4.　折れ線グラフ、棒グラフ、ヒストグラム、散布図

4.1 実際の気象データで折れ線グラフを試してみる

前章で折れ線グラフの基本的な使い方やグラフの体裁の整え方を学んだところで、実際の気象データを使った作図をしてみましょう。複雑なグラフを作る場合には手続きをスクリプトにしてファイルに保存した方が再利用できて便利です。Jupypter Notebook の場合には、%run スクリプト名.py で実行可能です。macOS や Linux のターミナルでは ./スクリプト名.py でも実行できます。Windows の WSL2 上に構築した Linux 環境でも同じです。なお、コードの中身を見ながら何が起こるか逐次試していく場合には、Jupypter Notebook のセルにコードを貼り付けてください。

気象データを python で取得するには、python の知識が必要なので、サンプルデータを取得するプログラムを用意しました。ドキュメントと一緒に公開している met_sample.zip を使います。met_sample.zip を展開すると、サンプルプログラムが入った met_sample というディレクトリが生成されます。Jupypter Notebook の場合には、met_sample まで移動します。それではサンプルスクリプトの amedas_temp.py を見てみます。最初の部分は、python3 と UTF-8 を使うおまじないです。python3 ではデフォルトで UTF-8 であるため、２行目を省いても問題ありません。以降のスクリプトでは省略しています。

３～４行目で import するのは、これまでと同じで plt と ticker です。最後の from amesta import AmedasStation は、amesta に入っている AmedasStation Class を利用するという意味ですが、ここでは AmedasStation という python の Class を使うということだけ把握できていれば問題ないです（Class も知らない場合は、matplotlib で plt や ticker を呼び出したように、参照して使う道具を呼び出したものと思っておいて下さい)。

全体に有効なものとして、地点名 sta="Tokyo"と月 month="jul"を定義しています。他の地点名を選ぶことも可能なので、amesta_list.txt に記載されている地点名から見たい地点を選んでください（各県で 1 ヶ所以上、主要な気象台や測候所を選択可能にしています)。月については"jan"から"dec"まで全ての月が参照可能です。年平均を選ぶ場合は"ann"を指定してください。

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
from amesta import AmedasStation

sta = "Tokyo" #  地点名
month = "jul" #  月（3 文字）
```

次がプログラムの主要部分です。まずはアメダスの平均気温データを取得します。先ほどの AmedasStation Class を初期化します。このように地点名(Tokyo)を渡すと、対応する地点のアメダスデータの情報が設定されるものと思っておいてください。東京のアメダス地点の情報が設定されたものを amedas に渡しているので、以降は amedas を操作する（メソッドを使用する）と東京のアメダスデータを取得可能になります。なお、Class から生成した amedas のようなものをインスタンスと呼んでおり、初期化を行うとクラスの中に記述された関数（メソッドと呼ばれる）を呼び出すことができるようになります。amedas.メソッドのように使用するので、これをインスタンスメソッドと呼んでいます。（よく分からない場合は、変数がツールを持っているようなものと思っておいてください）

```
# AmedasStation Class の初期化
amedas = AmedasStation(sta)
```

実際のデータ取得は、amedas.retrieve_mon で行います。AmedasStation Class の中に retrieve_mon というメソッドが入っており、これを利用するとデータを取得してきます。引数として取得したいデータの名前を渡します（ここでは平均気温を表す tave を渡す）。得られたデータが tave_i に格納されます。このうちの 7 月のデータを取り出したいので、tave_i.loc[:, month]というインディックス参照の方法で 7 月のみ選択しています。month="jul"としたので、tave_i.loc[:, "jul"]と同じ処理が行われます。tave_i は Pandas の DataFrame 形

式になっていて、.loc はデータの名前で取り出すメソッドを表しています。この.loc では、python のリストで使われているスライスと似た記述方法を使うことができ、年時系列に当たる 1 次元目は全て（コロンを単独で用いると全て選択するという意味)、月に当たる２次元目は７月のみを取り出すことを意味しています。２次元目は１つのデータのみを選択したので、取り出された変数の次元が１つ下がり、１次元目の変数で構成された Series に変わります。なお、通常のスライス記法と違い、Pandas の場合には**始点:終点**と記載した場合、終点の１つ前ではなく終点まで取り出されるため注意してください。

ここでは python の Pandas を知らなくても問題ないので、こういう書き方をしたらデータを選択できることだけ把握しておいて下さい。

なお、バージョン０系（バージョン 0.7.3 以前）の Pandas では既に古いものとなっている.ix メソッドを.loc メソッドの代わりに使うことができました。最新版の Pandas では.ix メソッドが廃止され、参照するものが行・列のラベルか行・列番号かで.loc メソッド（行・列ラベルの名前で参照）か.iloc メソッド（行・列番号で参照）を使い分けることになりました。バージョン０系でも.loc や.iloc を使うことはできるため、最新版でも問題なく動作するように、本稿の解説では.loc、.iloc メソッドを用いたサンプルプログラムを使用します。

```
# AmedasStation.retrieve_mon メソッドを使い、平均気温データを取得
tave_i = amedas.retrieve_mon("tave")
#  データの取り出し
tave = tave_i.loc[:, month]
```

作図部分に移ります。前節のグラフのように、ウィンドウサイズを(6, 3)の横長にして１つのサブプロットを追加し、タイトルを付けています。

```
fig = plt.figure(figsize=(6, 3))
ax = fig.add_subplot(1, 1, 1)
#
#  タイトルを付ける
title = "Yearly timeseries of July, Tokyo"
plt.title(title)
```

次が気温の折れ線グラフをプロットする部分です。x 軸の年と y 軸の気温データが tave に入っているので、tave のみで折れ線グラフを作図可能です。plt.ylim([y 軸の最小値,y 軸の最大値])は、y 軸の範囲を指定するメソッドです。plt.ylabel は前節で出てきましたが、文字の他に tex と同じ書式で数式を記述できます。

```
plt.ylim([20, 30])
plt.plot(tave, color='r', ls='-', label='Ave. Temp.')
plt.ylabel('Temperature ($^{¥circ}$C)')
```

最後に図の体裁を整えていきます。前節同様、y 軸、x 軸の大目盛り、小目盛りを set_major_locater、set_minor_locator で設定します。別の地点データでも対応できるように目盛りの間隔は自動設定にしたいので、AutoLocator を使いました。

```
# y 軸の目盛り
ax.yaxis.set_major_locator(ticker.AutoLocator())
ax.yaxis.set_minor_locator(ticker.AutoMinorLocator())
# x 軸の目盛り
ax.xaxis.set_major_locator(ticker.AutoLocator())
ax.xaxis.set_minor_locator(ticker.AutoMinorLocator())
```

凡例を付け、灰色の点線でグリッド線も描いておきます。プロット範囲を調整し、水平を 8 割の大きさに下に 2 割の空きを付けます。

