Lecture 13

Python's ecosystem for scientific computing.

Advanced Programming - SISSA, UniTS, 2024-2025

Giuseppe Alessio D'Inverno

16 Dec 2024

Outline

Part of these notes is re-adapted from these lectures (license).

The role of Python in modern scientific computing

The role of Python in modern scientific computing

Python has emerged as a pivotal language in scientific computing, distinguished by:

- Intuitive and readable syntax, making coding accessible to scientists from various fields.
- A vast array of libraries and tools tailored for scientific applications.

Python's versatility extends across numerous scientific domains:

- In physics, it's used for simulations and theoretical calculations.
- In biology and chemistry, Python aids in molecular modeling and genomic data analysis.
- Its application in astronomy includes data processing from telescopes and space missions.
- In environmental science, it's pivotal in climate modeling and biodiversity studies.

Python's library ecosystem for scientific computing

The power of Python in scientific computing is amplified by its extensive library ecosystem:

- NumPy and SciPy are fundamental for numerical computations.
- pandas enhances data manipulation and analysis capabilities.
- Matplotlib and Seaborn excel in creating scientific visualizations.
- TensorFlow and PyTorch are at the forefront of machine learning research and applications.

Python's role in democratizing scientific research is underscored by its open-source nature, fostering collaboration and innovation.

Real-world applications of Python in scientific research

Python's impact in scientific research is evident through numerous real-world applications:

- In physics, it has been used to analyze data from the Large Hadron Collider.
- In biology, Python is integral in genome sequencing projects like the Human Genome Project.
- Environmental scientists utilize Python in modeling the effects of climate change on different ecosystems.
- In astronomy, it played a key role in processing the first image of a black hole.

These applications underscore Python's versatility and effectiveness in advancing scientific knowledge.

How to get your system ready

Most Python libraries can be installed with `pip`, with `Conda`, with a package manager on Linux and macOS, or from source.

- Using `pip`:

  ```
  pip install numpy scipy matplotlib seaborn pandas
  ```

- Using `Conda`:

  ```
  conda create -n sci-env
  conda activate sci-env
  conda install numpy scipy matplotlib seaborn pandas
  ```

Best practices in setting up a scientific computing environment include creating isolated environments and maintaining updated library versions.

NumPy

Introduction

NumPy is a core library for numerical computing in Python, offering an efficient interface for working with arrays and matrices. Known for its high performance, it forms the basis of many other scientific computing tools.

To get started with NumPy:

```
import numpy as np
```

NumPy arrays provide an efficient way to store and manipulate numerical data, offering advantages over traditional Python lists, particularly in terms of performance and functionality.

Creating NumPy arrays

There are several ways to create arrays in NumPy:

- Converting from Python lists or tuples.
- Using array-generating functions like `np.arange`, `np.linspace`, etc.
- Reading data from files.

From lists

Creating arrays from Python lists:

```python
# Creating a vector from a list
vector = np.array([1, 2, 3, 4])

# Creating a matrix from a nested list
matrix = np.array([[1, 2], [3, 4]])
```

These arrays are instances of NumPy's `ndarray` type.

The shape of an array can be accessed using the `shape` attribute:

```python
print(vector.shape)  # Output: (4,)
print(matrix.shape)  # Output: (2, 2)
```

Lists vs. NumPy arrays

NumPy arrays offer several advantages over Python lists, such as:

- Faster access in reading and writing items.
- More convenient and efficient for mathematical operations.
- Occupying less memory.

Unlike Python lists, NumPy arrays are statically typed, homogeneous, memory-efficient, and support efficient mathematical operations implemented in compiled languages like C and Fortran.

`dtype` (1/2)

The `dtype` (data type) property reveals the type of an array's data:

```
M.dtype
```

> dtype('int64')

Attempting to assign an incompatible type raises an error:

```
M[0, 0] = "hello"
```

> ValueError: invalid literal for long() with base 10: 'hello'

`dtype` (2/2)

Explicitly defining the array data type during creation is possible using the `dtype` keyword argument:

```
M = np.array([[1, 2], [3, 4]], dtype=complex)
```

Common data types for `dtype` include `int`, `float`, `complex`, `bool`, and others. Additionally, bit sizes like `int64`, `int16`, `float128`, and `complex128` can be specified.

Using array-generating functions (1/2)

NumPy provides various functions to generate arrays:

```
x = np.arange(0, 10, 1)  # Arguments: start, stop, step.

# Using linspace, where both end points are included.
x = np.linspace(0, 10, 25)

# Create an array with logarithmically spaced elements.
x = np.logspace(0, 10, 10, base=np.e)

x, y = np.mgrid[0:5, 0:5]  # Similar to 'meshgrid' in MATLAB.
```

Using array-generating functions (2/2)

Additional array-generating functions include random number generation, diagonal matrices, zeros, and ones:

```
np.random.rand(5, 5)  # Uniform random numbers in [0, 1].
np.random.randn(5, 5)  # Standard normal distributed random numbers.
np.diag([1, 2, 3])  # Diagonal matrix.
np.diag([1, 2, 3], k=1)  # Diagonal with an offset from the main diagonal.
np.zeros((3, 3))
np.ones((3, 3))
```

File I/O (1/2)

Comma-separated values (CSV)

