プログラミング実習I クラス5 (井村担当)

知能・機械工学課程 井村 誠孝

m.imura@kwansei.ac.jp

本日の内容

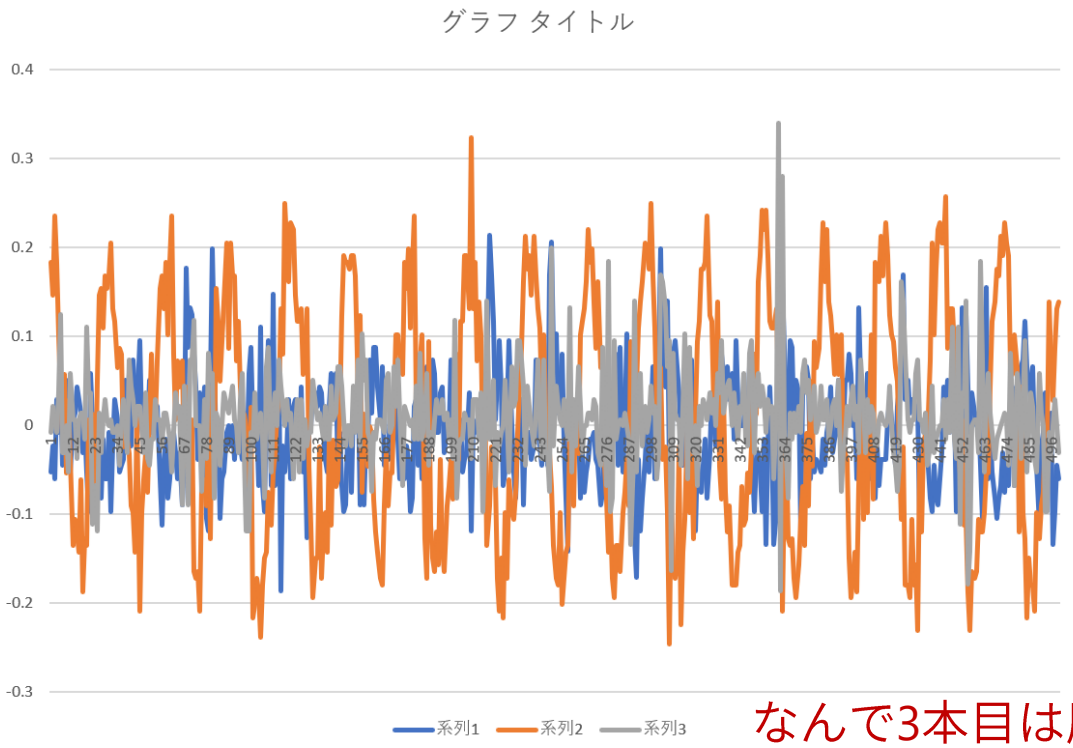
- 大学での学習・調査・研究に有用なPythonのパッケージを使う
- グラフを描く Matplotlib
- 数値計算をする NumPy

- 参考: 他にも様々なものがある.
  - 科学技術計算: SciPy
  - 統計処理: Pandas
  - 機械学習: scikit-learn

Matplotlib

グラフ? Excelで描けばよいのでは?

- Excelは表計算ソフトウェアであって，グラフ描画ソフトウェアではありません．
  - デフォルトだとなかなか突っ込みどころのあるグラフができあがる．
  - データが更新されるごとに手作業でグラフを作るのは時間の無駄．



なぜ横線のみ?

線が太すぎて細かいところがわからない

なんで目盛りの間隔が11?

なんで3本目は灰色なん?

グラフを描くための準備

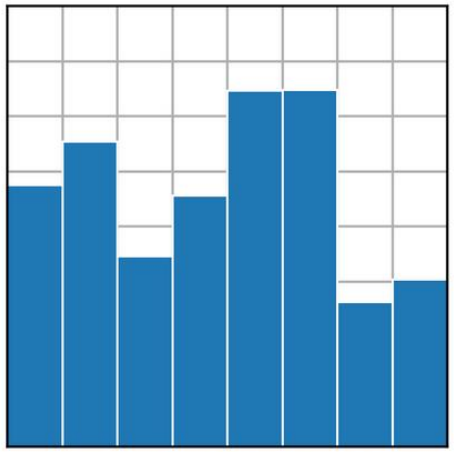
- 必要なモジュールをimport
  - グラフを描くためのモジュール Matplotlib
    - matplotlib.pyplot を plt と名付けて使う
  - 数値演算のためのモジュール NumPy
    - np と名付けて使う
- 定型的な書き方

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

グラフ描画パッケージ Matplotlib

matplotlib

Plot types Examples Tutorials Reference User guide Develop Release notes



bar(x, height) / barh(y, width)

Matplotlib: Visualization with Python

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.
Matplotlib makes easy things easy and hard things possible.

- Create publication quality plots.
- Make interactive figures that can zoom, pan, update.
- Customize visual style and layout.
- Export to many file formats.
- Embed in JupyterLab and Graphical User Interfaces.
- Use a rich array of third-party packages built on Matplotlib.

Try Matplotlib (on Binder) →

Getting Started

Exam

at Sheets

Documentation

https://matplotlib.org/
非常に充実しています

データの準備

- グラフを描くためには，描く対象となるデータが必要
- データの準備方法
  - プログラム内で生成する
    - 例: 式がわかっている関数のグラフを描く
  - データをファイルから読み込む
    - 例: センサによる計測値のグラフを描く
- データの型
  - リスト型
  - NumPyのndarray型

実践的にはこちらを使います
(実習の後半で使うことになる)

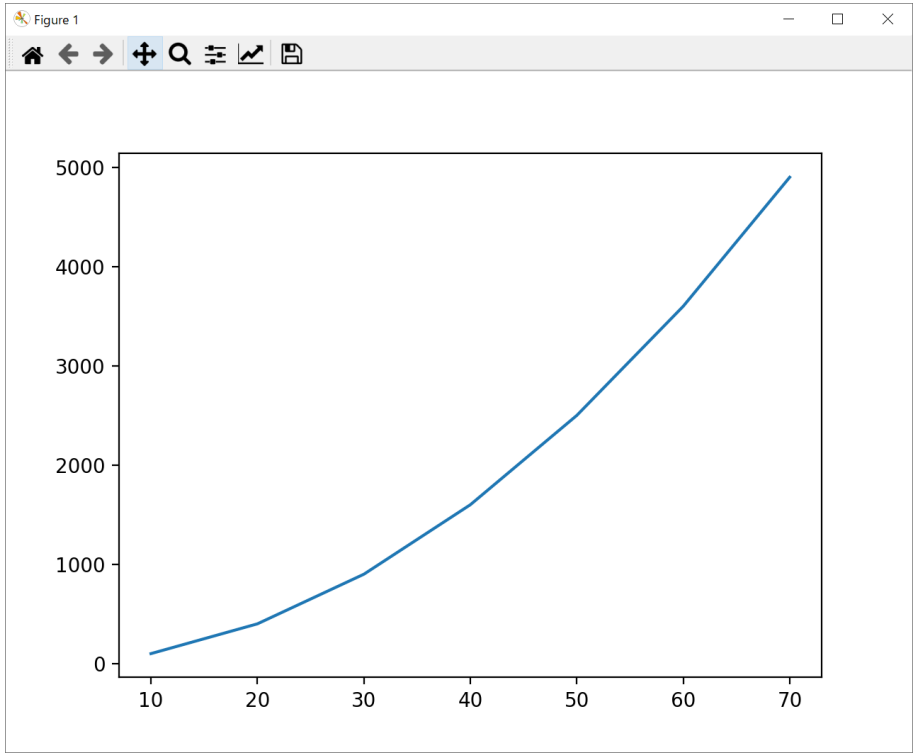
最も簡単なグラフの例: plot

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

x = [10, 20, 30, 40, 50, 60, 70]
y = [100, 400, 900, 1600, 2500, 3600, 4900]

