DM587

Scientific Programming

Introduction to Python - Part 3

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

[*Based on booklet Python Essentials*]

Outline

Outline

1. Matplotlib
2. Other Data Visualization Libraries
3. Pandas
4. Miscellaneous

Matplotlib Library

Matplotlib is a low level graph plotting library in Python that serves as a visualization utility.
Most of the Matplotlib utilities lay under the `pyplot` submodule, and are usually imported under the `plt` alias

```
>>> %matplotlib inline
>>> import matplotlib.pyplot as plt
```

In IPython, Jupyter and Jupyter Lab, functions with `%` are extra functions of IPython that add functionalities to the environment. They are called magic functions.

- `%matplotlib inline` shows the plot
- `%matplotlib notebook` shows the plot and provides controls to interact with the plot

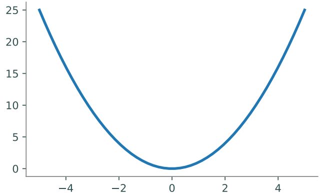
Other useful magic function are:

- `%timeit` to determine running time of a command and
- `%run` to run a script from a file.

Line Plots

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt

>>> y = np.arange(-5,6)**2
>>> y
array([25, 16, 9, 4, 1, 0, 1, 4, 9, 16, 25])

# Visualize the plot.
>>> plt.plot(y)   # default values x-values 0, 1, 2, 3,
[<matplotlib.lines.Line2D object at 0x10842d0>]
>>> plt.show()    # Reveal the resulting plot.
```


```
>>> x = np.linspace(-5, 5, 50)
>>> y = x**2      # Calculate the range of f(x) = x**2.
>>> plt.plot(x,y)
>>> plt.show()
```



Plotting a Polynomial Function

Let's plot the following polynomial of degree 3:

$$P_3(x) = x^3 - 7x + 6 = (x-1)(x-2)(x+3)$$

The `numpy` function `numpy.poly1d` takes an array of coefficients of length n+1 (try `numpy.polyfit?`):

`a[0] * x**n + a[1] * x**(n-1) + ... + a[n-1]*x + a[n]`

```
>>> import numpy as np
>>> a=[1,0,-7,6]
>>> P=np.poly1d(a)
>>> print(P)
   3
1 x - 7 x + 6

>>> x = np.linspace(-3.5, 3.5, 500)
>>> plt.plot(x, P(x), '-')
>>> plt.axhline(y=0)
>>> plt.title('A polynomial of order 3');
```

- `numpy.roots(p)` Return the roots of a polynomial with coefficients given in p.
- `numpy.poly` Find the coefficients of a polynomial with a given sequence of roots.

Multidimensional Root Finding and Optimization

- `numpy` vs `scipy`
- thematical submodules in `scipy`: eg, `scipy.linalg` and `scipy.optimize`
- `scipy.optimize.root_scalar` for scalar functions like `numpy.root`
- `scipy.optimize.root` for multidimensional functions
- `scipy.optimize.minimize` for optimize multimensional (continuous) functions

```
>>> import numpy as np
>>> from scipy.optimize import minimize

>>> def f(x):
        """The Rosenbrock function"""
        return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)
>>> x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
>>> res = minimize(f, x0, method='nelder-mead', options={'xatol':1e-8,'disp':True})
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 339
         Function evaluations: 571
>>> print(res.x) # [1. 1. 1. 1. 1.]
```

Interactive Plotting

In non-interactive mode (deafult):

- newly created figures and changes to figures will not be reflected until explicitly asked to be;
- `.pyplot.show` will block by default.

The interactive mode is mainly useful to build plots from the command line and see the effect of each command while building the figure.
In interactive mode:

- newly created figures will be shown immediately;
- figures will automatically redraw on change;
- `.pyplot.show` will not block by default.

`plt.ion()`	Enable interactive mode
`plt.clf()`	Clear figure
`plt.ioff()`	Disable interactive mode
`plt.show()`	Show all figures (and maybe block)
`plt.pause()`	Show all figures, and block for a time.

Plot Customization

For `plt.plot()`

marker	Style
'.'	point marker
'o'	circle marker
'*'	star marker
'+'	plus marker
...	...

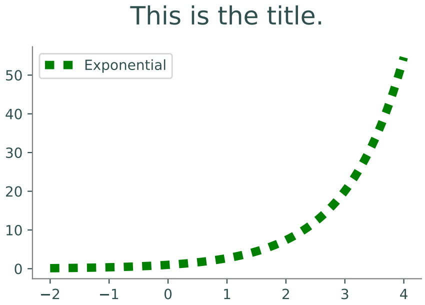
linestyle	Style
'-'	solid line
'- -'	dashed line
'-.'	dash-dot line
':'	dotted line
...	...

color	Color
'b'	blue
'g'	green
'r'	red
'c'	cyan
'k'	black

The parameter `fmt` is written with this syntax `marker|line|color` `markersize` (or `ms`) to set the size of the markers. Other functions

Function	Description
`legend()`	Place a legend in the plot
`title()`	Add a title to the plot
`xlim()` / `ylim()`	Set the limits of the x- or y-axis
`xlabel()` / `ylabel()`	Add a label to the x- or y-axis
`grid()`	Add the grid lines