Reading data from comma-separated values (CSV) files into NumPy arrays is accomplished using `np.genfromtxt`:

```
data = np.genfromtxt('filename.csv')
data.shape
```

Storing a NumPy array to a CSV file can be done with `np.savetxt`:

```
M = random.rand(3, 3)
np.savetxt("random-matrix.csv", M)
np.savetxt("random-matrix.csv", M, fmt='%.5f')  # fmt specifies the format.
```

File I/O (2/2)

NumPy's native file format

NumPy provides its own file format for storing and reading array data using `np.save` and `np.load`:

```
np.save("random-matrix.npy", M)

np.load("random-matrix.npy")
```

Manipulating arrays

Indexing

Array elements are accessed using square brackets and indices for reading and writing:

```
# v is a vector, taking one index.
v[0]

# M is a matrix, taking two indices.
M[1, 1]

# Omitting an index returns the whole row or a N-1 dimensional array.
M[1]

M[1, :]  # Row 1.
M[:, 1]  # Column 1.
```

Index slicing

Index slicing (`M[lower:upper:step]`) extracts portions of an array:

```
A = np.array([1, 2, 3, 4, 5])
A[1:3]
A[0:2] = [-2, -3]

A[::]  # Lower, upper, and step default to the beginning, end, and 1.
A[::2]  # Step is 2, lower and upper default to the beginning and end.
A[:3]  # First three elements.
A[3:]  # Elements from index 3.

# Negative indices count from the end of the array.
A[-1]  # The last element in the array.
A[-3:]  # The last three elements.
```

Fancy indexing

Fancy indexing involves using an array or list in place of an index:

```
row_indices = [1, 2, 3]
A[row_indices]

col_indices = [1, 2, -1]
A[row_indices, col_indices]
```

Index masks, arrays of type `bool`, can also be used to select elements:

```
mask = np.array([True, False, True])
A[mask]

x = np.arange(0, 10, 0.5)
mask = (5 < x) * (x < 7.5)
x[mask]
```

Linear algebra (1/2)

NumPy is well-suited for linear algebra operations:

- Scalar-array and element-wise array-array operations.
- Matrix multiplication using the `dot` function or `@` operator.
- Computing inverses, determinants, and solving linear equations.

Scalar-array operations

Arithmetic operators are employed for scalar-array operations:

```
v1 = np.arange(0, 5)
v1 * 2
v1 + 2
A * 2
A + 2
```

Linear algebra (2/2)

Element-wise array-array operations

When we add, subtract, multiply and divide arrays with each other, the default behavior is **element-wise** operations:

```
A * A
v1 * v1
```

If we multiply arrays with compatible shapes, we get an element-wise multiplication of each row:

```
A * v1
```

Matrix algebra (1/2)

What about matrix multiplication? There are two ways. We can either use the `@` or `dot` function, which applies a matrix-matrix, matrix-vector, or inner vector multiplication to its arguments:

```
A @ A
np.dot(A, A)
np.dot(A, v1)
np.dot(v1, v1)
```

Alternatively, we can cast the array objects to the type `matrix`. This changes the behavior of the standard arithmetic operators `+`, `-`, `*` to use matrix algebra.

```
M = np.matrix(A)
v = np.matrix(v1).T  # Make it a column vector.
M * M
v.T * v, v + M * v
```

Matrix algebra (2/2)

Inverse and determinant

```
np.linalg.inv(M)  # Same as M.I.
M.I * M

np.linalg.det(M)
np.linalg.det(M.I)
```

Data processing

NumPy provides various functions to calculate statistics of datasets in arrays. Here are some examples:

```
np.mean(data[:, 3])
np.std(data[:, 3]), np.var(data[:, 3])
np.min(data[:, 3])
np.max(data[:, 3])

np.sum(d)  # Sum up all elements.
np.prod(d + 1)  # Product of all elements.
np.cumsum(d)  # Cumulative sum.
np.cumprod(d + 1)  # Cumulative product.

np.trace(A)  # Same as: np.diag(A).sum()
```

Calculations with higher-dimensional data

When functions such as `min`, `max`, etc. are applied to multidimensional arrays, it is sometimes useful to apply the calculation to the entire array or on a row or column basis. Using the `axis` argument, we can specify how these functions should behave:

```
M = random.rand(3, 3)

M.max()  # Global max.
M.max(axis=0)  # Max in each column.
M.max(axis=1)  # Max in each row.
```

Many other functions and methods in the `array` and `matrix` classes accept the same (optional) `axis` keyword argument.

Reshaping, resizing, and stacking arrays

The shape of a NumPy array can be modified without copying the underlying data, making it a fast operation even for large arrays:

```
n, m = A.shape
B = A.reshape((1, n * m))

B[0, 0:5] = 5  # Modify the array.
# The original variable is also changed. B is only a different view of the same data.
```

We can also use the function `flatten` to make a higher-dimensional array into a vector. But this function creates a copy of the data:

```
B = A.flatten()
B[0:5] = 10
```

Adding a new dimension: `newaxis`

With `newaxis`, we can insert new dimensions in an array, for example converting a vector to a column or row matrix:

```
v = np.array([1, 2, 3])
np.shape(v)
```

> (3,)

```
# Make a column matrix of the vector v.
v[:, np.newaxis]

# Make a row matrix of the vector v.
v[np.newaxis,:]
```

Stacking and repeating arrays (1/2)