plt.plot(x, y)

plt.show()
```

データの準備

折れ線グラフを描くメソッド plot()

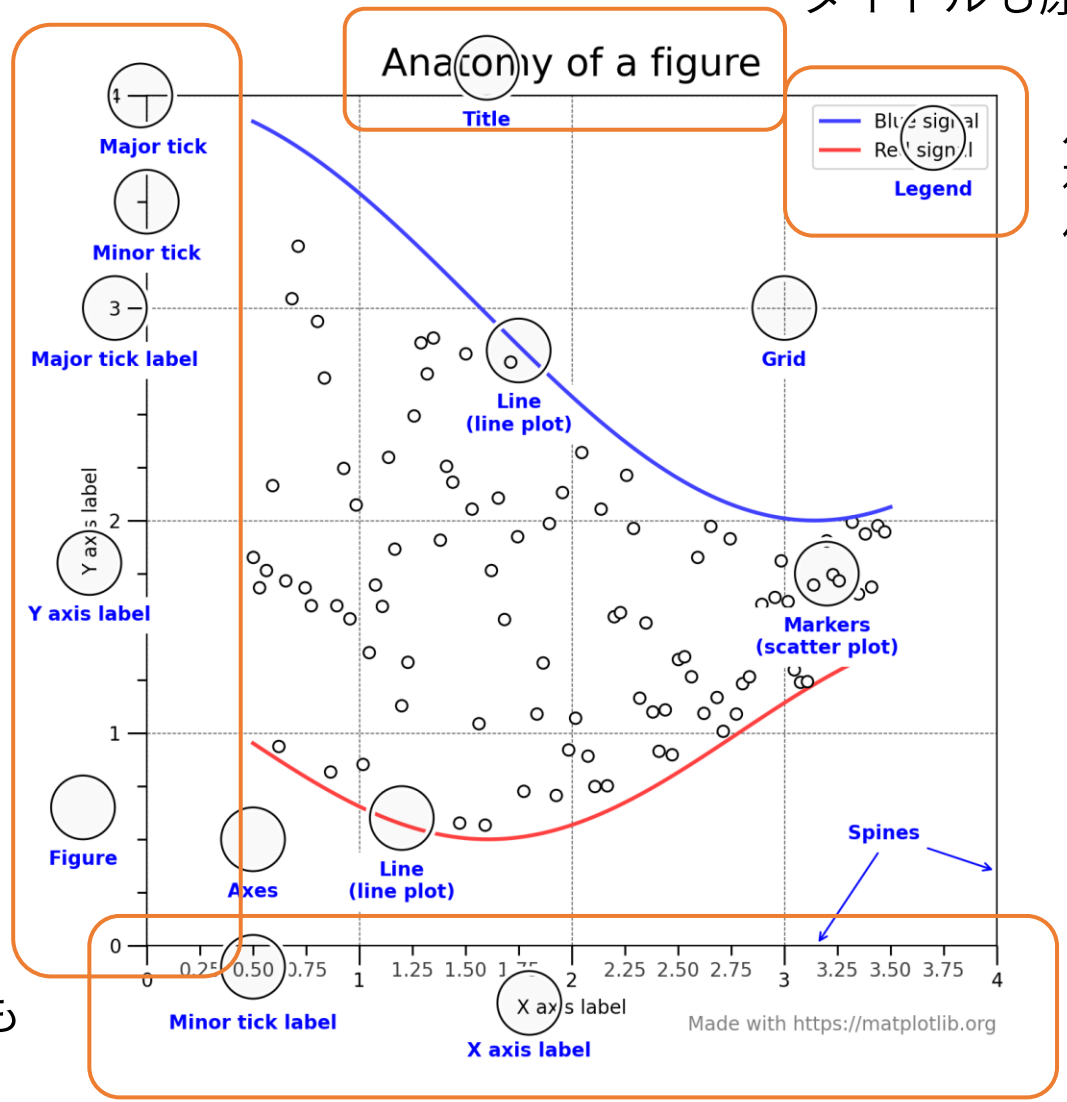
表示する



横軸・縦軸が何かわからんし
タイトルも無い

グラフを構成する要素

タイトルも原則として付ける



凡例 (はんれい)
複数のプロットがある場合は
何が何なのかわかるようにする

軸が何であるかは必ず示す
単位がある場合は単位を明記
目盛りをどのように振るかにも
気を配る

様々な要素の入った例

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

x = [10, 20, 30, 40, 50, 60, 70]
y1 = [100, 400, 900, 1600, 2500, 3600, 4900]
y2 = [200, 600, 1200, 2000, 3000, 4200, 5600]

plt.plot(x, y1, marker='o', color='red', linestyle='--')
plt.plot(x, y2, marker='v', color='blue', linestyle=':')

plt.axis([0, 100, 0, 6000])
plt.grid(True)
plt.title('matplotlib test1')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(['y1', 'y2'])

plt.show()
```