```
>>> x1 = np.linspace(-2, 4, 100)
>>> plt.plot(x1, np.exp(x1), 'g:', ↪
    ↪linewidth=6, label="Exponential"↪
    ↪)
>>> plt.title("This is the title", ↪
    ↪fontsize=18)
>>> plt.legend(loc="upper left")
>>> plt.show()
```



Layout

Function	Description
`figure()`	Create a new figure or grab an existing figure
`axes()`	Add an axes to the current figure
`gca()`	Get the current axes
`gcf()`	Get the current figure
`subplot()`	Add a single subplot to the current figure
`subplots()`	Create a figure and add several subplots to it

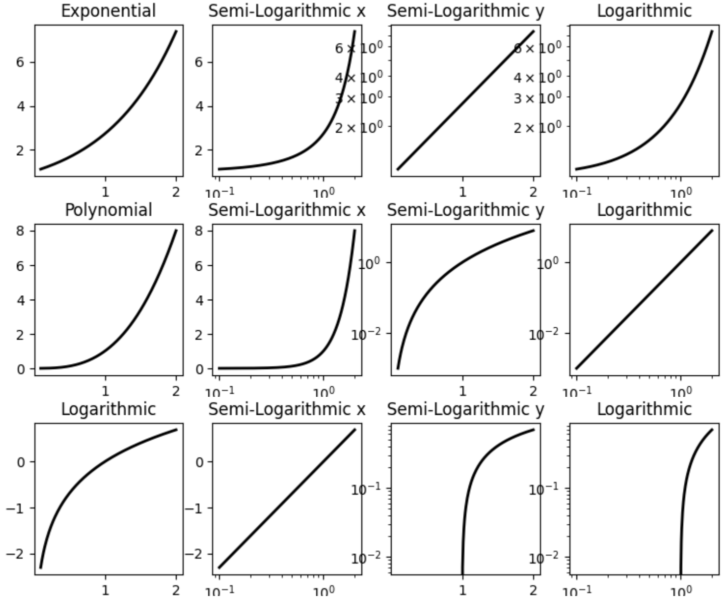
`subplot` takes three arguments: the layout is organized in rows and columns, which are represented by the first and second argument. The third argument represents the index of the current plot.

```
# 3. Use plt.subplots() to get the figure and all subplots simultaneously.
>>> fig, axes = plt.subplots(1, 2)
>>> axes[0].plot(x, 2*x)
>>> axes[1].plot(x, x**2)
```

Compare `axes()` vs `axis()` (access properties of the current plot)

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

def make_figure(x,f,name):
    plt.figure(figsize=(9,2))
    ax1 = plt.subplot(141)
    ax1.plot(x, f(x), 'k', lw=2)
    plt.title(name)

    ax2 = plt.subplot(1,4,2)
    ax2.semilogx(x, f(x), 'k', lw=2)
    ax2.set_title("Semi-Logarithmic x")

    ax3 = plt.subplot(143)
    ax3.semilogy(x, f(x), 'k', lw=2)
    ax3.set_title("Semi-Logarithmic y")

    ax4 = plt.subplot(144)
    ax4.loglog(x, f(x), 'k', lw=2)
    ax4.set_title("Logarithmic")
    plt.savefig(name+".png")
```