Using functions `repeat`, `tile`, `vstack`, `hstack`, and `concatenate`, we can create larger vectors and matrices from smaller ones:

```
a = np.array([[1, 2], [3, 4]])

# Repeat each element 3 times.
np.repeat(a, 3)
```

> array([1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4])

```
# Tile the matrix 3 times.
np.tile(a, 3)
```

> array([[1, 2, 1, 2, 1, 2], [3, 4, 3, 4, 3, 4]])

Stacking and repeating arrays (2/2)

```
b = np.array([[5, 6]])

# Add a new row.
np.concatenate((a, b), axis=0)
# Same as:
np.vstack((a, b))

# Add a new column.
np.concatenate((a, b.T), axis=1)
# Same as:
np.hstack((a, b.T))
```

Reference vs. deep copy

To achieve high performance, assignments in Python usually do not copy the underlying objects. This is important, for example, when objects are passed between functions to avoid an excessive amount of memory copying when it is not necessary:

```
A = np.array([[1, 2], [3, 4]])

B = A  # B is referring to the same array data as A.
B[0, 0] = 10  # Changing B affects A.

C = np.copy(A)  # Deep copy.
C[0, 0] = -5  # If we modify C, A is not affected.
```

Iterating over array elements (1/2)

Generally, we want to avoid iterating over the elements of arrays whenever possible. The Python `for` loop is the most convenient way to iterate over an array when necessary:

```
v = np.array([1, 2, 3, 4])
for element in v:
    print(element)

M = np.array([[1, 2], [3, 4]])
for row in M:
    print("row", row)
    for element in row:
        print(element)
```

Iterating over array elements (2/2)

When we need to iterate over each element of an array and modify its elements, it is convenient to use the `enumerate` function to obtain both the element and its index in the `for` loop:

```python
for row_idx, row in enumerate(M):
    print("row_idx", row_idx, "row", row)

    for col_idx, element in enumerate(row):
        print("col_idx", col_idx, "element", element)

        # Update the matrix M: square each element.
        M[row_idx, col_idx] = element ** 2
```

Vectorizing functions (1/2)

As mentioned several times, to achieve good performance, we should avoid looping over elements in our vectors and matrices and instead use vectorized algorithms. The first step is to make sure that functions work with vector inputs:

```
def Heaviside(x):
    """
    Scalar implementation of the Heaviside step function.
    """
    if x >= 0:
        return 1
    else:
        return 0

Heaviside(np.array([-3, -2, -1, 0, 1, 2, 3]))
```

> ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()

Vectorizing functions (2/2)

Implement the function to accept a vector input from the beginning:

```python
def Heaviside(x):
    """
    Vector-aware implementation of the Heaviside step function.
    """
    return 1 * (x >= 0)

Heaviside(np.array([-3, -2, -1, 0, 1, 2, 3]))
```

See also the NumPy function `vectorize`.

Using arrays in conditions

When using arrays in conditions, for example `if` statements and other boolean expressions, use `any` or `all`, requiring that any or all elements in the array evaluate to `True`:

```
if (M > 5).any():
    print("At least one element in M is larger than 5.")

if (M > 5).all():
    print("All elements in M are larger than 5.")
```

Type casting

Since NumPy arrays are *statically typed*, the type of an array does not change once created. But we can explicitly cast an array of some type to another using the `astype` functions (see also the similar `asarray` function). This always creates a new array of new type:

```
M.dtype
```

```
dtype('int64')
```

```
M2 = M.astype(float)
M3 = M.astype(bool)
```

Further reading

- NumPy documentation
- NumPy tutorials

SciPy

SciPy overview

SciPy, built on NumPy's foundation, offers higher-level scientific algorithms and modules for specialized tasks like optimization, integration, signal processing, and linear algebra. It provides seamless interoperability with NumPy for efficient data representation and manipulation.

Relationship between NumPy and SciPy:

- NumPy serves as the foundation for numerical operations in SciPy.
- SciPy utilizes NumPy arrays for efficient data representation.
- Seamless interoperability between NumPy and SciPy.
- *NumPy for basic operations, SciPy for specialized tasks*.

SciPy modules

- `scipy.special` for special functions like Bessel or gamma functions.
- `scipy.integrate` for numerical integration and solving differential equations.
- `scipy.optimize` for optimization algorithms.
- `scipy.interpolate` for interpolating functions and data.
- `scipy.fft` for Fourier transforms.
- `scipy.signal` for signal processing tools.
- `scipy.sparse` for sparse matrices and related algorithms.
- `scipy.linalg` for advanced linear algebra operations.
- `scipy.stats` for statistical distributions and functions.
- `scipy.ndimage` for multi-dimensional image processing.
- `scipy.io` for file I/O operations.

Importing SciPy

In this lecture, we will explore how to use some of these subpackages.

To access the SciPy package in a Python program, we start by importing everything from the `scipy` module:

```python
from scipy import *
```

If we only need to use part of the SciPy framework, we can selectively include only those modules we are interested in. For example, to include the linear algebra package under the name `la`, we can do:

```python
import scipy.linalg as la
```

Numerical integration (1/3)

For the numerical evaluation of a definite integral of the type $\int_a^b f(x), \mathrm{d}x$, SciPy provides a series of functions for different kinds of quadrature, such as `quad`, `dblquad`, and `tplquad` for single, double, and triple integrals, respectively.

```python
from scipy.integrate import quad, dblquad, tplquad
```

The `quad` function takes a large number of optional arguments, which can be used to fine-tune the behavior of the function (try `help(quad)` for details).