```
plt.legend(loc='best') #  凡例を付ける
plt.grid(color='gray', ls=':') #  グリッド線を描く
plt.subplots_adjust(hspace=0.8, bottom=0.2) #  プロット範囲の調整
```

作成した図をファイルに保存するため、plt.savefig(fig_fname)を行なっています。ファイルを開くと図 4 – 1 – 1 のように表示されるでしょう。図の名前

fig_fname の記述は、python に慣れていないと違和感があるかもしれません。"文字列 1 "+"文字列 2"のように文字列同士を加算すると、"文字列 1 文字列 2"のように追記するという意味になります。plt.savefig では、３章で出てきた解像度を 300 dpi にする dpi=300 と余白を少なくする bbox_inches='tight'を使いました。スクリプトにしてターミナルから実行する際には、画面上で表示するために plt.show()が必要です。Jupyter Notebook 上にコードをコピーペーストして実行した際や、%run amedas_temp.py で実行した場合には plt.show()は不要です。なお plt.savefig と plt.show には実行する順番があり、先に plt.show を行うと、その時点でプロットがクリアされてしまうので、plt.savefig を行なった後で plt.show を行います。

```
fig_fname = "Fig4-1-1.png"
plt.savefig(fig_fname, dpi=300, bbox_inches='tight') #  ファイルへの書き出し
plt.show() #  画面へ表示
```

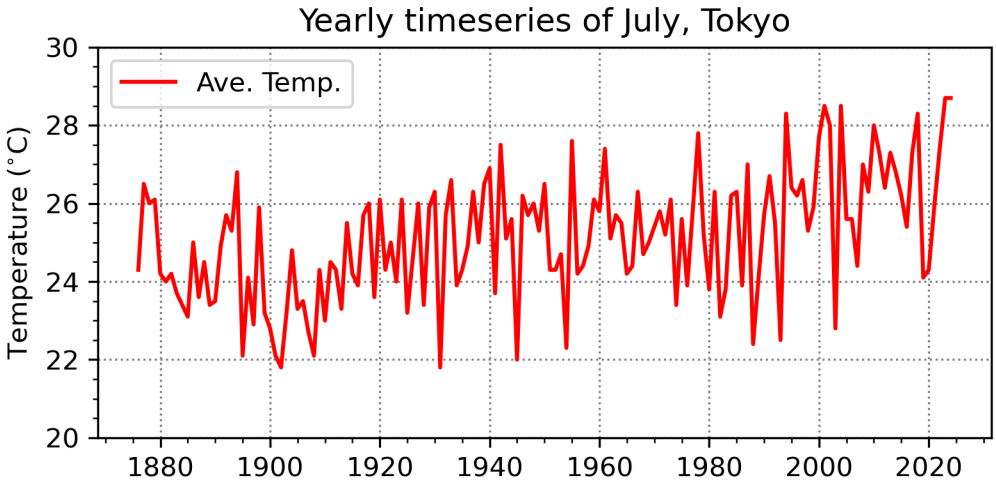


図４－１－１　東京のアメダス地点における７月平均気温を年時系列図にしたもの

Jupyter Notebook 上で作図する場合、作図を開始する fig = plt.figure から図を保存する plt.savefig までは、同じセル上に記載します。インポート部分は別のセルに記載可能ですが、Notebook 上では上から順に実行していき、先にインポートを行ってから作図が行われるようにしないと不具合が発生します。なお、plt.savefig や plt.show を行うと、それまでの作図がリセットされるので、

それ以降に作図の記載を行うと不具合が発生します。また、同じ fig や ax を定義する記載を複数回行うと、それ以前に記載した内容がリセットされるため、不具合が起こった時は最初のみに記載しているか確認してみてください。それから、長時間同じ Notebook を使っている場合や、多くのモジュールをインポートした場合、様々なテストをして多くの変数を定義した場合など、メモリ使用量が増え、途中から不具合が発生することがあります。その場合、まずは新しい Notebook を立ち上げて必要なコードだけをコピーペーストしてみてください。また多くのタブを開いている場合は、必要なものだけ残して閉じましょう。

次に amedas_temp2.py を見てみます。先ほどは平均気温を取り出しましが、今度は同じ amedas.retrieve_mon メソッドを使い、最高気温データ（"tmax"）と最低気温データ（"tmin"）も取り出しています。他には wind で風速（m/s）、slp で海面更生気圧（hPa）、ps で地表気圧（hPa）、RH で相対湿度（%）、prep で降水量（mm）などを取得可能です。取得可能な変数名は、amesta_list.txt に記載しています。

```
# AmedasStation.retrieve_mon メソッドを使い、最高気温データを取得
tmax_i = amedas.retrieve_mon("tmax")
# AmedasStation.retrieve_mon メソッドを使い、平均気温データを取得
tave_i = amedas.retrieve_mon("tave")
```

データの取り出し部分では、作図に用いる年の範囲を指定できるように書き換えています。まずプログラムの最初の部分で、次のように開始年（syear）、終了年（eyear）を整数で指定します。ここでは開始年を 1900 年、終了年を 2024 年としました。

```
syear = 1900 #  開始年
eyear = 2024 #  終了年
```

指定しない場合には、次のように None とします（none や文字列"None"ではないので注意）。

```
syear = None #  開始年
eyear = None #  終了年
```

データ取り出し部分では、tmin_i.loc で年時系列に当たる 1 次元目に syear:eyear を持ってきており、時系列の該当する年を選択しています。syear、eyear の両方に None を入力すると、コロンを単独で用いた場合と同じ意味になり、全ての期間が選択されます。他にも python のリストで使われているスライスと似た記述方法が使えて、コロンの場合は全ての範囲を選択、**始点:終点**では始点〜終点までの範囲を選択、**:終点**ではデータの最初から終点まで、**始点:**では始点からデータの最後までが選択されます。(再掲：通常のスライス記法と違い、終点の 1 つ前ではなく終点まで選択されます。)

```
#  データの取り出し
tmin = tmin_i.loc[syear:eyear, month]
tmax = tmax_i.loc[syear:eyear, month]
tave = tave_i.loc[syear:eyear, month]
```

気温の折れ線グラフをプロットする部分では、まず y 軸の範囲の指定方法を自動化するように変更しました。そのために、追加で各種数学関数を含む math パッケージを import しています。

```
import math
```

plt.ylim メソッドに y 軸の範囲を渡す部分では、次のように最低気温データの最低値よりも 2 度以上小さく、最高気温データの最高値よりも 5 度以上大きい整数になるような調整を行っています。ここで math.floor(x)は x の「床」(x 以下の最大の整数) を返す関数、math.ceil(x)は x の「天井」(x 以上の最小の整数) を返す関数です。math パッケージには他にも多くの数学関数が含まれます。一覧を表 4 – 1 – 1 にまとめました。Numpy の数学関数と同じものがあり、どちらを使っても良い場合もありますが、Numpy ではスカラーだけではなく多次元の ndarray を扱うことができ、ndarray の場合には Numpy を選びます。

なお tmin や tmax は Pandas の Series で、Series では Numpy の ndarray で使われている max()や min()というメソッドを持っています。そのため、tmin.min()で最低値、tmax.max()で最大値を返します。Pandas を知らない場合は、こういう便利な機能があるというくらいに思っておいて下さい。