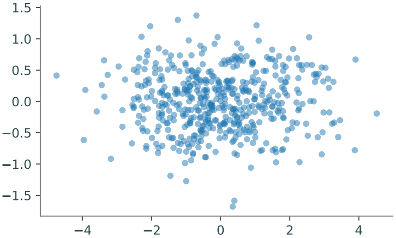
```python
xx = np.linspace(.1, 2, 200)
make_figure(xx, lambda xx: np.exp(xx), "Exponential")
make_figure(xx, lambda xx: xx**3, "Polynomial")
make_figure(xx, lambda xx: np.log(xx), "Logarithmic")
```

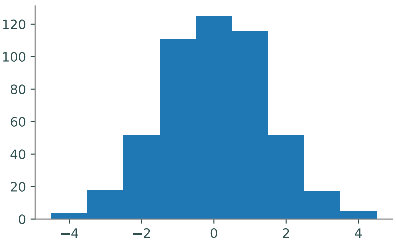


Exponential
Semi-Logarithmic x
Semi-Logarithmic y
Logarithmic
Polynomial
Semi-Logarithmic x
Semi-Logarithmic y
Logarithmic
Logarithmic
Semi-Logarithmic x
Semi-Logarithmic y
Logarithmic

Scatter Plots and Histograms

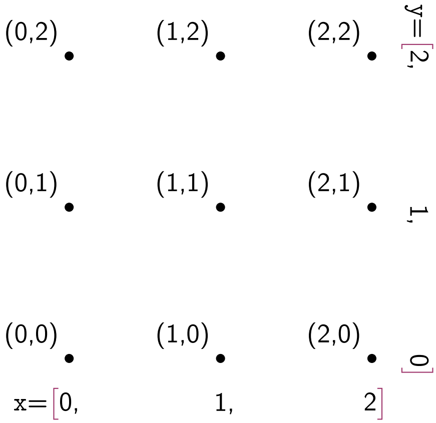
```
>>> x = np.random.normal(scale=1.5, size=500)
>>> y = np.random.normal(scale=0.5, size=500)

>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, y, 'o', markersize=5, alpha=.5) # transparent circles

>>> ax2 = plt.subplot(122)
>>> ax2.hist(x, bins=np.arange(-4.5, 5.5))
>>> plt.show()
```





3D Surfaces

`np.meshgrid()` given two 1-dimensional coordinate arrays, creates two corresponding coordinate matrices: `(X[i,j], Y[i,j]) = (x[i],y[j])`.



$$X = \begin{vmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{vmatrix}$$

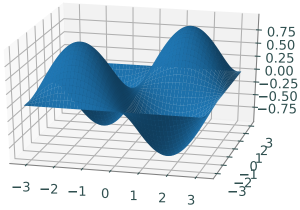
$$Y = \begin{vmatrix} 2 & 2 & 2 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{vmatrix}$$

```
>>> x, y = [0, 1, 2], [3, 4, 5]      # A rough domain over [0,2]x[3,5].
>>> X, Y = np.meshgrid(x, y)         # Combine the 1-D data into 2-D data.
>>> for trows in zip(X,Y):
...     print(trows)
...
(array([0 1 2]), array([3 3 3]))
(array([0 1 2]), array([4 4 4]))
(array([0 1 2]), array([5 5 5]))
```

3D Surfaces

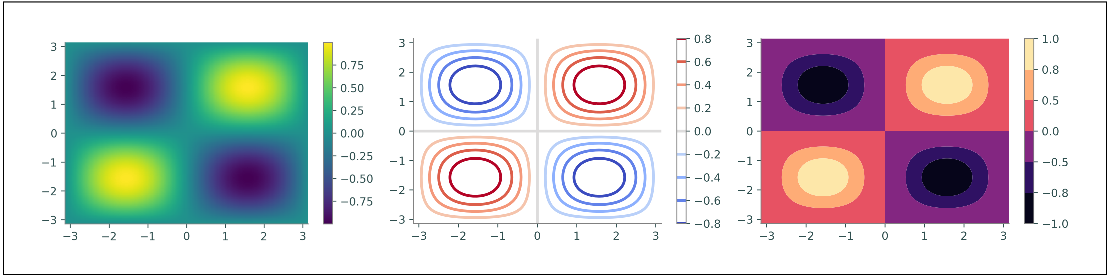
$$g(x, y) = \sin(x)\sin(y)$$

```
>>> x = np.linspace(-np.pi, np.pi, 200)
>>> y = np.copy(x)
>>> X, Y = np.meshgrid(x, y)
>>> Z = np.sin(X) * np.sin(Y)

# Draw the corresponding 3-D plot using some ↪
    ↪extra tools.
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = plt.figure()
>>> ax = fig.add_subplot(1,1,1, projection='3d')
>>> ax.plot_surface(X, Y, Z)
>>> plt.show()
```