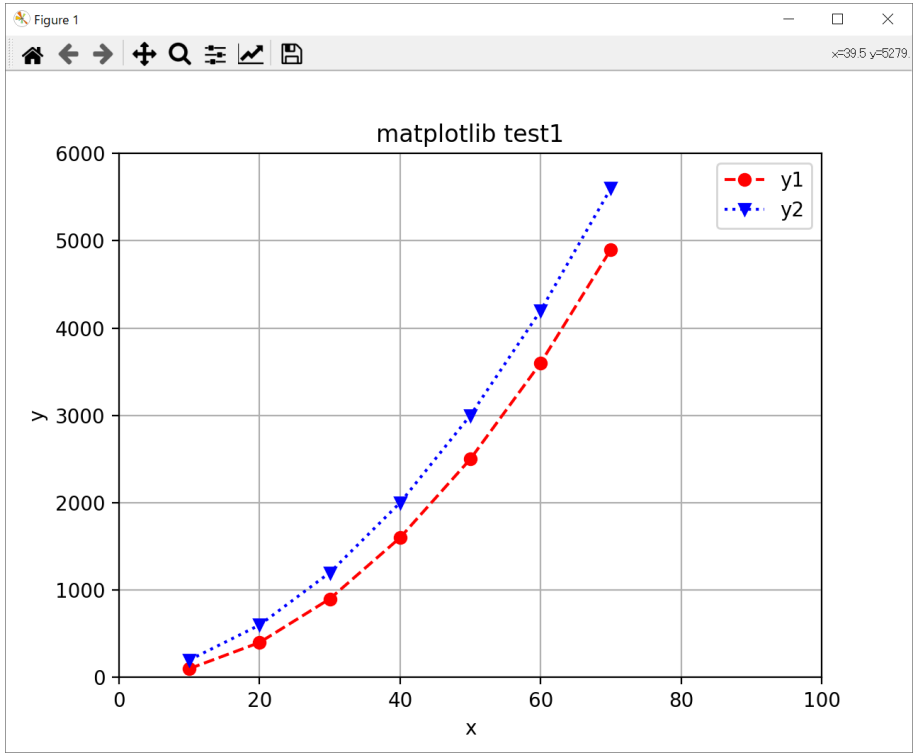
データをリストで用意
要素数は縦横で同じ

スタイルは後述

複数回plot()を実行すると
同じ領域に重ね描きされる

x軸の最小，最大，
y軸の最小，最大 の順

これは凡例
文字列であることに注意



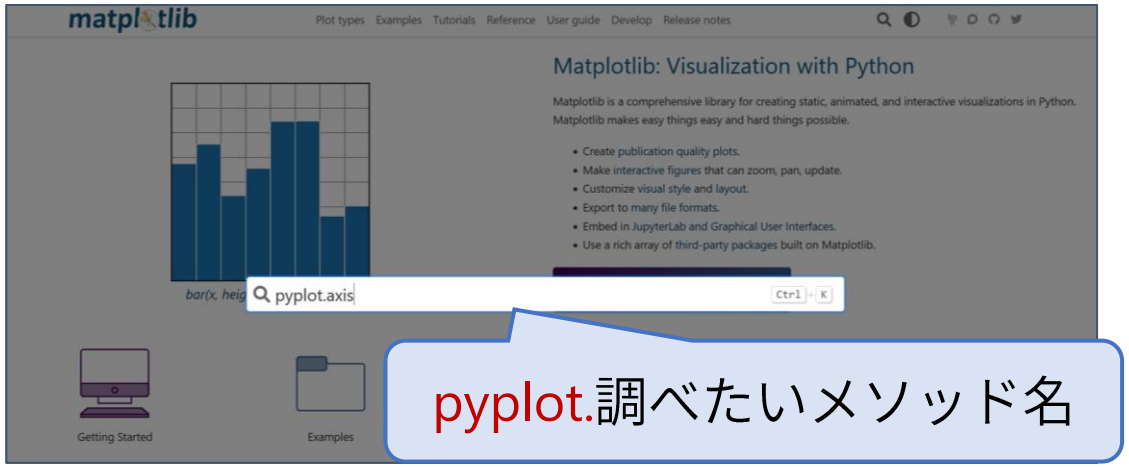
いろいろ調整したい(1)

- 軸の名前: xlabel(), ylabel()
- タイトル: title()
- 描画範囲: axis() (軸ごとに指定したい場合は xlim(), ylim())
- 軸の目盛り: xticks(), yticks()
- x軸・y軸を描画: axhline(), axvline()
- 格子を描画: grid()
- 対数軸: xscale(), yscale()
  - 'linear': 線形
  - 'log': 対数

いずれも matplotlib.pyplot の関数
他にもいろいろある

公式サイトで調べよう

https://matplotlib.org/stable/index.html



pyplot.調べたいメソッド名

Search

Searching for multiple words only shows matches that contain all words.

If you want to limit your search to working code examples, include the keyword "codex" (mnemonic for *code example*) in your search, e.g., "codex ellipse".

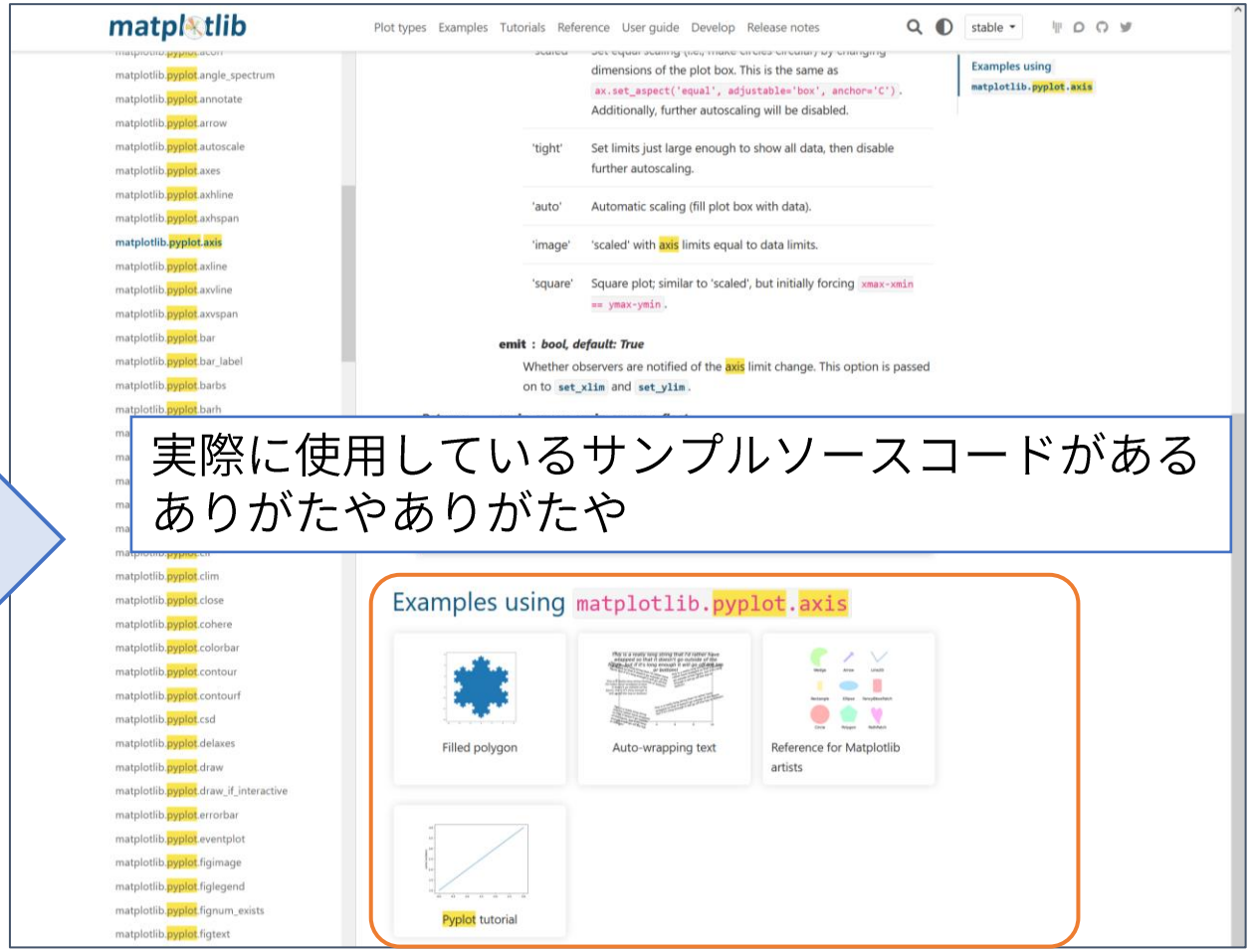
pyplot.axis search

Search Results

Search finished, found 288 page(s) matching the search query.