```
plt.ylim([math.floor(tmin.min() - 2), math.ceil(tmax.max()) + 5])
```

先ほどの平均気温の折れ線に加えて、最低気温から最高気温の範囲までを plt.fill_between で塗り潰しています（図４－１－２）。plt.fill_between では、先ほど述べたように最初の３つの引数が順に x 軸の値、y 軸を塗りつぶす下限値、y 軸を塗りつぶす上限値のように解釈されます。ここで alpha=0.4 の指定が威力を発揮します。平均値のグラフを隠さずに、最低気温と最高気温の範囲を表示できているのが分かると思います。

先ほどは平均気温の折れ線を作図する時に x 軸の年と y 軸の気温データを tave のみで渡していましたが、ここでは２つの引数にして渡しているのに気が付いたでしょうか。tave 自体が x 軸、y 軸両方の情報を持っていますが、plt.plot で最初の引数を２つ渡した場合は、１番目（index）の引数のデータから x 軸の情報を、２番目（tave）の引数のデータから y 軸の情報を取り出します。（実際には index は tmin.index で取り出された Pandas の index、tave は pandas の Series ですが、Pandas を知らない場合は、この程度の理解でも十分です）

```
index = tmin.index
plt.plot(index, tave, color='r', ls='-', label='Ave. Temp.')
plt.fill_between(index, tmin, tmax, color='r', alpha=0.4)
```

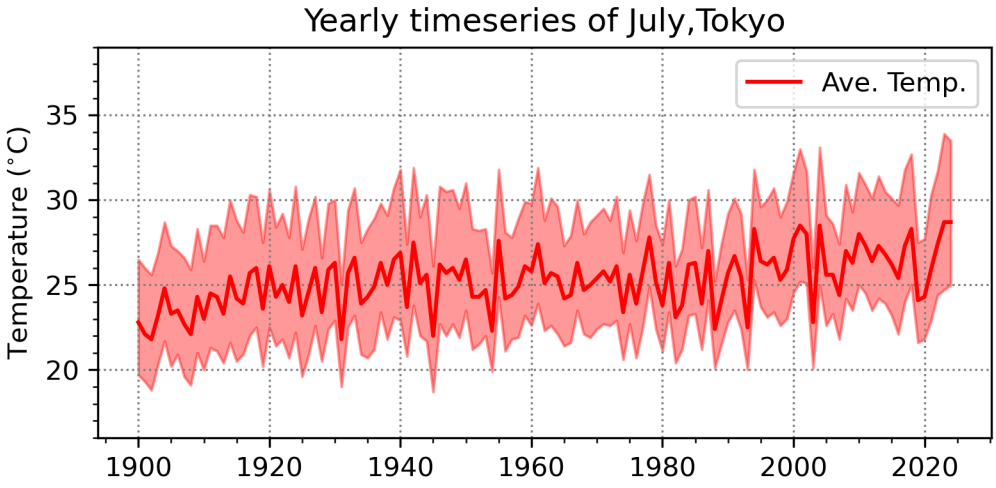


図４－１－２　東京の７月平均気温（赤線）の年時系列図に、最低気温と最高気温の間を赤の陰影で塗りつぶしたものを重ねた

表４－１－１　mathパッケージに含まれる主要な数学関数の一覧

数学関数	説明
math.ceil(x)	x の「天井」（x 以上の最小の整数）を返す
math.floor(x)	x の「床」（x 以下の最大の整数）を返す
math. fabs(x)	x の絶対値を返す
math. fmod(x, y)	x % yと同じ演算（x, yが浮動小数点数の場合、%より正確）
math.exp(x)	e**x を返す
math.expm1(x)	e**x – 1を返す
math.log(x[, base])	引数が1つの場合、 x の自然対数：loge(x) 引数が2つの場合、loge(x)/loge(base)
math.pow(x, y)	x の y 乗（x**y）を返す
math.factorial(x)	x の階乗を返す。xは正数（正の整数値）のみ可
math.copysign(x, y)	x の大きさ（絶対値）で y と同じ符号の浮動小数点数を返す。 t例えばcopysign(1.0, -0.0) は -1.0
math.fsum(iterable)	iterable 中の値の浮動小数点数の正確な和を返す
math.gamma(x)	x の ガンマ関数 を返す
math.lgamma(x)	x のガンマ関数の絶対値の自然対数を返す
math.pi	π を返す
math.e	eを返す
math.inf	浮動小数の正の無限大、-math.infで負の無限大。float('inf')と等価
math.nan	浮動小数の非数。float('nan')と等価
math.degrees(x)	角 x をラジアンから度に変換
math.radians(x)	角 x を度からラジアンに変換
math.hypot(x, y)	ユークリッドノルム(sqrt(x*x + y*y))を返す
math.cos(x)	x ラジアンの余弦cos(x)を返す
math.sin(x)	x ラジアンの正弦sin(x)を返す
math.tan(x)	x ラジアンの正接tan(x)を返す
math.acos(x)	x の逆余弦をラジアンで返す
math.asin(x)	x の逆正弦をラジアンで返す
math.atan(x)	x の逆正接をラジアンで返す
math.cosh(x)	x の双曲線余弦cosh(x)を返す
math.sinh(x)	x の双曲線正弦sinh(x)を返す
math.tanh(x)	x の双曲線正接tanh(x)を返す
math.acosh(x)	x の逆双曲線余弦を返す
math.asinh(x)	x の逆双曲線正弦を返す
math.atanh(x)	x の逆双曲線正接を返す
math.atan2(y, x)	atan(y / x) を、-pi～pi の間で返す 極座標平面において原点から (x, y) へのベクトルが X 軸の正の方向となす角

ところで、プロットしたデータそのものを保存しておきたいこともあると思います。Pandas には csv 形式で書き出す to_csv という便利なメソッドがあり、tmax_i.to_csv("tmax.csv")のようにデータ保存が可能です。

このケースでは tmin は必ず tmax 以下であるので意味はありませんが、4 番目の引数として where=tmin<tmax のような条件式を追加できます。

```
plt.fill_between(index, tmin, tmax, where=tmin<tmax, color='r', alpha=0.4)
```

条件式を利用できるようなケースを考えてみます。2018 年 7 月は、中旬から下旬にかけて高温が続き、各地で夏日、真夏日、猛暑日の日数が更新されました。夏日は日最高気温が 25℃以上の日、真夏日は日最高気温が 30℃以上の日、猛暑日は日最高気温が 35℃以上の日のことです。

これまで使ってきた AmedasStation クラスには、retrieve_day(年，月)という指定した月の日毎のデータを返すメソッドもあります。それを用いて日毎のアメダスデータを取得し、日最高気温を抽出してみます。プログラムは amedas_day_temp.py です。このプログラムを使い、東京、名古屋と 7 月 23 日に観測史上 1 位を更新した熊谷（41.1℃）の日最高気温をプロットしたものが図 4－1－3 です。色を塗った部分は真夏日の基準を満たしていた日で、名古屋、熊谷の場合は数日間だけ連続して真夏日ではない日があるのに対して、東京では真夏日の基準から外れた期間が 2 度あったことが読み取れます。また東京と熊谷では日最高気温の日々の変化の仕方は似ていますが、熊谷では全体的に気温が嵩上げされている様子も分かります。

プログラムでは、最初に年、月とその月の日数を指定しています。日数は作図範囲の右端の指定に用いているので、実際のデータ長（7/31 に対応する 31）より短い値も指定可能です。

```
year=2018
mon=7
days=31
```

AmedasStation クラスの初期化方法はこれまでと同じで、東京（sta= "Tokyo"）と名古屋（sta="Nagoya"）、熊谷（sta = "Kumagaya"）のアメダス地点の情報を取得して amedas に渡しています。retrieve_day メソッドは、年、月を引数とし、1 ヶ月分の毎日のデータを返却します。返却されたデータには、1 次元目として該当月の日数分のデータ、2 次元目として日平均気温（tave、単位は℃）、日最高気温（tmax、℃）、日最低気温（tmin、℃）、降水量（prep、mm）などの変数を含んでいます。ここでは全期間の日最高気温を取り出すため、tmax = dat_i.loc[:, 'tmax']のように、1 次元目にコロン、2 次元目に変数名'tmax'を指定して取り出します。