Numerical integration (2/3)

The basic usage is as follows:

```python
def f(x):
    return x

x_lower = 0
x_upper = 1

val, abserr = quad(f, x_lower, x_upper)
```

For simple functions, we can use a lambda function:

```python
val, abserr = quad(lambda x: exp(-x ** 2), -Inf, Inf)
```

Numerical integration (3/3)

Higher-dimensional integration works in the same way:

```python
def integrand(x, y):
    return exp(-x**2 - y**2)

x_lower = 0
x_upper = 10
y_lower = lambda x: x
y_upper = lambda x: x + 1

val, abserr = dblquad(integrand, x_lower, x_upper, y_lower, y_upper)
```

Ordinary Differential Equations (ODEs) (1/2)

A system of ODEs is usually formulated in standard form before it is attacked numerically. The standard form is

$$y' = f(y, t),$$

where $y = [y_1(t), y_2(t), \ldots, y_n(t)]$, and f is some function that gives the derivatives of the function $y_i(t)$.

To solve an ODE, we need to know the function f and an initial condition, $y(0)$.

Note that higher-order ODEs can always be written in this form by introducing new variables for the intermediate derivatives.

Ordinary Differential Equations (ODEs) (2/2)

SciPy provides two different ways to solve ODEs: an API based on the function `odeint` and an object-oriented API based on the class `ode`. Usually, `odeint` is easier to get started with, but the `ode` class offers some finer level of control. Here we will use the `odeint` functions. For more information about the class `ode`, try `help(ode)`.

To use `odeint`, first import it from the `scipy.integrate` module

```
from scipy.integrate import odeint, ode
```

Once we have defined the Python function `f` and array `y_0`, we can use `odeint` as:

```
y_t = odeint(f, y_0, t)
```

where `t` is an array with time-coordinates for which to solve the ODE problem. `y_t` is an array with one row for each point in time in `t`, where each column corresponds to a solution $y_i(t)$ at that point in time.

Fourier transform

SciPy's `fft` module enables easy computation of Fourier transforms, a critical tool in computational physics, signal processing and data analysis.

```python
from scipy.fft import fft, ifft, fftfreq
import numpy as np

N = 600 # Number of sample points.

T = 1.0 / 800.0 # Sample spacing.

x = np.linspace(0.0, N*T, N, endpoint=False)
y = np.sin(50.0 * 2.0*np.pi*x) + 0.5*np.sin(80.0 * 2.0*np.pi*x)

yf = fft(y)
xf = fftfreq(N, T)[:N//2]
```

Linear algebra

The linear algebra module contains various matrix-related functions, including linear equation solving, eigenvalue solvers, matrix functions (e.g., matrix exponentiation), decompositions (SVD, LU, Cholesky), etc.

Linear systems

```
from scipy.linalg import *

A = array([[1,2,3], [4,5,6], [7,8,9]])
b = array([1,2,3])

x = solve(A, b)
```

Linear algebra

Eigenvalues and eigenvectors

The eigenvalue problem for a matrix A reads $Av_n = \lambda_n v_n$, where v_n is the nth eigenvector and λ_n is the nth eigenvalue.

To calculate eigenvalues of a matrix, use the `eigvals`, and for calculating both eigenvalues and eigenvectors, use the function `eig`:

```
lam = eigvals(A)
lam, v = eig(A)
```

The eigenvectors corresponding to the nth eigenvalue (stored in `lam[n]`) is the nth *column* in `v`, i.e., `v[:,n]`.

There are also more specialized eigensolvers, like the `eigh` for Hermitian matrices.

Linear algebra

Matrix operations

```
inv(A) # Matrix inverse.
det(A) # Matrix determinant.

# Matrix norms of various orders.
norm(A, ord=1)
norm(A, ord=2)
norm(A, ord=Inf)
```

Sparse matrices (1/3)

Sparse matrices are often useful in numerical simulations dealing with large systems, where the problem can be described in matrix form, and matrices or vectors mostly contain zeros. SciPy has good support for sparse matrices, with basic linear algebra operations (e.g., equation solving, eigenvalue calculations, etc.).

There are many possible strategies for storing sparse matrices efficiently, such as coordinate form (COO), list of lists (LIL) form, and compressed-sparse column CSC (and row, CSR). Each format has some advantages and disadvantages. Most computational algorithms (equation solving, matrix-matrix multiplication, etc.) can be efficiently implemented using CSR or CSC formats, but they are not so intuitive and not so easy to initialize. So often, a sparse matrix is initially created in COO or LIL format (where we can efficiently add elements to the sparse matrix data), and then converted to CSC or CSR before used in real calculations.

Sparse matrices (2/3)

When we create a sparse matrix, we have to choose which format it should be stored in. For example,

```python
from scipy.sparse import *

# Dense matrix.
M = array([[1,0,0,0], [0,3,0,0], [0,1,1,0], [1,0,0,1]])

# Convert from dense to sparse.
A = csr_matrix(M)

# Convert from sparse to dense.
A.todense()
```

Sparse matrices (3/3)

More efficient way to create sparse matrices: create an empty matrix and populate it using matrix indexing (avoids creating a potentially large dense matrix).

```
A = lil_matrix((4,4)) # Empty 4x4 sparse matrix.
A[0,0] = 1
A[1,1] = 3
A[2,2] = A[2,1] = 1
A[3,3] = A[3,0] = 1

# Sparse matrix - dense array multiplication.
A * v
```

LIL stands for *LI*sts of *L*ists.