- matplotlib.pyplot.axis
  matplotlib.pyplot.axis(*args, emit=True, **kwargs)[source] Convenience method to get or set some axis properties. Call signatures: xmin, xmax, ymin, ymax = axis() xmin, xmax, ymin, ymax = axis([xmin, xmax, ymin, y...

matplotlib.pyplot.axis の説明...ページの下の方に行く



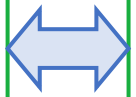
実際に使用しているサンプルソースコードがある
ありがたやありがたや

いろいろ調整したい(2)

- マーカー，線，色，その他
  - plot()のキーワード引数で指定
    - マーカー: marker
    - 色: color (省略形 c)
    - 線のスタイル: linestyle (省略形 ls)
    - 線幅: linewidth (省略形 lw)
    - マーカーサイズ: markersize (省略形 ms)
    - 凡例用のラベル: label　個々のプロットに対して指定することもできる

簡略化したフォーマット文字列でも指定可能

```
plt.plot(x, y1, marker='o', color='red', linestyle='--')
plt.plot(x, y2, marker='v', color='blue', linestyle=':')
plt.legend(['y1', 'y2'])
```



```
plt.plot(x, y1, 'or--', label='y1')
plt.plot(x, y2, 'vb:', label='y2')
plt.legend()
```

マーカーと線と色の指定

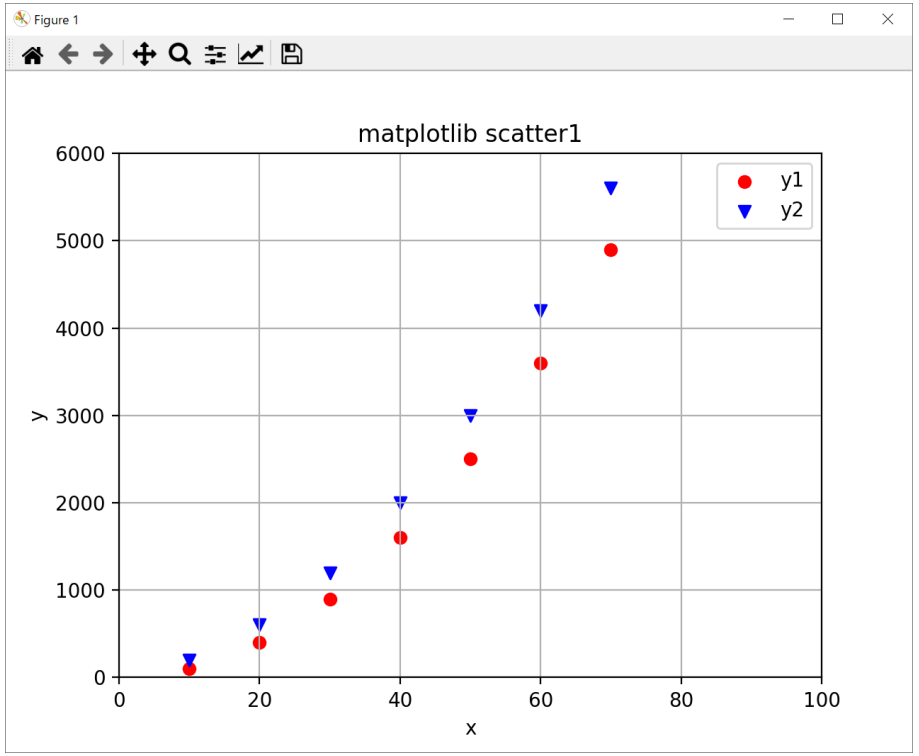
category	character	description
marker	.	point marker
marker	,	pixel marker
marker	o	circle marker
marker	v	triangle_down marker
marker	^	triangle_up marker
marker	<	triangle_left marker
marker	>	triangle_right marker
marker	1	tri_down marker
marker	2	tri_up marker
marker	3	tri_left marker
marker	4	tri_right marker
marker	8	octagon marker
marker	s	square marker
marker	p	pentagon marker
marker	P	plus (filled) marker
marker	*	star marker
marker	h	hexagon1 marker
marker	H	hexagon2 marker
marker	+	plus marker
marker	x	x marker
marker	X	x (filled) marker
marker	D	diamond marker
marker	d	thin_diamond marker
marker	|	vline marker
marker	_	hline marker

category	character	description
linestyle	-	solid line style
linestyle	--	dashed line style
linestyle	-.	dash-dot line style
linestyle	:	dotted line style
color	b	blue
color	g	green
color	r	red
color	c	cyan
color	m	magenta
color	y	yellow
color	k	black
color	w	white

散布図: scatter

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

x = [10, 20, 30, 40, 50, 60, 70]
y1 = [100, 400, 900, 1600, 2500, 3600, 4900]
y2 = [200, 600, 1200, 2000, 3000, 4200, 5600]

plt.scatter(x, y1, marker='o', color='red')
plt.scatter(x, y2, marker='v', color='blue')

plt.axis([0, 100, 0, 6000])
plt.grid(True)
plt.title('matplotlib scatter1')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(['y1', 'y2'])

plt.show()
```

scatter()



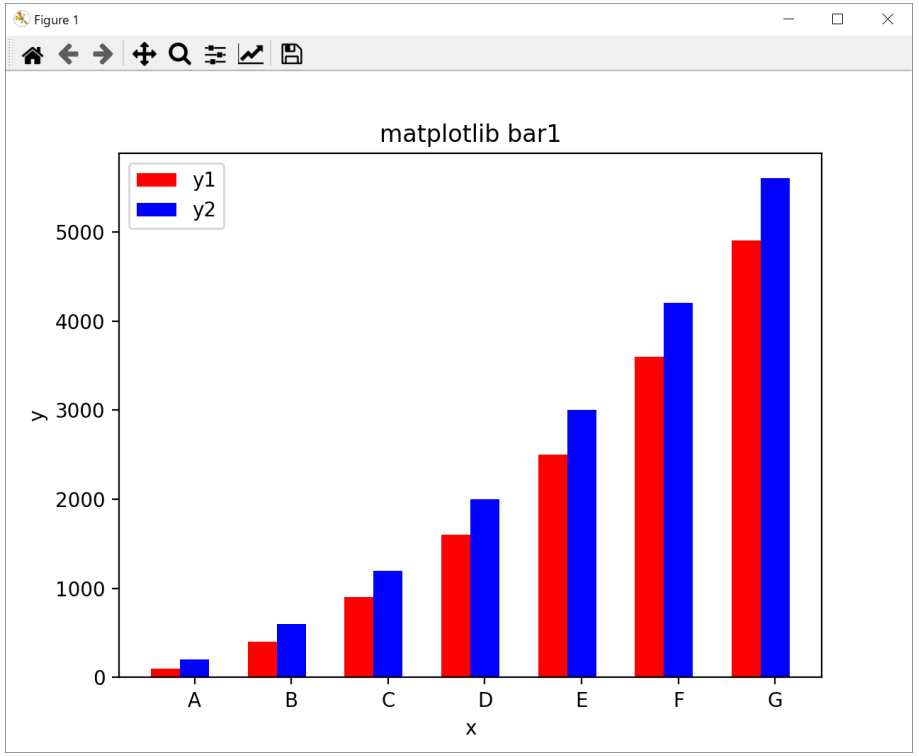
棒グラフ: bar

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

x1 = [1, 2, 3, 4, 5, 6, 7]
x2 = [1.3, 2.3, 3.3, 4.3, 5.3, 6.3, 7.3]
labels = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
y1 = [100, 400, 900, 1600, 2500, 3600, 4900]
y2 = [200, 600, 1200, 2000, 3000, 4200, 5600]
plt.bar(x1, y1, width=0.3, color='red', tick_label=labels)
plt.bar(x2, y2, width=0.3, color='blue', tick_label=labels)

plt.title('matplotlib bar1')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(['y1', 'y2'])

plt.show()
```