```
amedas = AmedasStation(sta) # AmedasStation Class の初期化
# AmedasStation.retrieve_day メソッドを使い、データを取得
dat_i = amedas.retrieve_day(year, mon)
tmax = dat_i.loc[:,'tmax'] #  日最高気温データの取り出し
```

作図部分ですが、index の取り出しや plt.plot はこれまで同様で、tave の代わりに tmax を使っています。x 軸の範囲については、7/1 から先ほど設定した days まで（7/31 まで）を指定しています。y 軸の範囲に関しては、計算方法はこれまで同様ですが、今回は日最高気温のみを使うため、tmax.min()、tmax.max()を使い、日最高気温の最低値と最大値を取り出しています。

```
index = tmax.index # index 取り出し
plt.xlim([1, days]) # x 軸の範囲
plt.ylim([math.floor(tmax.min() - 2), math.ceil(tmax.max()) + 5]) # y 軸の範囲
plt.plot(index, tmax, color='r', ls='-', label='Max. Temp.') #  折れ線グラフ
```

さらに、plt.fill_between を使い、30 度以上の日の領域全体を塗り潰しています。1 番目の引数は index、2 番目の引数は np.zeros(len(index))で、日数分だけ 0 を並べた値です。3 番目の引数は tmax なので、0℃から日最高気温までの間を塗りつぶすことを意味しています。0℃は y 軸の下限よりもはるかに小さな値なので、グラフの下端から塗り始めることになります。4 番目の引数が条件式で、where=tmax>=30 なので、日最高気温が 30℃以上の場合にのみ塗り潰す

ことになっています。

```
# 30 度以上の日の領域を塗り潰す
plt.fill_between(index, np.zeros(len(index)), tmax, where=tmax>=30, ¥
                        color='r', alpha=0.4)
```

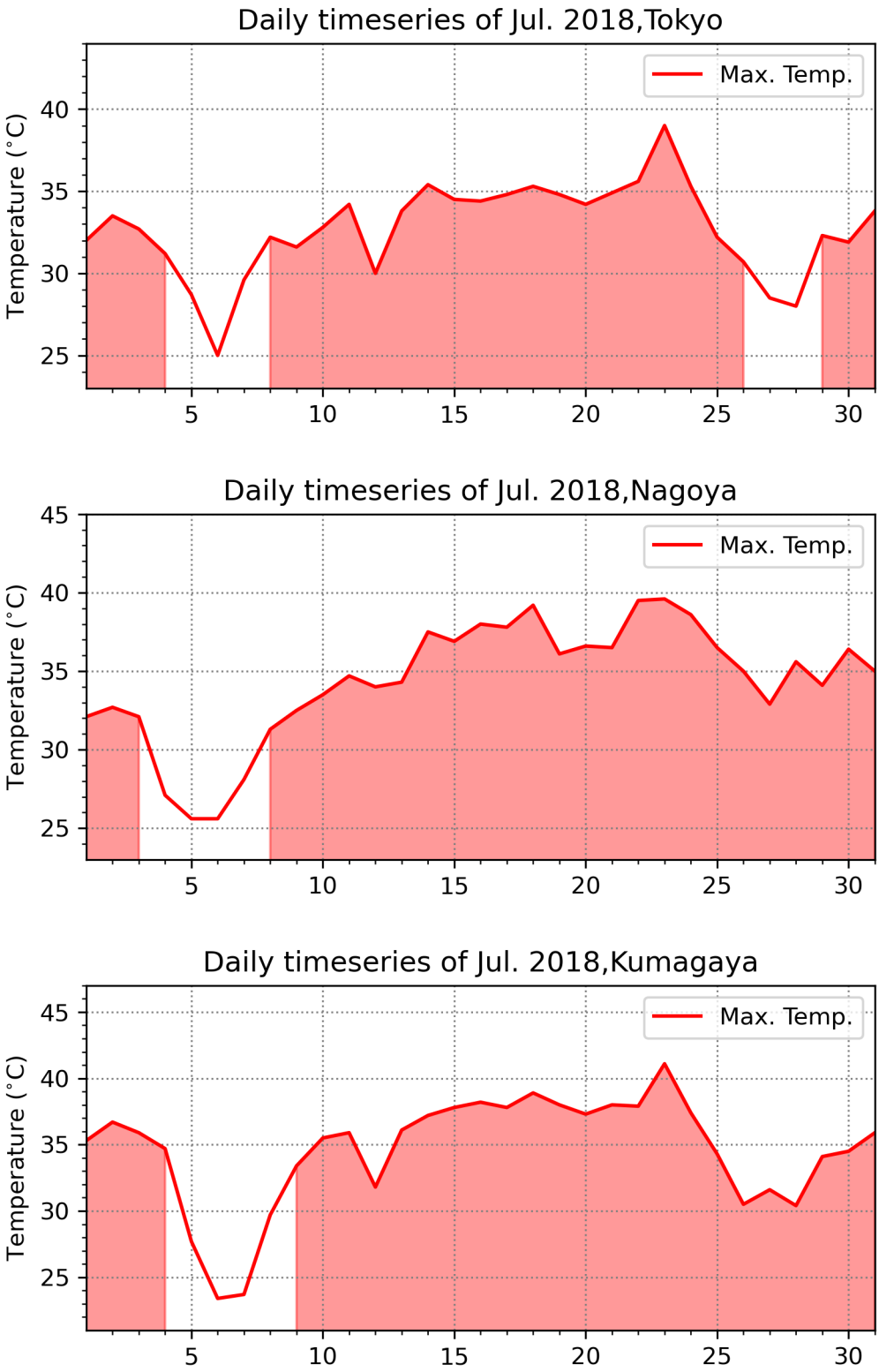


図４－１－３　（上）東京と（中）名古屋、（下）熊谷のアメダス地点における 2018 年 7 月の日最高気温。真夏日には色を付けた。

4.2 棒グラフの作成

4.2.1 基本的な棒グラフ

これまでは折れ線グラフでしたが、他のグラフを作成したいこともあるでしょう。ここでは棒グラフの作成を行います。棒グラフは plt.bar で描きます。まず、図４－２－１のような棒グラフを作成するプログラム（amedas_prep.py）を見てみましょう。

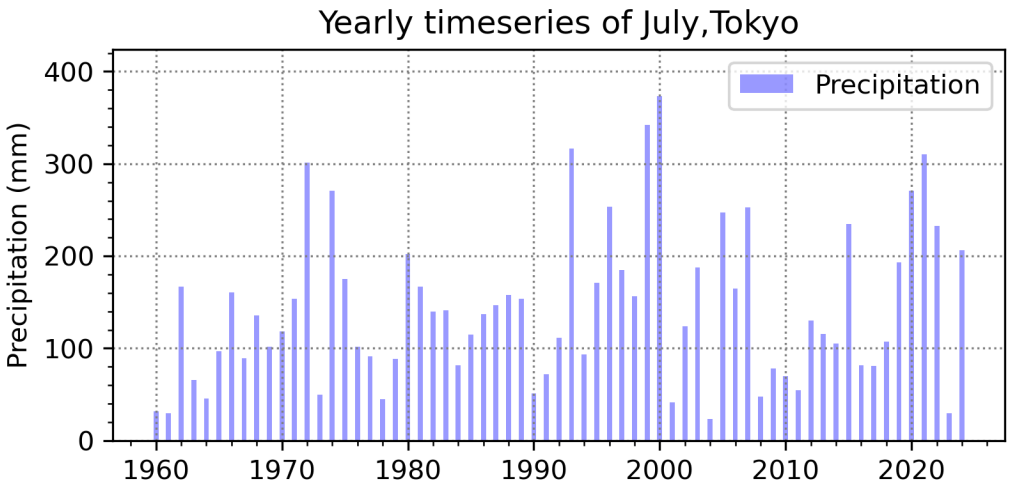


図４－２－１　東京のアメダス地点における７月積算降水量データの年時系列図

月の指定はこれまで通り 7 月です。このプログラムでは、最初に開始年と終了年を指定しています。