Optimization

Optimization (finding minima or maxima of a function) is a large field in mathematics, and optimization of complicated functions or in many variables can be rather involved.

Finding minima

We can use several algorithms to find the minima of a function:

```python
from scipy import optimize

def f(x):
    return x**4 + 4*x**3 + (x-2)**2

x_min1 = optimize.fmin_bfgs(f, -2)
x_min2 = optimize.brent(f)
x_min3 = optimize.fminbound(f, -4, 2)
```

Optimization

Finding function roots

To find the root for a function of the form $f(x) = 0$, we can use the `fsolve` function. It requires an initial guess:

```python
from scipy import optimize

def f(omega):
    return tan(2 * np.pi * omega) - 3.0 / omega

optimize.fsolve(f, 0.1)
```

Interpolation

The `interp1d` function, when given arrays describing X and Y data, returns an object that behaves like a function that can be called for an arbitrary value of x (in the range covered by X), and it returns the corresponding interpolated y value:

```
from scipy.interpolate import *

def f(x):
    return sin(x)

n = arange(0, 10)
x_meas = linspace(0, 9, 100)
y_meas = f(n) + 0.1 * randn(len(n))

linear_interpolation = interp1d(x_meas, y_meas)
y_interp1 = linear_interpolation(x)

cubic_interpolation = interp1d(x_meas, y_meas, kind='cubic')
y_interp2 = cubic_interpolation(x)
```

Statistics

The `scipy.stats` module contains a large number of statistical distributions, statistical functions, and tests.

```python
from scipy import stats

X = stats.poisson(3.5)
Y = stats.norm()

X.mean(), X.std(), X.var()
Y.mean(), Y.std(), Y.var()
```

Statistical tests

SciPy includes functions for conducting statistical tests, like t-tests and ANOVA, aiding in hypothesis testing and data analysis.

Calculate the T-test for the means of two independent samples:

```
t_statistic, p_value = stats.ttest_ind(X.rvs(size=1000), X.rvs(size=1000))
```

Test if the mean of a single sample of data is 0.1:

```
stats.ttest_1samp(Y.rvs(size=1000), 0.1)
```

A low p-value means that we can reject the hypothesis.

Further reading

- SciPy documentation

Data visualization

Introduction to Matplotlib and Seaborn

Data visualization plays a vital role in data analysis, enabling the effective communication of complex information. Matplotlib and Seaborn are two widely-used Python libraries for data visualization. Matplotlib offers a wide range of plotting tools, while Seaborn provides a high-level interface for drawing attractive statistical graphics.

Matplotlib (1/2)

Matplotlib is a widely-used 2D plotting library in Python. It provides a high-level interface for drawing attractive and informative statistical graphics. Let's start with a simple example to create a basic line plot:

```python
import matplotlib.pyplot as plt
import numpy as np

# Generate data.
x = np.linspace(0, 10, 100)
y = np.sin(x)

# Create a simple line plot.
plt.plot(x, y, label='sin(x)')
plt.title('Simple line plot')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.legend()
plt.show()
```

Matplotlib (2/2)

In this example, we use NumPy to generate data points for the x-axis and calculate corresponding y-values. The `plot` function is then used to create a line plot. Finally, `title`, `xlabel`, `ylabel`, and `legend` functions are used to add a title, axis labels, and a legend to the plot.

Matplotlib supports various types of plots, including scatter plots, bar plots, histograms, and more. Explore the documentation for more plot types and customization options.

2D plots with Matplotlib

Let's create a 2D contour plot using Matplotlib.

```python
import matplotlib.pyplot as plt
import numpy as np

# Generate data.
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))

# Create a 2D contour plot.
plt.contourf(X, Y, Z, cmap='viridis')
plt.colorbar(label='sin(sqrt(x^2 + y^2))')
plt.title('Contour plot')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.show()
```

Seaborn

Seaborn, built on Matplotlib, provides a more intuitive interface for creating statistical plots. It integrates well with pandas data structures and offers built-in themes for enhanced visual appeal.

```python
import seaborn as sns
import numpy as np

# Generate data.
x = np.random.randn(100)
y = 2 * x + np.random.randn(100)

# Create a scatter plot using Seaborn.
sns.scatterplot(x=x, y=y, color='blue')
plt.title('Scatter Plot with Seaborn')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.show()
```

Customizing histograms with Seaborn

Let's create a customized histogram with Seaborn, including specific bin edges, colors, and additional statistical annotations.

```python
import seaborn as sns
import matplotlib.pyplot as plt

# Load a dataset.
tips = sns.load_dataset('tips')

# Create a histogram with customizations.
sns.histplot(tips['total_bill'], bins=[10, 20, 30, 40, 50], color='salmon')
plt.title('Customized histogram')
plt.xlabel('Total bill ($)')
plt.ylabel('Frequency')

# Annotate with mean and median.
plt.axvline(tips['total_bill'].mean(), color='blue', linestyle='dashed', linewidth=2, label='Mean')
plt.axvline(tips['total_bill'].median(), color='green', linestyle='dashed', linewidth=2, label='Median')

plt.legend()
plt.show()
```