Heat Map and Contour Plot

```
>>> x = np.linspace(-np.pi, np.pi, 100)
>>> y = x.copy()
>>> X, Y = np.meshgrid(x, y)
>>> Z = np.sin(X) * np.sin(Y)       # Calculate g(x,y) = sin(x)sin(y).
# Plot the heat map of f over the 2-D domain.
>>> plt.subplot(131)
>>> plt.pcolormesh(X, Y, Z, cmap="viridis")
>>> plt.colorbar()
>>> plt.xlim(-np.pi, np.pi)
>>> plt.ylim(-np.pi, np.pi)
# Plot a contour map of f with 10 level curves.
>>> plt.subplot(132)
>>> plt.contour(X, Y, Z, 10, cmap="coolwarm")
>>> plt.colorbar()
# Plot a filled contour map, specifying the level curves.
>>> plt.subplot(133)
>>> plt.contourf(X, Y, Z, [-1, -.8, -.5, 0, .5, .8, 1], cmap="magma")
>>> plt.colorbar()
>>> plt.show()
```


3
2
1
0
−1
−2
−3
−3 −2 −1 0 1 2 3
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
1.0
0.8
0.5
0.0
−0.5
−0.8
−1.0

Animations

1. Calculate all data that is needed for the animation.
2. Define a figure explicitly with `plt.figure()` and set its window boundaries.
3. Draw empty objects that can be altered dynamically.
4. Define a function to update the drawing objects.
5. Use `matplotlib.animation.FuncAnimation()`.

```python
from matplotlib.animation import FuncAnimation
from mpl_toolkits.mplot3d import Axes3D

def sine_animation():
    # 1. Calculate the data to be animated.
    x = np.linspace(0, 2*np.pi, 200)[:-1]
    y = np.sin(x)    #
    # 2. Create a figure and set the window boundaries of the axes.
    fig = plt.figure()
    plt.xlim(0, 2*np.pi)
    plt.ylim(-1.2, 1.2) #
    # 3. Draw an empty line. The comma after 'drawing' is crucial.
    drawing, = plt.plot([],[])  #
    # 4. Define a function that updates the line data.
    def update(index):
        drawing.set_data(x[:index], y[:index])
        return drawing,                     # Note the comma!
    # 5.
    a = FuncAnimation(fig, update, frames=len(x), interval=10)
```

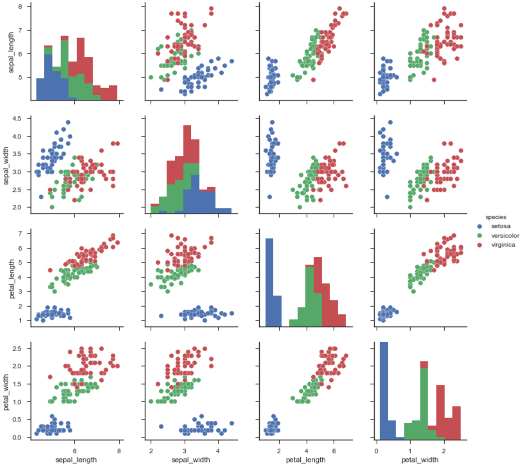
Further Reading and Tutorials

- https://www.w3schools.com/python/matplotlib_intro.asp
- https://www.labri.fr/perso/nrougier/teaching/matplotlib/.
- https://matplotlib.org/users/pyplot_tutorial.html.
- http://www.scipy-lectures.org/intro/matplotlib/matplotlib.html.
- https://matplotlib.org/2.0.0/examples/animation/index.html