```
syear = 1960
eyear = 2024
month = "jul"
```

そのため、prep_i.loc でデータを取り出した際には、1960～2024 年の 7 月の積算降水量データが prep に取り出されます。

```
# AmedasStation.retrieve_mon メソッドを使い、降水量データを取得
prep_i = amedas.retrieve_mon("prep")
#  降水量(mm)データの取り出し
prep = prep_i.loc[syear:eyear, month]
```

降水量データに関しても、気温の場合と同様に、plt.ylim に渡す y 軸範囲の指定を自動化しています。下限は 0 mm とし、上限は math.ceil を用いて、降水量の最大値よりも 50 mm 以上大きい整数になるように調整しています。plt.ylabel で y 軸のラベルを設定します。ここで取得したデータは 7 月の積算降水量なので単位を mm にしていますが、ひと月当たりという意味でもあるので、mm/mon と表記しても良いでしょう。棒グラフを描く部分が plt.bar です。最初の引数は x 軸の値、2 番目の引数は y 軸の値で、それぞれ x 軸上における棒グラフの中心座標と棒グラフの高さに変換されます。棒グラフの横幅は、width=0.4 で指定しています。値は x 軸上の幅です。不透明度（alpha）とラベル（label）に付いては、plt.plot 同様です。

```
index = prep.index
plt.ylim([0, math.ceil(prep.max()) + 50])
plt.bar(index, prep, color='b', width=0.4, alpha=0.4, label='Precipitation')
plt.ylabel('Precipitation (mm)')
```

4.2.2 棒グラフを横に並べる

棒グラフを横に並べた図を見たとこもあると思います。図４－２－２のように夏季３ヶ月分（６～８月）のデータを横に並べる方法を考えてみます。作図に用いたプログラムが amedas_prep_3mon.py です。取り出す月を python のリストにしています。リストを定義する場合、リストの要素間をカンマで区切り、要素全体を[]で囲みます。参照する場合には、months[0]、months[1]、などのように要素番号を指定します。

```
months = ["jun", "jul", "aug"] #  月のリスト
```

先ほどと同様に取得した全期間の月別降水量データ（prep_i）を用いて、6〜8 月のデータを切り出します。プログラムの冒頭部分で、Numpy の呼び出しを行っています。Numpy（参照名 np）の np.arange(3)を使い、0〜2 まで 3 つの整数を出力します。for 文は python のループで for n in np.arange(3):で n に 0、1、2 の順に値を代入してループを回すことを意味します。ループ内では、prep という python のリストにデータを追記しています。上では[]でリストを定義していましたが、空のリストを作成することも可能で、prep = list()で行っています。リストは append というメソッドを持っており、prep.append(追記する内容)で、リストの最後に追記されます。空のリストに対して prep_i.loc で切り出したデータを順に追記したので、months[0]（6 月）のデータが 1 番目、months[1]（7 月）のデータが 2 番目、months[2]（8 月）のデータが 3 番目に入ります。

```
import numpy as np # Numpy 呼び出し（プログラム冒頭部分）

prep_i = amedas.retrieve_mon("prep") #  降水量データ取得
prep = list() #  空のリスト作成
for n in np.arange(3):
    prep.append(prep_i.loc[syear:eyear, months[n]]) #  リストに追記
```

他の言語でプログラムしてきた方にとっては、months に入れた文字列でも、prep_i.loc で切り出した Pandas の Series でも、同じリストとして扱われる仕様に違和感を感じるかもしれません。この辺りの曖昧さを許容している部分が、python の柔軟なところでもあります。

3 ヶ月分の棒グラフを同時に描くためプログラムを少し工夫し、先ほどの降水量リスト（prep）に対応させた作図オプションリスト STYLES を定義し、plt.bar に渡す際にリストの参照番号を変えることで、6 月、7 月、8 月を同時に描けるようにしています。その準備として、プログラムの最初で STYLES という作図オプションのリストを定義しています。これまでのリストより複雑で、リストの内部に python の辞書が入るという入れ子構造になっています。python の辞書は、{key0: value0, key1: value1}という形式か、dict(key1=value1, key2=value2)という形式で定義します。python の辞書について知らない場合

は、オプションとオプションに渡される値の組み合わせのように思っておいて下さい。1 番目の辞書が 6 月、2 番目の辞書が 7 月、3 番目の辞書が 8 月のオプション一覧で、それらが、STYLES というリストの 1 番目（STYLES[0]）から 3 番目（STYLES[2]）までに入力されます。

```
STYLES = [
    dict(label='Jun.', color='g', width=0.2, alpha=0.4),
    dict(label='Jul.', color='b', width=0.2, alpha=0.4),
    dict(label='Aug.', color='aqua', width=0.2, alpha=0.4)
]
```

棒グラフの作図部分に移ります。グラフを横に並べるには x 軸上の座標をずらす必要があるため、中心座標に対してのずれを off_x という変数にして、prep[n].index で得られた中心座標の値に加えています。初期値は off_x = -0.2 でループの最後で毎回 0.2 を加えています。棒グラフの幅を 0.2 に設定したので、年毎の目盛線を中心にして隙間なく 3 つの棒グラフが並びます。index = prep[n].index + off_x の計算は、index と prep[n].index が 1 次元配列、off_x が 0 次元のスカラーなので、他の言語では文法エラーになるような記法です。python の場合には、off_x の要素で埋められた 1 次元の配列を足すように解釈されています。

plt.bar には x 軸の値（index）と y 軸の値（prep[n]、ループ毎に n は変化）を渡す他、**STYLES[n]を渡しています。先ほど定義した STYLES のリストから **STYLES[0]、**STYLES[1]、**STYLES[2]の順に取り出したという意味で、これらは入れ子の内側に入っていた python の辞書です。plt は python の辞書をそのままオプションとして処理できるので（これまで行ってきた処理でも、内部では python の辞書として渡していました）、辞書を表す**STYLES[0]、**STYLES[1]、**STYLES[2]をオプションとして記述することができます。C 言語を学習してきた方は、この表記を見るとポインタと混同してしまうかもしれませんが、python では全くの別物です。

y 軸の範囲を計算するため max_y というリストを作り、6 月、7 月、8 月の降水量最大値を max_y.append(math.ceil(prep[n].max()))で追記しています。デ

ータの最大値を返す np.max を使い、np.max(max_y)+50 で 3 ヶ月の最大値よりも 50 mm 以上大きい整数になるように調整しています。