0.3ずらす

棒の幅も0.3にする=ずらした量と対応

bar()



補足: 2つのコーディングスタイル

- Matplotlibには記述方式が2種類ある.
  - オブジェクト指向(Object Oriented = OO)スタイル
    - グラフを描くキャンバスとなる Figure と，キャンバス内でグラフを描く領域である Axes を自分で生成する.
    - 細部まで制御が可能であり，複雑なグラフを描画できる.
  - pyplotスタイル (MATLABスタイル)
    - FigureとAxesの生成をライブラリにおまかせにする.
    - 理解が容易であり簡単にグラフを描画できる.

今日説明したのはこちら

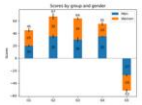
- ウェブを調べて参考にする際には，どちらのスタイルで記述されているかを見極める必要がある.

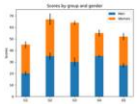
plt のみ登場→pyplotスタイル

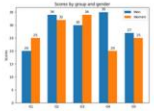
fig, ax が登場→OOスタイル

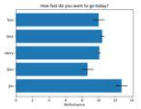
サンプルを眺めるだけで楽しい

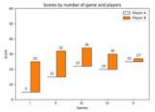
https://matplotlib.org/stable/gallery/index.html


Bar Label Demo

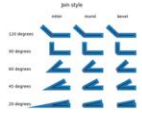

Stacked bar chart

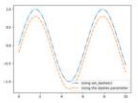

Grouped bar chart with labels

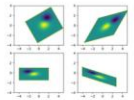

Horizontal bar chart

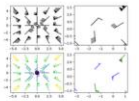

Hat graph


Discrete distribution as horizontal bar chart

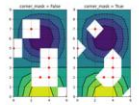

JoinStyle

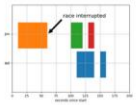

Customizing dashed line styles


Affine transform of an image

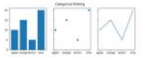

Wind Barbs

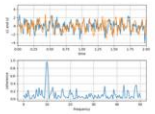

Barcode

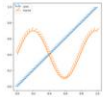

Contour Corner Mask


Broken Barh

CapStyle

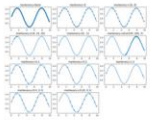

Plotting categorical variables

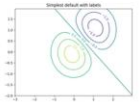

Plotting the coherence of two signals

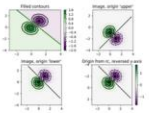

Lines with a ticked patheffect

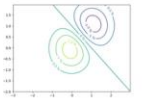

Linestyles

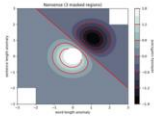

Marker reference

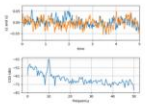

Markevery Demo

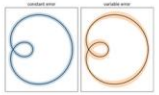

Contour Demo

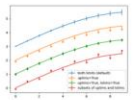

Contour Image

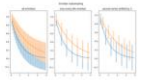

Contour Label Demo

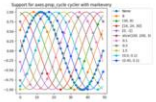

Contourf Demo

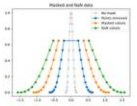

CSD Demo

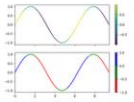

Curve with error band

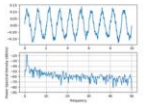

Errorbar limit selection

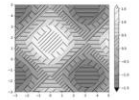

Errorbar subsampling

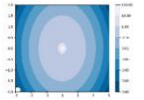

prop_cycle property markevery in rcParams

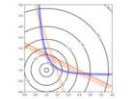

Plotting masked and NaN values

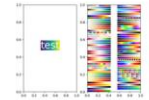

Multicolored lines

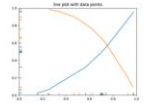

Psd Demo

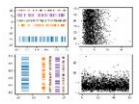

Contourf Hatching

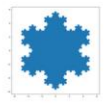

Contourf and log color scale


Contouring the solution space of optimizations


BboxImage Demo


EventCollection Demo


Eventplot Demo


Filled polygon


Fill Between and Alpha

Scatter Custom Symbol

Scatter Demo2

Scatter plot with histograms

Scatter Masked

Figimage Demo

Creating annotated heatmaps

Image antialiasing

Clipping images with patches

Filling the area between lines

Fill Betweenx Demo

Hatch-filled histograms

Bar chart with gradients

Scatter plot with pie chart markers

Marker examples

Scatter Symbol

Scatter plots with a legend

Image Demo

Image Masked

Image Nonuniform

Blend transparency with color in 2D images

NumPy+Matplotlib

Pythonの弱点

- Pythonはとっつきやすいプログラミング言語だが，実行速度は速くない．
- 特に，多数のデータに対して同じ処理を適用するときに，**for**文を使うとめちゃくちゃ遅い．
- 多数のデータに対して同じ処理を適用することは研究ではよくある．
  - 統計処理，機械学習，画像処理，数値シミュレーションなど

専用に設計された適切なパッケージを使うと
効率が桁違いに違う

数値演算パッケージ NumPy

Pythonによる科学技術計算の基礎パッケージ

最新リリース: NUMPY 1.26. すべてのリリースを表示する

numpy.orgが日本語とポルトガル語に対応しました。 2023-08-02

強力な多次元配列

NumPyの高速で多機能なベクトル化計算、インデックス処理、ブロードキャストの考え方は、現在の配列計算におけるデファクト・スタンダードです。

数値計算ツール群

NumPyは、様々な数学関数、乱数生成器、線形代数ルーチン、フーリエ変換などを提供しています。

相互運用性

NumPyは、幅広いハードウェアとコンピューティング・プラットフォームをサポートしており、分散処理、GPU、疎行列ライブラリにも対応しています。

高パフォーマ

NumPyの大部分
コードで構成さ
Pythonの柔軟性
ドの高速性の両

センスで公開
発で、互いを
コミュニティに
ンに開発され

https://numpy.org/

非常に充実しています (本日2回目)

NumPyの基本データ型: ndarray型

- 数値配列: 多数のデータを保持するという点は，リスト型に類似
  - 多次元
  - 要素のデータ型は全て同じ
- 標準のリスト型と比較した特長
  - 高速(圧倒的に)
  - 演算子の直接適用
  - 全要素に対する同一の処理の適用が容易