Outline

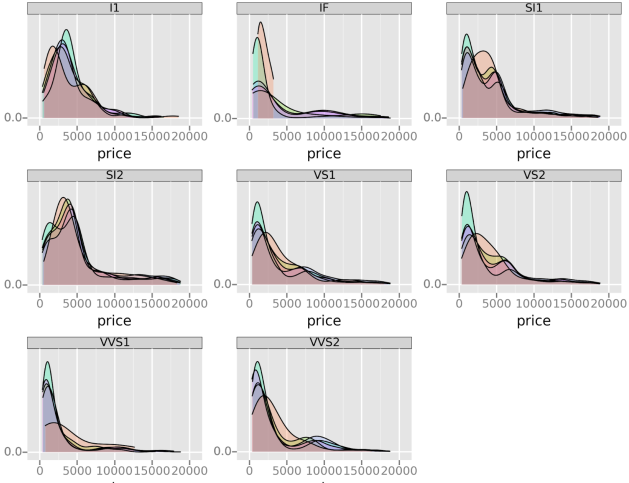
Seaborn

A high-level library on the top of Matplotlib. It's easier to generate certain kinds of plots: eg, heat maps, time series, and violin plots.



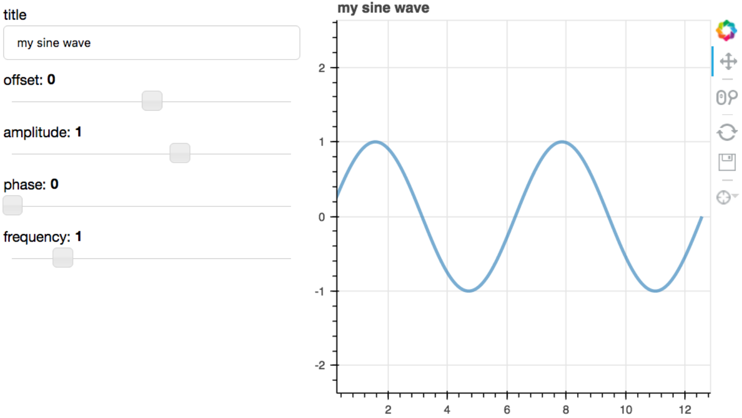
ggplot

Based on R's ggplot2 and the Grammar of Graphics



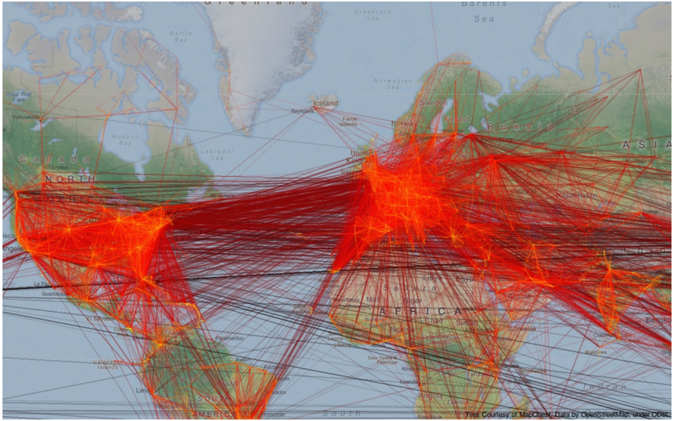
Bokeh, Plotly, Gleam, Dash and Altair

Create interactive, web-ready plots, as JSON objects, HTML documents, or interactive web applications.
Bokeh is based on the Grammar of Graphics like ggplot.
Gleam is inspired by R's Shiny package.



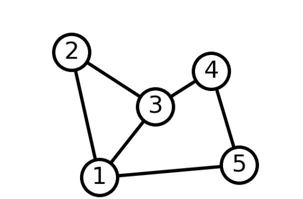
Geoplotlib, Leaflet and MapBox

Toolbox for plotting geographical data: map-type plots, like choropleths, heatmaps, and dot density maps.