```
off_x = -0.2
max_y = list() #  最大値を保存するリスト
for n in np.arange(3):
    index = prep[n].index + off_x # off_x の分だけ横にずらす
    plt.bar(index, prep[n], **STYLES[n]) #  棒グラフ作図
    max_y.append(math.ceil(prep[n].max())) #  最大値を保存
    off_x += 0.2

plt.ylim([0, np.max(max_y)+50]) # y 軸の範囲
plt.ylabel('Precipitation (mm)') # y 軸のラベル
```

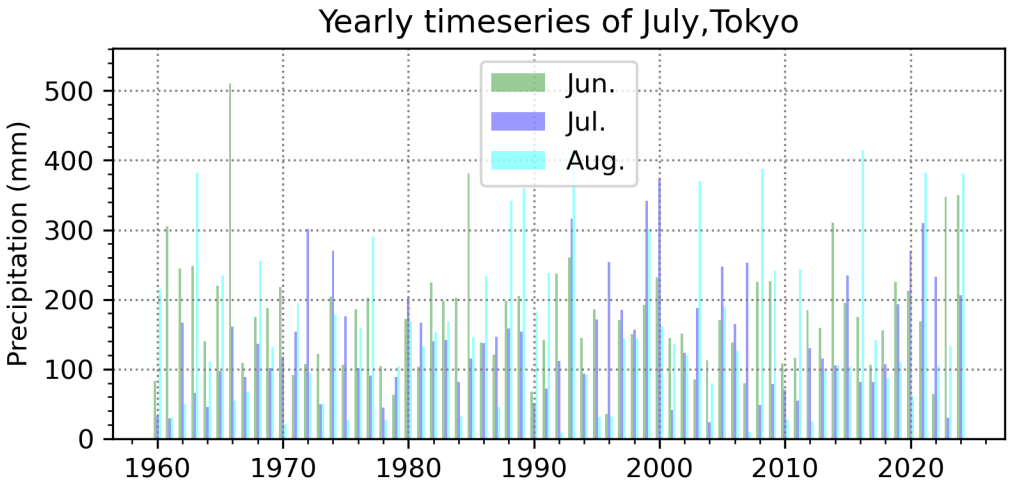


図４－２－２　東京のアメダス地点における 6～8 月積算降水量データのグラフを横に並べた

4.2.3 積み上げ棒グラフ

6〜8 月のデータを縦に積み上げて、図４－２－３のような 3 ヶ月積算量を出してみます。作図に用いたプログラムが amedas_prep_cum3mon.py です。作図オプションリスト STYLES は次のように書き換えています。

```
STYLES = [
    dict(label='Jun.', color='g', width=0.6, alpha=1.0, edgecolor='k', lw=1),
    dict(label='Jul.', color='b', width=0.6, alpha=1.0, edgecolor='k', lw=1),
    dict(label='Aug.', color='aqua', width=0.6, alpha=1.0, edgecolor='k', lw=1)
]
```

棒の幅を width=0.6 として、グラフの横幅を広くしました。また棒グラフの色（color）の他に、枠の色を表す edgecolor を新たに使っていて、黒色を指定しました。3 ヶ月積算したグラフの周囲に枠を付けて、1 つの棒グラフを 6 月、7 月、8 月に分割したように見せるためです。lw=1 は枠線の太さです。ここで alpha=1.0 と指定して、棒の色を不透明にしています。不透明度の設定は棒の色と枠線の色の両方に有効です。透明になっている場合、同じ色が重なった部分で色が濃くなるため、3 ヶ月分の棒グラフを積み上げると 6・7 月、7・8 月の境目で枠線が重なって色が濃くなり見栄えが悪いため、このようにしました（透明にした場合はどうなるか、実際に試してみて下さい）。

棒グラフの作図部分です。今度は x 軸上の同じ場所にグラフを積み上げるため、index は棒グラフの中心軸の値に固定します。棒グラフを作図する時に、y 軸上における開始位置の指定（bottom）が可能です。6 月の場合に 0、7 月の場合に 6 月の値、8 月の場合に 6・7 月の積算値を開始位置にすることで棒グラフの積み上げを行います。まず off_y = np.zeros(len(prep[0]))で初期開始位置を 0 に設定します。np.zeros(長さ)は、指定した長さまで 0 の値を入力した 1 次元配列を生成します。月毎のループの最後で off_y = off_y + add_y を行うことで、off_y に 6、7、8 月の降水量データを順番に足して行きます。add_y には降水量データ np.array(prep[n])が格納されています。np.array()は、Numpy の配列に変換することを意味していて、prep[n]の中身を 1 次元配列にしています。配列への足し込みを行う際、添字をスライスで指定しないことに違和感を感じるかもしれません。python では、こうした配列同士の演算を表記する際の曖昧さを

許容していて、off_y = off_y + add_y は off_y[:] = off_y[:] + add_y[:]と同じです。plt.bar で bottom=off_y としているため、前月までの積算値を y 軸上の開始位置とし、y 軸上で add_y の長さ分だけ上側に棒グラフを伸ばします。

```
off_y = np.zeros(len(prep[0])) #  棒グラフの初期開始位置（０）
for n in np.arange(3):
    index = prep[n].index # x 軸上の位置
    plt.bar(index, prep[n], bottom=off_y, **STYLES[n]) #  棒グラフ積み上げ
    add_y = np.array(prep[n]) #  足しこむ配列
    off_y = off_y + add_y #  棒グラフの開始位置再計算