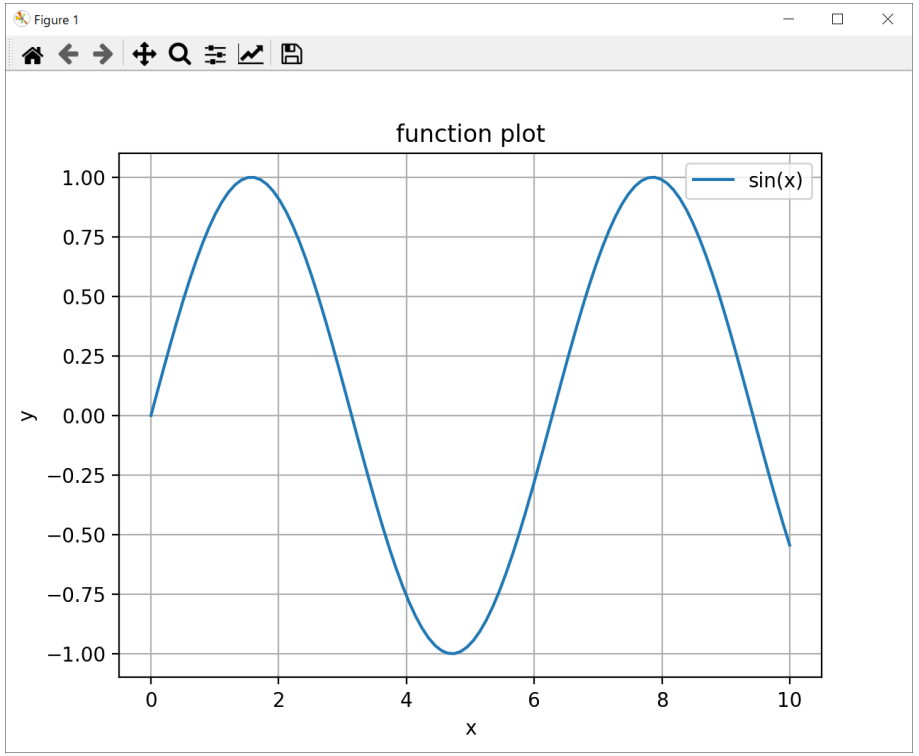
ベクトル(に限らず1次元の数値列)や
行列(に限らず2次元の数値列)を扱う際には
積極的に使おう

関数のグラフを描く

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

x = np.linspace(0, 10, 100) # 0から10まで，100点
y = np.sin(x) # NumPyのsin()を使う．mathではない．

plt.plot(x, y, label='sin(x)', linestyle='-')

plt.grid(True)

plt.title('function plot')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()

plt.show()
```

ここが新しいところ

折れ線でも，点数が十分に多ければ，
曲線に見える．



ndarray型のインスタンスの生成

- 生成方法のあれこれ
  - リストから生成: np.array([1, 2, 3, 4])
  - startからstopまでを等間隔にnum点: メソッド linspace(start, stop, num)
  - startからstopまでをstep間隔で: メソッド arange(start, stop, step)

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

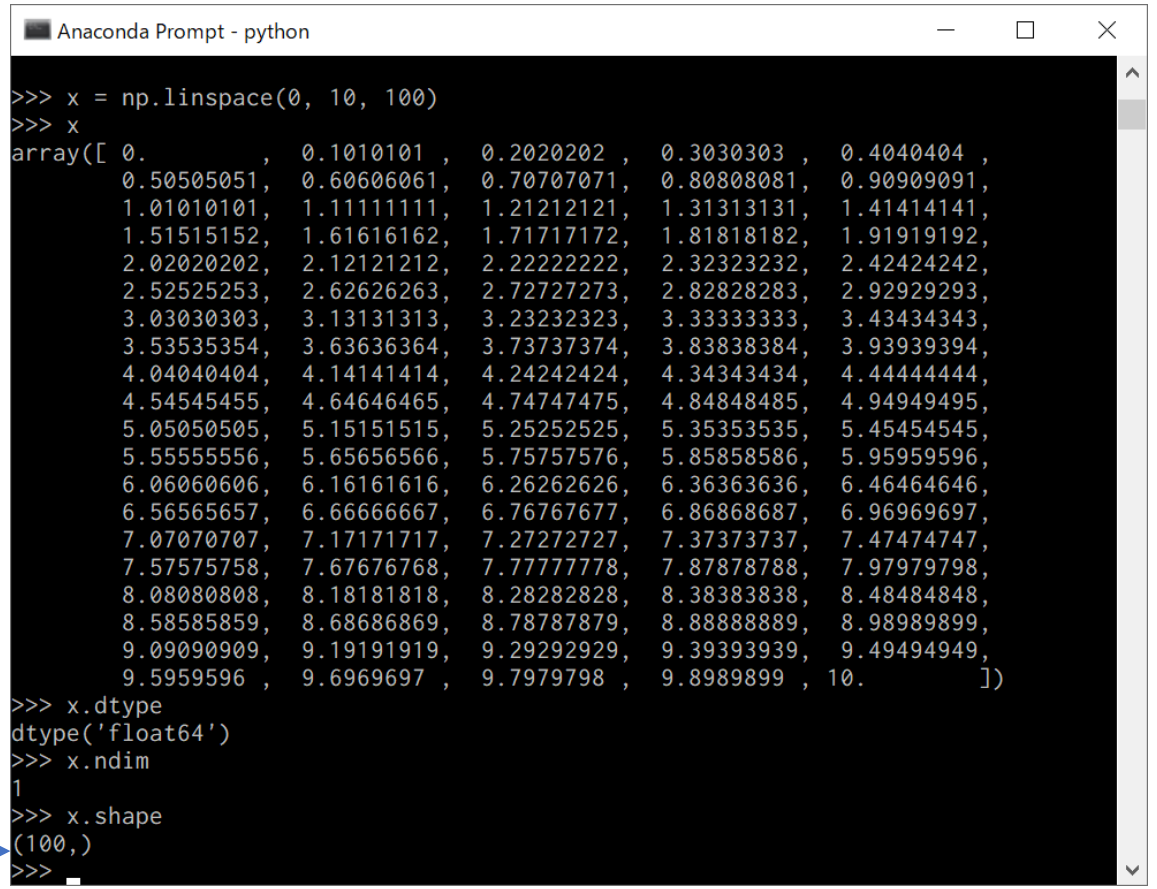
```
>>> np.linspace(0, 10, 100)
array([ 0.        ,  0.1010101 ,  0.2020202 ,  0.3030303 ,  0.4040404 ,
        0.50505051,  0.60606061,  0.70707071,  0.80808081,  0.90909091,
        1.01010101,  1.11111111,  1.21212121,  1.31313131,  1.41414141,
        1.51515152,  1.61616162,  1.71717172,  1.81818182,  1.91919192,
        2.02020202,  2.12121212,  2.22222222,  2.32323232,  2.42424242,
        2.52525253,  2.62626263,  2.72727273,  2.82828283,  2.92929293,
        3.03030303,  3.13131313,  3.23232323,  3.33333333,  3.43434343,
        3.53535354,  3.63636364,  3.73737374,  3.83838384,  3.93939394,
        4.04040404,  4.14141414,  4.24242424,  4.34343434,  4.44444444,
        4.54545455,  4.64646465,  4.74747475,  4.84848485,  4.94949495,
        5.05050505,  5.15151515,  5.25252525,  5.35353535,  5.45454545,
        5.55555556,  5.65656566,  5.75757576,  5.85858586,  5.95959596,
        6.06060606,  6.16161616,  6.26262626,  6.36363636,  6.46464646,
        6.56565657,  6.66666667,  6.76767677,  6.86868687,  6.96969697,
        7.07070707,  7.17171717,  7.27272727,  7.37373737,  7.47474747,
        7.57575758,  7.67676768,  7.77777778,  7.87878788,  7.97979798,
        8.08080808,  8.18181818,  8.28282828,  8.38383838,  8.48484848,
        8.58585859,  8.68686869,  8.78787879,  8.88888889,  8.98989899,
        9.09090909,  9.19191919,  9.29292929,  9.39393939,  9.49494949,
        9.5959596 ,  9.6969697 ,  9.7979798 ,  9.8989899 , 10.        ])
>>>
```