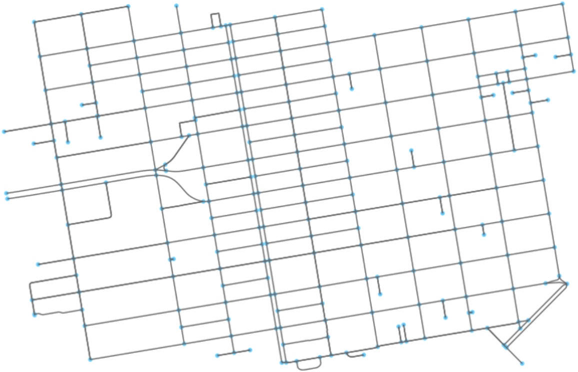
Graph Algorithms, Graph Drawings

```python
import networkx as nx
import matplotlib.pyplot as plt

G = nx.Graph()
G.add_edge(1, 2)
G.add_edge(1, 3)
G.add_edge(1, 5)
G.add_edge(2, 3)
G.add_edge(3, 4)
G.add_edge(4, 5)

# explicitly set positions
pos = {1: (0, 0), 2: (-1, 0.3), 3: (2, 0.17), 4: (4, 0.255), 5: (5, 0.03)}

nx.draw_networkx(G, pos)

# Set margins for the axes so that nodes aren't clipped
ax = plt.gca()
ax.margins(0.20)
plt.axis("off")
plt.show()
```



Street Networks

```python
import osmnx as ox
G = ox.graph_from_bbox(37.79, 37.78, -122.41, -122.43, network_type='drive')
G_projected = ox.project_graph(G)
ox.plot_graph(G_projected)
```



Outline

Pandas Data Structures: Series

Pandas library for data management and analysis that combines functionalities of NumPy, MatPlotLib, and SQL

- `Series` is a one-dimensional array that can hold any datatype, similar to a `ndarray` but with an `index` that gives a label to each entry.

```
>>> import pandas as pd
>>>
>>> # Initialize Series of student grades
>>> math = pd.Series(np.random.randint(0,100,4), ['Mark', 'Barbara', 'Eleanor',↪
    ↪ 'David'])
>>> english = pd.Series(np.random.randint(0,100,5), ['Mark', 'Barbara','David',↪
    ↪ 'Greg', 'Lauren'])
>>> math
Mark       30
Barbara    71
Eleanor    94
David      41
dtype: int64
```

Pandas Data Structures: Data Frames

- `DataFrame` is a collection of multiple `Series`. It can be thought of as a 2-dimensional array, a 2-dimensional tabular data structure that resembles a spreadsheet or an SQL table, where each row is a separate datapoint and each column is a feature of the data. The rows are labelled with an `index` (as in a `Series`) and the columns are labelled in the attribute `columns`.

```
>>> # Create a DataFrame of student grades
>>> grades = pd.DataFrame({"Math": math, "English": english})
>>> grades
          Math  English
Barbara   52.0     73.0
David     10.0     39.0
Eleanor   35.0      NaN
Greg       NaN     26.0
Lauren     NaN     99.0
Mark      81.0     68.0
```

Reading from files:

```
>>> # Creating a DataFrame from a CSV file
>>> df = pd.read_csv('data.csv')

>>> df2 = pd.read_excel(``data.xlsx'', keep_default_na=True,
sheet_name='sheet1', header=0, index_col=None, names=names, dtype={'a':
  str,'b': object})
# alternatively use openpyxl
```

Viewing data:

```
>>> # Display first few rows of DataFrame
>>> print(df.head())

>>> # Display random samples of data
>>> print(df.sample(2))

>>> # Display summary statistics
>>> print(df.describe())
```

Indexing and selecting data

```
# Select a single column
age_column = df['Age']

# Select multiple columns
subset = df[['Name', 'Age']]

# Select rows using row indices (labels)
row = df.loc[1]  # second row

# Select rows and columns simultaneously by labels
value = df.loc[0, 'Name']

# Select by position
value = df.iloc[0, 1]

# Slicing the first four rows (otherwise do this via .iloc)
rows = df[:3]
```

Converting between Pandas and built-in:

1. Convert to a list of lists

```
list_of_lists = grades.values.tolist()
#  [[7.0, 23.0], [29.0, 82.0], [6.0, nan], [nan, 97.0], [nan, 29.0], [77.0, 46.0]]
```

2. Convert to a dictionary (column names as keys)

```
dict_of_columns = grades.to_dict()
# {'Math': {'Barbara': 7.0, 'David': 29.0, 'Eleanor': 6.0,...}, 'English': {...}
```

3. Convert to a list of dictionaries (each row as a dictionary)

```
list_of_dicts = grades.to_dict(orient='records')
# [{'Math': 7.0, 'English': 23.0}, {'Math': 29.0, 'English': 82.0}, ...]
```