Scatter plots with Seaborn

Create a scatter plot with Seaborn that includes a regression line, different colors based on a categorical variable, and markers with varied sizes.

```python
import seaborn as sns
import matplotlib.pyplot as plt

# Load a dataset.
tips = sns.load_dataset('tips')

# Create a scatter plot.
sns.scatterplot(x='total_bill', y='tip', hue='day', size='size', sizes=(20, 200),
                data=tips, palette='Set2', alpha=0.8)
plt.title('Advanced Scatter Plot')
plt.xlabel('Total bill ($)')
plt.ylabel('Tip ($)')
plt.legend(title='Day')
plt.show()
```

Combining Matplotlib and Seaborn

One of the strengths of Seaborn is its ability to work seamlessly with Matplotlib. You can use Matplotlib functions alongside Seaborn to customize your plots further. Here's an example combining Matplotlib and Seaborn to create a histogram with a kernel density estimate:

```python
import seaborn as sns
import matplotlib.pyplot as plt

# Load a dataset.
tips = sns.load_dataset('tips')

# Create a histogram with a Kernel Density Estimate using Seaborn.
sns.histplot(tips['total_bill'], kde=True, color='skyblue')

# Customize Matplotlib features.
plt.title('Histogram with KDE')
plt.xlabel('Total bill ($)')
plt.ylabel('Frequency')
plt.show()
```

Advanced plotting

Matplotlib supports advanced plots like contour plots, 3D plots, and subplots:

- Contour plots for visualizing three-dimensional data.
- Subplots for displaying multiple plots in a single figure.

Seaborn excels in creating complex statistical plots:

- Heatmaps for representing matrix data.
- Pair plots for exploring relationships in a dataset.
- Facet grids for plotting subsets of data on multiple axes.

Matplotlib and Seaborn can directly plot from pandas DataFrame, simplifying the workflow in data analysis tasks.

Further reading

- Matplotlib documentation
- Seaborn documentation

pandas

pandas overview

pandas is a powerful Python library for data manipulation and analysis. It provides fast, flexible data structures like Series and DataFrame, designed to work with structured data intuitively and efficiently.

In the pandas library, the standard import convention involves using the aliases `np` for NumPy and `pd` for pandas:

```
import numpy as np
import pandas as pd
```

Fundamental data structures in pandas

- **Series**: A one-dimensional labeled array that can hold various data types.
- **DataFrame**: A two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns).

Creating objects (1/2)

Series creation

You can create a Series by providing a list of values. pandas will generate a default RangeIndex:

```
s = pd.Series([1, 3, 5, np.nan, 6, 8])
```

DataFrame creation

Creating a DataFrame involves passing a NumPy array with a datetime index and labeled columns:

```
dates = pd.date_range("20130101", periods=6)
df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list("ABCD"))
```

Creating objects (2/2)

DataFrame creation

Alternatively, a DataFrame can be formed from a dictionary of objects:

```
df2 = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20130102"),
        "C": pd.Series(1, index=list(range(4)), dtype="float32"),
        "D": np.array([3] * 4, dtype="int32"),
        "E": pd.Categorical(["test", "train", "test", "train"]),
        "F": "foo",
    }
)
```

The resulting DataFrame has diverse data types.

Viewing data

To examine the top and bottom rows of a DataFrame, use `head()` and `tail()`:

```
df.head()
df.tail(3)
```

Retrieve the DataFrame's index or column labels:

```
df.index
df.columns

df.to_numpy()  # Convert the DataFrame to a NumPy array.
```

Note: NumPy arrays have one dtype for the entire array while pandas DataFrames have one dtype per column. When you call `DataFrame.to_numpy()`, pandas will find the NumPy dtype that can hold *all* of the dtypes in the DataFrame. If the common data type is `object`, `DataFrame.to_numpy` will require copying data.

Data selection (1/3)

pandas offers various methods for data selection. We'll explore both label-based and position-based approaches.

For a **DataFrame**, passing a single label selects a columns and yields a **Series** equivalent to `df.A`:

```
df["A"]
```

Passing a slice `:` selects matching rows:

```
df[0:3]
df["20130102":"20130104"]
```

Data selection (2/3)

Label-based selection

Use `loc` and `at` for label-based indexing:

```
df.loc[dates[0]] # Selecting a row by label.
df.loc[:, ["A", "B"]] # Selecting all rows for specific columns.
df.loc["20130102":"20130104", ["A", "B"]] # Both endpoints are included.
df.at[dates[0], "A"] # Fast scalar access.
```

Position-based selection

For position-based indexing, employ `iloc` and `iat`:

```
df.iloc[3] # Selecting via position.
df.iloc[3:5, 0:2] # Slicing rows and columns.
df.iat[1, 1] # Fast scalar access.
```

Data selection (3/3)

Selecting values from a **DataFrame** where a boolean condition is met:

```
df[df > 0]
```

Select rows based on a condition:

```
df[df["A"] > 0]
```

Use `Series.isin` method for filtering:

```
df2 = df.copy()
df2["E"] = ["one", "one", "two", "three", "four", "three"]
df2[df2["E"].isin(["two", "four"])]
```