```
>>> np.arange(0, 10, 0.1)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. , 1.1, 1.2,
       1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. , 2.1, 2.2, 2.3, 2.4, 2.5,
       2.6, 2.7, 2.8, 2.9, 3. , 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8,
       3.9, 4. , 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 5. , 5.1,
       5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 6. , 6.1, 6.2, 6.3, 6.4,
       6.5, 6.6, 6.7, 6.8, 6.9, 7. , 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7,
       7.8, 7.9, 8. , 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9. ,
       9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9])
>>>
```

他にもいろいろある

数値配列の属性

- データ型: dtype
  - NumPy特有の型もある．
- 次元数: ndim
- 各次元のサイズ: shape
  - ndim要素のタプル



細かいが注目: 1要素のタプルなので，
数値の後ろにカンマ(,)が付いている

NumPyは超便利なのだが，複雑な計算をするときは，
自分の思ったように処理がなされているかを常に
チェックしないと落とし穴にハマるのだ…

NumPyの利点: 一括して演算

- NumPyが用意している関数の引数に，ndarray型のインスタンスを渡すと，要素に対して演算が行われる(よう設計されていることが多い).

```
x = np.linspace(0, 10, 100)
y = np.sin(x)
```

これがすばらしい

```
>>> x
array([ 0.        ,  0.1010101 ,  0.2020202 ,  0.3030303 ,  0.4040404 ,
```

以下略

sin(0)　sin(0.1010101)　sin(0.2020202)　sin(0.3030303)　sin(0.4040404)

```
>>> y
array([ 0.        ,  0.10083842,  0.20064886,  0.2984138 ,  0.39313661,
```

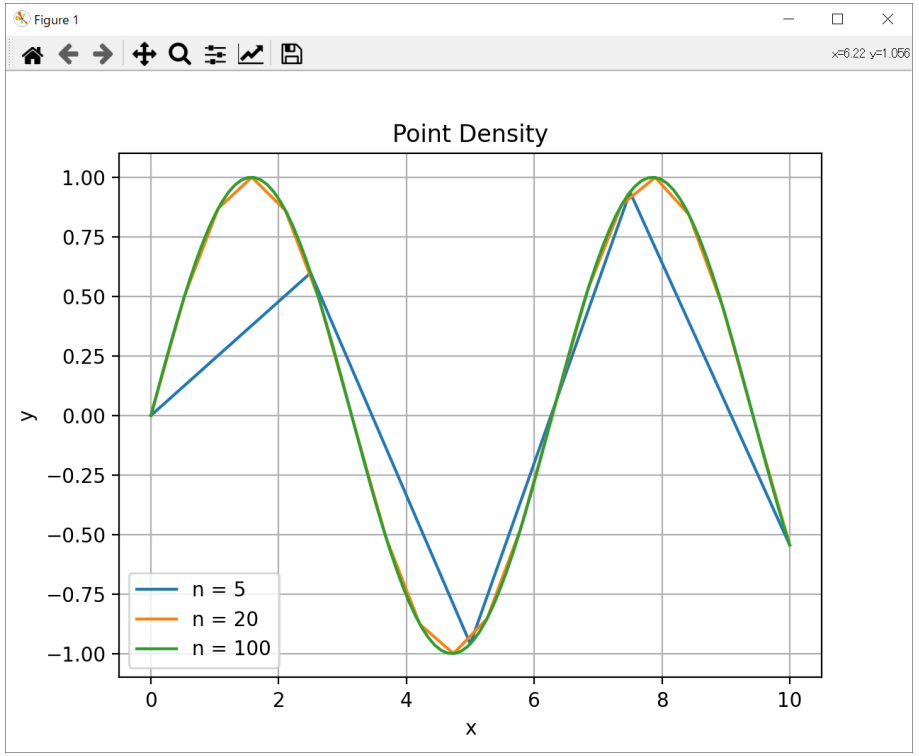
以下略

描画点の密度の違いを見てみよう

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

nums = [5, 20, 100]
for n in nums:
    x = np.linspace(0, 10, n)
    y = np.sin(x)
    plt.plot(x, y, label=f'n = {n}')
plt.grid(True)

plt.title('Point Density')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()

plt.show()
```

0から10を5,20,100点で分割

各プロット時に凡例のラベルを指定



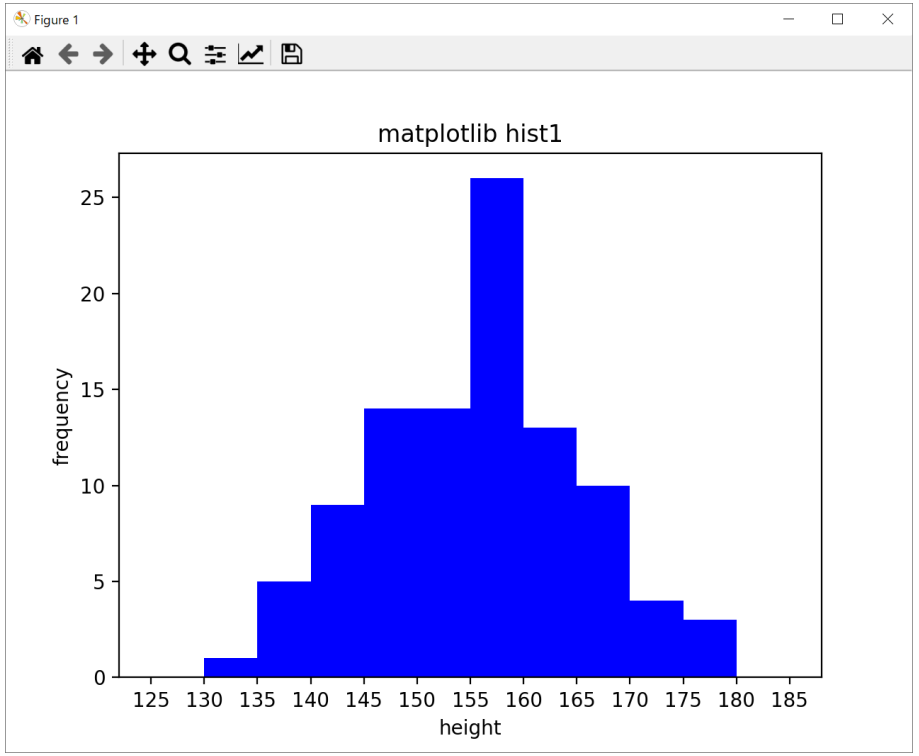
ヒストグラムの描画

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

y = [144.7772377, 156.61120511,
        : (中略)
]

bins = np.arange(125, 190, step=5)
plt.xticks(bins)
plt.hist(y, bins=bins, color='b')

plt.title('matplotlib hist1')
plt.xlabel('height')
plt.ylabel('frequency')

plt.show()
```

[125, 130, 135, ..., 185]

yの度数分布を
自動的に数えて
ヒストグラムを描画する

階級の境界は
キーワード引数binsで指定



NumPyで乱数の生成

- 一様分布
  - 0以上1未満: numpy.random.rand()