4. Convert to a list of tuples (each row as a tuple)

```
list_of_tuples = [tuple(x) for x in grades.values]
# [(7.0, 23.0), (29.0, 82.0), (6.0, nan), (nan, 97.0), (nan, 29.0), (77.0, 46.0)]
```

5. Convert from built-in to data frame

```
pd.DataFrame.from_dict(list_of_dicts, orient='columns')
```

Filtering Data:

```
# Filter rows where Age is greater than 30
filtered_df = df[df['Age'] > 30]

# Combining conditions
filtered_df = df[(df['Age'] > 30) & (df['Name'] != 'Charlie')]
```

Adding and Modifying Data:

```
# Add a new column
df['City'] = ['New York', 'San Francisco', 'Los Angeles']

# Modify values in a column
df.loc[0, 'Age'] = 26
```

Grouping and Aggregating:

```
# Group by 'City' and calculate the average age in each city
city_groups = df.groupby('City')
city_averages = city_groups['Age'].mean()
```

Sorting Data:

```
# Sort by Age in descending order
df.sort_values(by='Age', ascending=False, inplace=True)
```

Dealing with Missing Data:

```
# Drop rows with missing values
df.dropna()

# Fill missing values with a specified value
df.fillna(0)
```

Summary

Outline

The `assert` statement

Assert statements are a convenient way to insert debugging assertions into a program:

```
assert_stmt ::=  "assert" expression ["," expression]
```

The simple form, assert expression, is equivalent to

```
if __debug__:
    if not expression: raise AssertionError
```

The extended form, `assert expression1, expression2`, is equivalent to

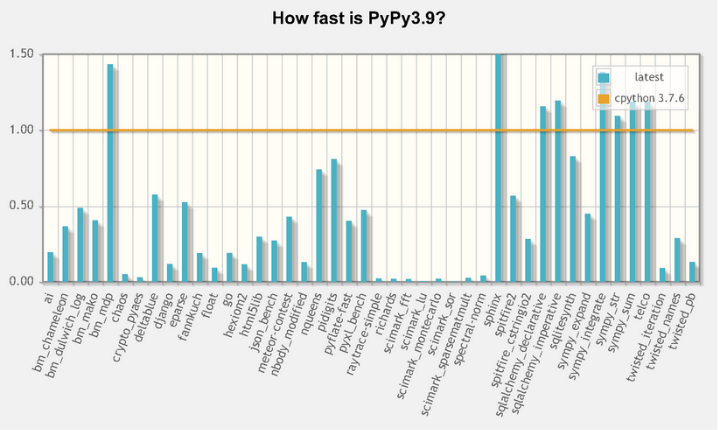
```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

- `__debug__` and `AssertionError` refer to the built-in variables with those names
- Assignments to `__debug__` are illegal. Value determined when the interpreter starts
- `__debug__` is normally `True`, `False` when optimization is requested (command line option `-O`)
- Code generator emits no code for an assert when `-O`

Python implementations

- CPython is the default and most widely used implementation of the Python language.
- CPython can be defined as both an interpreter and a compiler as it compiles Python code into bytecode before interpreting
- PyPy is a Just-in-Time compiler for Python programs

On average, PyPy is **4.8 times faster** than CPython 3.7. We currently support python 3.10, 3.9, and 2.7.



PyPy (with JIT) benchmark times normalized to CPython. Smaller is better. Based on the geometric average of all benchmarks

The `pass` statement

```
pass_stmt ::=  "pass"
```

`pass` is a null operation – when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed, for example:

```
def f(arg): pass    # a function that does nothing (yet)

class C: pass       # a class with no methods (yet)
```

Parsing arguments

As of Python 2.7, optparse is deprecated. The standard now is `argparse`

```
import argparse
def main():
    parser = argparse.ArgumentParser()
    # Positional arguments
    parser.add_argument("input_directory", help="The directory of the input data", type=str, required
        =True)
    # Optional arguments
    parser = argparse.ArgumentParser(description='Smth', epilog='Example')
    parser.add_argument('-s','--source', metavar='SourceData', dest='source', type=str, action='store
        ', nargs=1, required=True, help='smth')

    group = parser.add_mutually_exclusive_group()
    group.add_argument('--iscritti', action='store_true')
    group.add_argument('--maleFemaleSplit', action='store_true')

    args = parser.parse_args()
    return args

if __name__ == "__main__":
    main()
```

https://docs.python.org/3/howto/argparse.html

Handling paths

- For low-level path manipulation on strings, you can use the `os.path` module.
- But since Python 3.4 `pathlib` is the more modern way.
- It doesn't make much difference for joining paths, but other path commands are more convenient with pathlib compared to os.path. For example, to get the "stem" (filename without extension):