plt.ylim([0, math.ceil(off_y.max())+100]) # y 軸の範囲
```

先ほどとは異なり、y 軸の範囲を指定する計算には off_y を用いています。この時点で 3 ヶ月分の積算値が off_y に格納されています。off_y の最大値を off_y.max()メソッドで取り出し、それよりも 100 mm 以上大きな整数を y 軸の上限値としています。

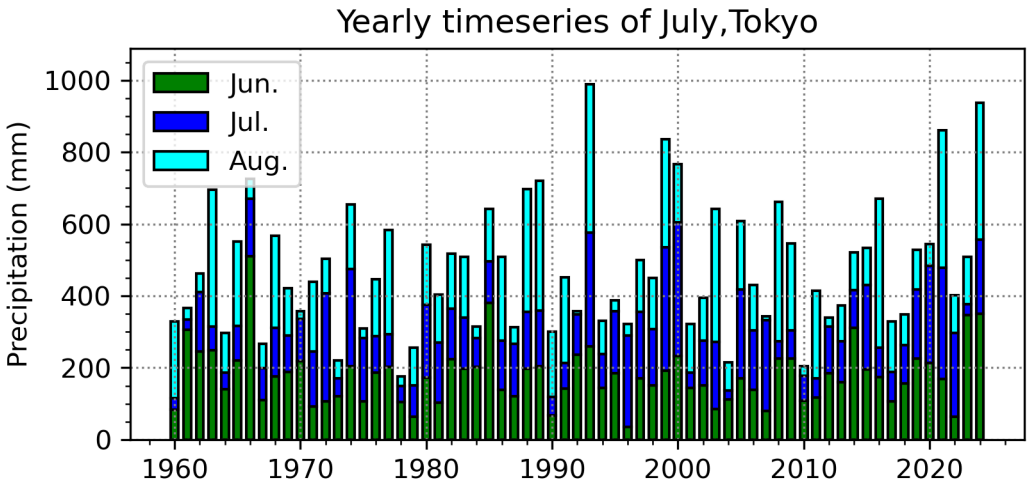


図４－２－３　東京のアメダス地点における 6～8 月降水量を縦に積み上げた

4.2.4 棒グラフにエラーバーを付ける

平均値を棒グラフで、標準偏差の範囲をヒゲで同時に表示したグラフを見たことがあると思います。同じ東京の降水量データを使い、期間 1960～2024 年で 7 月の平均降水量とその標準偏差をプロットしたものが図４－２－４です。

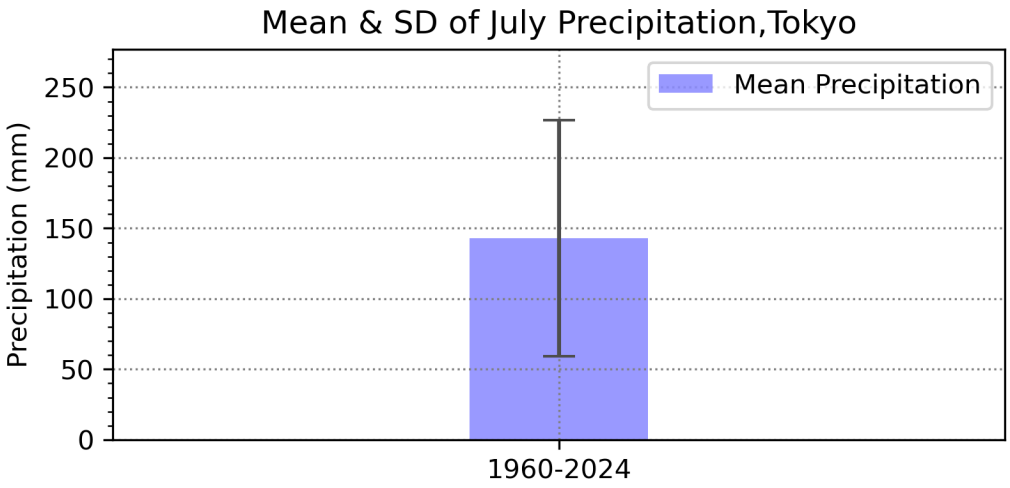


図４－２－４　東京のアメダス地点における 7 月の平均降水量と標準偏差の範囲（1960～2023 年）

作図には amedas_prep_mean+sd_jul.py を使いました。このプログラムでは、最初の棒グラフ作成時と同様、syear=1960、eyear=2024、sta="Tokyo"、month="jul"、としました。データを取り出した後、平均、標準偏差の計算を行なっています。ここでも、取り出された prep は Pandas の Series になっていて、Series では Numpy の ndarray のインスタンスメソッドが使える状態です。そのため、prep.mean()で算術平均、prep.std()で標準偏差を直接計算可能です。他にも様々な統計処理のツールがあるので、表４－２－１にまとめておきます。今扱っている 1 次元配列の場合は引数なしの形ですが、2 次元以上の配列では、prep.mean(axis=0)や prep.mean(0)のように軸を明示的に指定する必要が出てきます（いずれも 0 番目の軸を指定する場合）。なお表４－２－１のメソッドは、np.mean(**配列名**)のように使っても、同様に配列の算術平均を返します。

なお Pandas の Series では、Numpy に含まれないものも追加で実装されているようです。表４－２－１の median、skew、kurtosis は Numpy の ndarray 単独で使うとエラーになりますが、Pandas の Series では利用可能です。

```
prep = prep_i.loc[syear:eyear, month] #  データの取り出し
prepm = prep.mean() #  算術平均
prepsd = prep.std() #  標準偏差
```

表４－２－１　Pandas の Series や Numpy の ndarray で使用可能な統計処理のインスタンスメソッド一覧（data が ndarray でインスタンス）。skew、kurtosis は、Pandas の Series のみで利用可能

data.メソッド()	説明
data.mean()	算術平均（mean）を返す
data.median()	中央値（median）を返す
data.std()	標準偏差（standard deviation）を返す
data.var()	分散（variance）を返す
data.skew()	歪度（skewness）を返す、Pandasのみ
data. kurtosis()	尖度（kurtosis）を返す、Pandasのみ
data.max()	最大値を返す
data.min()	最小値を返す
data.argmax()	最大値を持つ要素のindexを返す
data.argmin()	最小値を持つ要素のindexを返す
data.cumsum()	累積和を返す（最初からその番号までの和）
data.cumprod()	累積積を返す（最初からその番号までの積）

作図範囲の設定方法です。x 軸を plt.xlim を使い 0～1 の範囲に設定して、x=1 の場所に棒グラフを描こうとしています。x 軸上の座標を index=[1]のようにリストにしています。これまで同様、plt.ylim で y 軸の範囲を設定し、平均+分散の値よりも y 軸上で 50 より大きな整数に設定されます。

```
index = [1] # x 軸の座標で指定
plt.xlim([0, 2]) # x 軸の範囲
plt.ylim([0, math.ceil(prepm+prepsd) + 50]) # y 軸の範囲
```

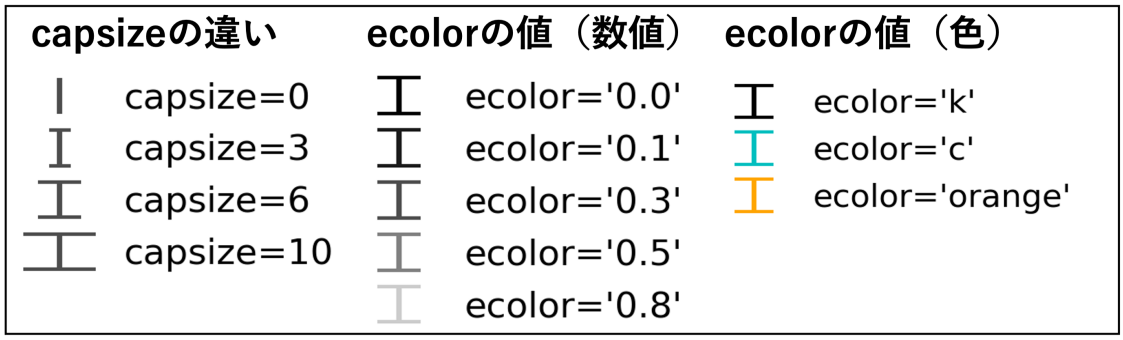
棒グラフを作図する部分では、x 軸のデータとして index、y 軸のデータとして prepm を渡しています。そのため、先ほど設定した x=1 の場所に降水量平均値の棒グラフが作成されます。グラフの色（color）、グラフの幅（width）、不透明度（alpha）、ラベル（label）の設定方法はこれまで同様です。

エラーバーに用いる標準偏差データを渡すのが、yerr=prepsd の部分です。平均値の棒グラフの上端から下側に 1 標準偏差、上側に 1 標準偏差のエラーバーが描かれます。エラーバーの書式を設定するのが error_kw=error_config の部分です。設定可能なものは、ヒゲのキャップサイズ（capsize）とエラーバーの色や透明度（ecolor）です。error_config = {'ecolor': '0.3', 'capsize': 6}のように、辞書として渡します。capsize の大きさはポイントで指定し整数値のみ指定可です。デフォルト値は 0 で、数字を増やすほど横に長いキャップになっていきます（表４－２－２左）。ecolor の設定は文字列で行うので、0.3 ではなく'0.3'にしないとエラーになります。ecolor が 0.3 というのは、透明度が 0.3 に対応します。数値を指定した場合には黒色に対して透明度が設定されるので、ecolor が 0.0 で黒になり、ecolor が 1.0 にかけて徐々に薄い灰色になっていきます（表４－２－２中）。不透明度 alpha の設定とは逆です。色の設定もできて、その場合には'ecolor': 'c'（水色）のように行います（表４－２－２右）。ecolor の設定がないときは黒色です。