Viewing and sorting data

Generate quick statistics using `describe()`:

```
df.describe()
```

Transpose the DataFrame with `.T`:

```
df.T
```

Sort the DataFrame by index or values:

```
df.sort_index(axis=1, ascending=False) # Sort by index.
df.sort_values(by="B") # Sort by values.
```

Operations (1/3)

Statistics

Compute the mean for each column or row:

```
df.mean()
df.mean(axis=1)
```

Operations with Series or DataFrame

Perform operations with another Series or DataFrame:

```
s = pd.Series([1, 3, 5, np.nan, 6, 8], index=dates).shift(2)
df.sub(s, axis="index")
```

Operations (2/3)

User-defined functions

Apply user-defined functions using `agg` and `transform`:

```
# Calculate the mean of each column and then multiply it by 5.6.
# The result will be a pandas Series with the aggregated value for each column in df.
df.agg(lambda x: np.mean(x) * 5.6)

# Apply a function to each element of the DataFrame.
df.transform(lambda x: x * 101.2)
```

Value counts

Compute value counts for a Series:

```
s = pd.Series(np.random.randint(0, 7, size=10))
s.value_counts()
```

Operations (3/3)

pandas simplifies handling missing data using methods like `dropna()`, `fillna()`, and `interpolate()`.

Transform data using operations like pivoting, melting, and applying custom functions.

pandas excels in time series data analysis, offering functionalities for resampling, shifting, and window operations.

pandas supports categorical data types, which can be more efficient and expressive for certain types of data.

Plotting

pandas integrates with Matplotlib for easy data visualization. Here's a basic example:

```python
import matplotlib.pyplot as plt

ts = pd.Series(np.random.randn(1000), index=pd.date_range("1/1/2000", periods=1000))
ts = ts.cumsum()

ts.plot()
plt.show()
```

Importing and exporting data

pandas can read and write to various file formats, including CSV, Excel, SQL, and more.

CSV

```
df = pd.DataFrame(np.random.randint(0, 5, (10, 5)))
df.to_csv("foo.csv")
pd.read_csv("foo.csv")
```

Excel

```
df.to_excel("foo.xlsx", sheet_name="Sheet1")
pd.read_excel("foo.xlsx", "Sheet1", index_col=None, na_values=["NA"])
```

Further reading

- pandas documentation
- 10 minutes to pandas

PyTorch

PyTorch

Quoting the official PyTorch tutorial introduction :

PyTorch is a Python-based scientific computing package serving two broad purposes:

- a replacement for NumPy to use the power of GPUs and other accelerators.
- an automatic differentiation library that is useful to implement neural networks.

The package is imported in Python with `import torch`.

PyTorch: the class `Tensor` (1/2)

Tensors are the PyTorch equivalent to Numpy arrays, with the addition to also have support for GPU acceleration.

In multilinear algebra, tensor is a generalization of vector and matrix concepts:

- a vector is a 1-D tensor
- a matrix a 2-D tensor

  A tensor can simply be initialized calling `torch.Tensor`:

```
x = torch.Tensor(2,3,4)
# tensor([[[ 6.5254e+10,  3.0890e-41,  4.2039e-45, -6.3663e-15],
#          [ 6.5246e+10,  3.0890e-41,  1.6367e-42,  4.5787e-41],
#          [ 6.5254e+10,  3.0890e-41,  4.2039e-45,  4.5787e-41]],
#         [[ 6.5255e+10,  3.0890e-41,  1.4013e-45,  0.0000e+00],
#          [ 6.5255e+10,  3.0890e-41,  6.5246e+10,  3.0890e-41],
#          [ 1.7404e-42, -5.0820e+26,  6.5255e+10,  3.0890e-41]]])
```

PyTorch: the class `Tensor` (2/2)

The function `torch.Tensor` allocates memory for the desired tensor, but reuses any values that have already been in the memory. To directly assign values to the tensor during initialization, there are many alternatives including:

- `torch.zeros`, `torch.ones`, `torch.rand`, `torch.randn`, `torch.arange` ..
- `torch.Tensor`: (input list): Creates a tensor from the list elements you provide

```python
# Create a tensor from a (nested) list
x = torch.Tensor([[1, 2], [3, 4]])
print(x)
# tensor([[1., 2.],
#         [3., 4.]])
```

Tensors can be converted to numpy arrays (`.numpy()`), and numpy arrays back to tensors (`torch.from_numpy()`).

PyTorch: operations with Tensors (1/2)

Most operations that exist in numpy, also exist in PyTorch.

Another common operation aims at changing the shape of a tensor. In PyTorch, this operation is called `view`:

```
x = torch.arange(6)
print("X", x)
#X tensor([0, 1, 2, 3, 4, 5])
x = x.view(2, 3)
print("X", x)
#X tensor([[0, 1, 2],
#          [3, 4, 5]])
```

PyTorch: operations with Tensors (2/2)

Other commonly used operations include matrix multiplications, which are essential for neural networks:

- `torch.matmul`: Performs the matrix product over two tensors, where the specific behavior depends on the dimensions
- `torch.mm`: Performs the matrix product over two tensors, where the specific behavior depends on the dimensions
- `torch.bmm`: Performs the matrix product with a support batch dimension: If the first tensor T is of shape $(b \times n \times m)$, and the second tensor R $(b \times m \times p)$, the output O is of shape $(b \times n \times p)$, and has been calculated by performing b matrix multiplications of the submatrices of T and R: $O_i = T_i @ R_i$
- `torch.einsum`: Performs matrix multiplications and more (i.e. sums of products) using the Einstein summation convention.

