplotlib
- analiza danych z pandas i SciPy
- modelowanie statystyczne i uczenie maszynowe ze statsmodels i scikit-learn
- optymalizacja kodu za pomocą Numba i Cython

Dr Robert Johansson jest doświadczonym programistą Pythona. Od ponad dziesięciu lat zajmuje się obliczeniami naukowymi. Współtworzył popularny framework QuTiP do symulacji dynamiki układów kwantowych oraz wiele bibliotek w Pythonie. Jest pasjonatem obliczeń, programowania, a także nauczania i popularyzowania nowatorskich, powtarzalnych i rozszerzalnych metod obliczeniowych.

Apress