```
os.path: splitext(basename(path))[0]

pathlib: path.stem
```

Pickle

- `pickle` refers to the process of serializing and deserializing Python objects.
- Serialization is the process of converting a Python object into a byte stream, while deserialization is the process of reconstructing a Python object from that byte stream.
- Pickle is a built-in module in Python that makes it easy to achieve this.

```python
import pickle

# Define a Python object (in this case, a simple dictionary)
data = {'name': 'Alice', 'age': 30, 'city': 'New York'}

# Pickle the data and write it to a file
with open('data.pkl', 'wb') as file:
    pickle.dump(data, file)

# Now, let's read the pickled data from the file and deserialize it
with open('data.pkl', 'rb') as file:
    loaded_data = pickle.load(file)

# Display the deserialized data
    print(loaded_data)
```

Subprocess and Threading

- The `subprocess` module in Python allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. It makes it possible to interact with external programs or system commands from within your Python script.

- The `threading` module in Python allows you to work with threads for concurrent execution. Threads are lightweight sub-processes that run concurrently within a process.

- Note: Python's Global Interpreter Lock (GIL) can limit the true parallel execution of threads in some scenarios, particularly in CPU-bound tasks. For CPU-bound tasks, you might consider using the `multiprocessing` module, which uses separate processes instead of threads to take full advantage of multiple CPU cores.

```python
import threading
import time

# Function that will be executed by each thread
def print_numbers():
    for i in range(1, 6):
        print(f"Number {i}")
        time.sleep(1)

def print_letters():
    for letter in 'abcde':
        print(f"Letter {letter}")
        time.sleep(1)

# Create two threads
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)

# Start the threads
thread1.start()
thread2.start()

# Wait for both threads to finish
thread1.join()
thread2.join()

print("Both threads have finished.")
```

```python
import subprocess

# Run a simple command and capture its output
result = subprocess.run(['ls', '-l'], stdout=subprocess.PIPE, text=True)

# Print the command's output
print("Command Output:")
print(result.stdout)

# Run a command with arguments
file_name = "example.txt"
result = subprocess.run(['touch', file_name])

# Check the return code of the command
if result.returncode == 0:
    print(f"Command 'touch' successfully created {file_name}")
else:
    print(f"Command 'touch' failed")

# Run a command with input
text = "Hello, World!"
result = subprocess.run(['echo', text], stdout=subprocess.PIPE, text=True)

# Print the output of the 'echo' command
print("Echo Command Output:")
print(result.stdout)
```

And more...

- flask, django for dynamic web pages and web apps.
- beautifulsoup for web scraping
- scikitlearn, tensor flow, pytorch, keras for Machine Learning
- nltk for preprocessing natural language
- ...

Tensors

- Tensors are multi-dimensional arrays, and they are the fundamental data structures in libraries like PyTorch and TensorFlow.
- similar to NumPy arrays but are designed to work efficiently with GPUs, making them faster for large-scale computations.
- All tensors are immutable like Python numbers and strings: you can never update the contents of a tensor, only create a new one.

```python
import torch

# Create a 1D tensor (vector)
tensor_1d = torch.tensor([1, 2, 3, 4])
print(tensor_1d)

# Create a 2D tensor (matrix)
tensor_2d = torch.tensor([[1, 2], [3, 4], [5, 6]])
print(tensor_2d)

# Create a 3D tensor
tensor_3d = torch.tensor([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(tensor_3d)
```

```
import numpy as np

# Convert a NumPy array to a PyTorch tensor
numpy_array = np.array([1, 2, 3])
tensor_from_numpy = torch.from_numpy(numpy_array)
print(tensor_from_numpy)

# Convert a PyTorch tensor to a NumPy array
numpy_from_tensor = tensor_from_numpy.numpy()
print(numpy_from_tensor)
```