PyTorch: Dynamic Computation Graph (1/3)

One of the main reasons for using PyTorch in Deep Learning projects is that we can automatically get gradients/derivatives of functions that we define.

We will mainly use PyTorch for implementing neural networks, and they are just fancy functions. If we use weight matrices in our function that we want to learn, then those are called the **parameters** or simply the **weights**.

Given an input `x`, we define our function by manipulating that input, usually by matrix-multiplications with weight matrices and additions with so-called bias vectors. As we manipulate our input, we are automatically creating a computational graph.

This graph shows how to arrive at our output from our input. PyTorch is a **define-by-run** framework; this means that we can just do our manipulations, and PyTorch will keep track of that graph for us. Thus, we create a dynamic computation graph along the way.

PyTorch: Dynamic Computation Graph (2/3)

Example: let's compute the computational graph of the function

$$y = \frac{1}{\ell(x)} \sum_i [(x_i + 2)^2 + 3]$$

```
x = torch.arange(3, dtype=torch.float32, requires_grad=True) # Only float tensors can have gradients
# X tensor([0., 1., 2.], requires_grad=True)
```

```
a = x + 2
b = a ** 2
c = b + 3
y = c.mean()
print("Y", y)
# Y tensor(12.6667, grad_fn=<MeanBackward0>)
```

We can perform backpropagation on the computation graph by calling the function `backward()` on the last output, which effectively calculates the gradients for each tensor that has the property `requires_grad=True`.

PyTorch: Dynamic Computation Graph (3/3)

Computing `y.backward()`, now `x.grad` will contain the gradient $\frac{\partial y}{\partial x}$:

```python
y.backward()
print(x.grad)
# tensor([1.3333, 2.0000, 2.6667])
```

PyTorch calculates the gradients using the chain rule:

$$\frac{\partial y}{x_i} = \frac{\partial y}{\partial c_i} \frac{\partial c_i}{\partial b_i} \frac{\partial b_i}{\partial a_i} \frac{\partial a_i}{x_i}$$

Ex: try to compute the gradient by hand!

PyTorch: GPU support

A crucial feature of PyTorch is the support of GPUs, short for Graphics Processing Unit. A GPU can perform many thousands of small operations in parallel, making it very well suitable for performing large matrix operations in neural networks.
To check if we have a GPU available:

```
gpu_avail = torch.cuda.is_available()
print(f"Is the GPU available? {gpu_avail}")
# Is the GPU available? True
```

You can write your code with respect to this device object, and it allows you to run the same code on both a CPU-only system, and one with a GPU. Let’s try it below. We can specify the device as follows:

```
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
print("Device", device)
# Device cuda
```

Designing Neural Networks with PyTorch

- PyTorch is designed to build custom neural networks
- The package `torch.nn` makes building neural networks more convenient
- the easiest way to build neural networks is with **Modules**:

```python
class MyModule(nn.Module):

    def __init__(self):
        super().__init__()
        # Some init for my module

    def forward(self, x):
        # Function for performing the calculation of the module.
        pass
```

The `DataLoader` class

The class `torch.utils.data.DataLoader` represents a Python iterable over a dataset with support for automatic batching, multi-process data loading and many more features. The data loader communicates with the dataset using the function `__getitem__`, and stacks its outputs as tensors over the first dimension to form a batch. In contrast to the dataset class, we usually don't have to define our own data loader class, but can just create an object of it with the dataset as input. Additionally, we can configure our data loader with the following input arguments (only a selection, see full list here):

- `batch_size`: Number of samples to stack per batch
- `shuffle`: If True, the data is returned in a random order. This is important during training for introducing stochasticity
- ...

Optimization (1/3)

After defining the model and the dataset, it is time to prepare the optimization of the model. During training, we will perform the following steps:

- Get a batch from the data loader
- Obtain the predictions from the model for the batch
- Calculate the loss based on the difference between predictions and labels
- Backpropagation: calculate the gradients for every parameter with respect to the loss
- Update the parameters of the model in the direction of the gradients

Optimization (2/3): The Loss function

We can calculate the loss for a batch by simply performing a few tensor operations as those are automatically added to the computation graph. For instance, for regression we can use Mean Squared Error (MSE) which is defined as

$$\mathcal{L}_{MSE} = \sum_i (f(x_i) - y_i)^2$$

where $f(x_i)$ is the model prediction given the sample x_i and y_i is the label for x_i.

Optimization (3/3): The optimizer

For updating the parameters, PyTorch provides the package `torch.optim`.
The simplest of them is the Stochastic Gradient Descent (SGD): `torch.optim.SGD. Stochastic Gradient Descent updates parameters by multiplying the gradients with a small constant, called learning rate, and subtracting those from the parameters (hence minimizing the loss).

```
# Input to the optimizer are the parameters of the model: model.parameters()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
```

Now we are ready to train the model (see notebook).

Further reading

PyTorch documentation

UvA DL Notebooks : very nice tutorials from the DL group in Amsterdam

🎉 That's all Folks!