```
>>> import numpy as np
>>> np.random.rand()
0.3718781674778411
>>> np.random.rand(10)
array([0.74428865, 0.6600082 , 0.48094929, 0.90761384, 0.72157137,
       0.58062677, 0.53435835, 0.65049782, 0.34109312, 0.28902556])
>>> np.random.rand(2,3)
array([[0.00093929, 0.32747377, 0.83070672],
       [0.29319319, 0.59521091, 0.15475263]])
```

最大値未満であってます．https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html

  - 整数で最小値以上最大値未満: numpy.random.randint()

```
>>> np.random.randint(10, 20, (2,3))
array([[19, 13, 13],
       [12, 15, 15]])
```

NumPyで乱数の生成

- 正規分布(ガウス分布)
  - numpy.random.normal()

```
>>> np.random.normal(10, 3, 5)
array([ 7.00252856, 11.19460368, 10.37929223,  3.94753513, 13.87448736])
```

- カイ二乗分布
  - numpy.random.chisquare()

```
>>> np.random.chisquare(10, 3)
array([ 3.70247222,  5.00100389, 17.93678927])
```

他にもいろいろある

NumPyでベクトル・行列計算

NumPyでベクトル・行列計算: 生成

```
x = np.array([1.0, 2.0, 3.0])
print(f'x={x}')
print(f'dtype={x.dtype}')
print(f'ndim={x.ndim}')
print(f'shape={x.shape}')

A = np.array([[1, 2], [3, 4]])
print(f'A={A}')
print(f'dtype={A.dtype}')
print(f'ndim={A.ndim}')
print(f'shape={A.shape}')
```

メソッド reshape() で形状を変えられる

```
B = np.arange(6, dtype=float).reshape(2, 3)
print(f'B={B}')
print(f'dtype={B.dtype}')
print(f'ndim={B.ndim}')
print(f'shape={B.shape}')
```

dtype=floatと型を明示的に指定する

```
x=[1. 2. 3.]
dtype=float64
ndim=1
shape=(3,)

A=[[1 2]
 [3 4]]
dtype=int32
ndim=2
shape=(2, 2)

B=[[0. 1. 2.]
 [3. 4. 5.]]
dtype=float64
ndim=2
shape=(2, 3)
```

ここ要注意
初期化時に要素が全て整数だと，行列全体が整数型になる．

要素が全て整数でもfloat型になった

※見やすいように空白や空行を追加しています．

NumPyでベクトル・行列計算: 要素

```
x = np.array([1.0, 2.0, 3.0])
print(f'x[0]={x[0]}')
print(f'x[1]={x[1]}')
print(f'x[2]={x[2]}')

A = np.array([[1, 2], [3, 4]], dtype=float)
print(f'A[0,0]={A[0,0]}')
print(f'A[1,0]={A[1,0]}')
print(f'A[0,1]={A[0,1]}')
print(f'A[1,1]={A[1,1]}')

B = np.arange(6, dtype=float).reshape(2, 3)
print(f'B[0,0]={B[0,0]}')
print(f'B[0]={B[0]}')
print(f'B[:,1]={B[:,1]}')
print(f'B[:,0:1]={B[:,0:1]}')
```

```
x[0]=1.0
x[1]=2.0
x[2]=3.0


A[0,0]=1.0
A[1,0]=3.0
A[0,1]=2.0
A[1,1]=4.0


B[0,0]=0.0
B[0]=[0. 1. 2.]
B[:,1]=[1. 4.]
B[:,0:1]=[[0.]
 [3.]]
```

A[行,列]の順
0はじまりであることに注意

:だと全選択
列指定だと1次元に

スライスも使える
スライスだと縦ベクトル

※見やすいように空白や空行を追加しています．

NumPyでベクトル・行列計算: 加減算

```
A = np.array([[1, 2], [3, 4]], dtype=float)
B = np.array([[5, 6], [7, 8]], dtype=float)
C = A + B
D = A - B
E = A + 1
F = A * 2
print(f'A={A}')
print(f'B={B}')
print(f'A+B={C}')
print(f'A-B={D}')
print(f'A+1={E}')
print(f'2A={F}')
```

数学ではこの記法(行列に数値を加える)はNG

```
A=[[1. 2.]
 [3. 4.]]

B=[[5. 6.]
 [7. 8.]]

A+B=[[ 6.  8.]
 [10. 12.]]

A-B=[[-4. -4.]
 [-4. -4.]]

A+1=[[2. 3.]
 [4. 5.]]

2A=[[2. 4.]
 [6. 8.]]
```

ndarrayにスカラー値を加えると全要素に1が加えられる(broadcast機能)

スカラー倍は数学と同じ

※見やすいように空白や空行を追加しています.

NumPyでベクトル・行列計算: 行列の積

```
A = np.array([[1, 2], [3, 4]], dtype=float)
B = np.array([[5, 6], [7, 8]], dtype=float)
C = A * B
D = A.dot(B)
E = np.dot(A, B)
F = A @ B
print(f'A={A}')
print(f'B={B}')
print(f'A * B は要素ごとの積で行列の積ではない:
{C}')
print(f'AB = {D} (メソッドdot()で計算)')
print(f'AB = {E} (関数numpy.dot()で計算)')
print(f'AB = {F} (演算子 @ で計算)')
```

```
A=[[1. 2.]
 [3. 4.]]
B=[[5. 6.]
 [7. 8.]]
A * B は要素ごとの積で行列の積ではない:
[[ 5. 12.]
 [21. 32.]]
```

行列に * を適用すると要素ごとの積
=アダマール積 (Hadamard product)

```
AB = [[19. 22.]
 [43. 50.]] (メソッドdot()で計算)
AB = [[19. 22.]
 [43. 50.]] (関数numpy.dot()で計算)
AB = [[19. 22.]
 [43. 50.]] (演算子 @ で計算)
```

いずれでも正しく計算できる

※見やすいように空白や空行を追加しています.

NumPyでベクトル・行列計算: 逆行列

```
A = np.array([[1, 2], [3, 4]], dtype=float)
det_A = np.linalg.det(A)
A_inv = np.linalg.inv(A)
print(f'A={A}')
print(f'det(A)={det_A}')
print(f'A^-1={A_inv}')
E = A @ A_inv
F = A_inv @ A
print(f'A A^-1={E}')
print(f'A^-1 A={F}')
```

```
A=[[1. 2.]
 [3. 4.]]

det(A)=-2.0000000000000004

A^-1=[[-2.   1. ]
 [ 1.5 -0.5]]

A A^-1=[[1.00000000e+00 1.11022302e-16]
 [0.00000000e+00 1.00000000e+00]]

A^-1 A=[[1.0000000e+00 4.4408921e-16]
 [0.0000000e+00 1.0000000e+00]]
```

数値演算誤差のため-2にならない

完全に単位行列にならないのは
数値演算誤差のため

※見やすいように空白や空行を追加しています．