```
error_config = {'ecolor': '0.3', 'capsize': 6}
plt.bar(index, prepm, yerr=prepsd, error_kw=error_config, ¥
        color='b', width=0.4, alpha=0.4, label='Mean Precipitation')
```

なお、エラーバーのラベルは error_config = {'ecolor': '0.3', 'capsize': 6, 'label': 'ecolor=0.3'}のように設定可能ですが、棒グラフのラベルとは別に凡例が出るため、エラーバー単独で使う場合を除いて実用的ではありません。

表４－２－２　error_kw に渡すことができる capsize と ecolor オプションの効果



x 軸の目盛り線ラベルには、開始年から終了年を入れました。xlabel に 1960-2024 という文字列を設定した後に、ax.set_xticklabels([**ラベルのリスト**])でラベルを設定します。ラベルに対応する目盛線の値も設定が必要で、ax.set_xticks([**目盛り線の値のリスト**])で行います。このようにリストで与えているので、棒グラフが複数になった場合はリストの要素を増やすだけで同様の作図を行うことが可能です。

```
# x 軸の目盛り
xlabel = str(syear) + "-" + str(eyear)
ax.set_xticks([1]) #  目盛り線
ax.set_xticklabels([xlabel]) #  目盛り線ラベル
```

4.2.5 平均と標準偏差のグラフを月毎に並べる

では、実際に複数のグラフを並べてみましょう。先ほどは 7 月のみを作図しましたが、1～12 月の各月のデータを作図します（図４－２－５）。作図に用いるプログラムは、amedas_prep_mean+sd_mon.py です。Numpy を使うため、プログラムの最初で import し、np で参照できるようにしています。

```
import numpy as np
```

次のように計算を行う期間（1960～2024 年）を設定し、取り出したい月（12 ヶ月分）をリストにしています。

```
syear = 1960
eyear = 2024
months = [
      "jan", "feb", "mar", "apr", "may", "jun", "jul", "aug", "sep", "oct", ¥
      "nov", "dec"
] # 月のリスト
```

これまで同様に AmedasStation Class で取り出した降水量データを prep_i に入力します。prep_i から月毎に降水量データを取り出す部分は、ループの中に入っています。ここで作成しているのが、月間降水量の平均値リスト（prepm）、降水量の標準偏差のリスト（prepsd）、y 軸上の最大値リスト（max_y、平均と分散の和）、x 軸の目盛線ラベルのリスト（xlabel）です。いずれもループの直前で空のリストを定義し、月毎に計算した値を append で追加しています。

月間降水量を取得する部分は、prep = prep_i.loc[syear:eyear, months[n]]です。1 次元目で syear から eyear までの範囲のスライスを指定し、2 次元目で先ほど定義した月のリストのうち該当月名をスカラーで指定することで、該当月の月間降水量を syear から eyear までの Series として返却しています。

prep から算術平均と標準偏差を計算する部分が prep.mean()、prep.std()で、各月毎の結果が prepm.append(prep.mean())と prepsd.append(prep.std())のメソッドで prepm、prepsd に順次追加されていきます。グラフの y 軸上限を決めるために計算しているのが max_y です。ヒゲグラフの上限は平均＋1 標準偏差なので、それを超える整数を math.ceil(prep.mean()+prep.std())で算出します。max_y.append で各月毎に追加されていきます。

目盛線のラベルには、Jan、Feb のような最初を大文字にした月を表示しようとしています。文字列のメソッド capitalize()は最初の文字を大文字に変えるメソッドなので、str(months[n]).capitalize()で jan、feb のような表記を Jan、Feb のように変更します。xlabel.append で各月毎に追加されていきます。

```
prepm = list()
prepsd = list()
max_y = list()
xlabel = list()
for n in np.arange(len(months)):
    prep = prep_i.loc[syear:eyear,months[n]] #  月間降水量取得
    # prepm、prepsd、max_y、xlabel リストに追加
    prepm.append(prep.mean()) #  算術平均
    prepsd.append(prep.std()) #  標準偏差
    max_y.append(math.ceil(prep.mean()+prep.std())) # y 軸上の最大値
    xlabel.append(str(months[n]).capitalize()) # x 軸の目盛り線のラベル
```

次に作図部分です。x 軸の 1、2、…、12 の値の場所に棒グラフを描こうとしているので、index には 1.0～12.0 の 1.0 刻みの値（index=[0.0, 1.0, 2.0, …, 12.0]のようなリスト）、x 軸の範囲は下限 0.5、上限 12.5 が入るようにしています。ここで、len(months)は 12 を返します。y 軸の範囲は、先ほど作成した max_y を使い算出します。各月に関してヒゲグラフの上限となる y 軸上の値が入っているので、np.max(max_y)で最大値を計算し、それよりも 50 mm 大きな値を y 軸の範囲の上限としました。

plt.bar のオプション表記は 7 月のみの時と同じですが、渡されている引数の index、prepm、prepsd は、いずれも 1～12 月に対応する 12 個の要素を持ったリストになっています。7 月の場合、index は要素 1 個のリスト（index=[1]）、prepm、prepsd は 0 次元の浮動小数点数でした。このように、引数として与えることが可能な形式に曖昧さを許容した（python 的な？）仕様となっています。

```
index = np.arange(len(months)) + 1.0 # x 軸の index
plt.xlim([0.5, len(months)+0.5]) # x 軸の範囲
plt.ylim([0, np.max(max_y)+50]) # y 軸の範囲
error_config = {'ecolor': '0.3', 'capsize': 6}
plt.bar(index, prepm, yerr=prepsd, error_kw=error_config, ¥
        color='b', width=0.4, alpha=0.4, label='Mean Precipitation')
```

この状態では index の値を x 軸の値とするので、x 軸には 1、2、…、12 の数字が表示されます。これでも月と分かるのですが、先ほど xlabel に Jan、Feb などの文字列リストを設定したので、これらの文字列で置き換える方法を考えてみます。軸の目盛線とラベルを任意の刻み、任意の文字列で置き換えるメソッドが、set_xticks(**x 軸の目盛線を付ける値のリスト**)、set_xticklabels(**x 軸の目盛線ラベルのリスト**)です。x 軸の目盛線を 1、2、…、12 の場所に付けたいので、np.arange(len(months))+1 で生成した値を渡します。目盛線に対応するラベルリストとして、先ほどの xlabel を使います。なお、set_xticks と set_xticklabels でリストの要素数が違うとエラーになります。

```
ax.set_xticks(np.arange(len(months))+1) # x 軸の目盛線
ax.set_xticklabels(xlabel) # x 軸の目盛線ラベル
```

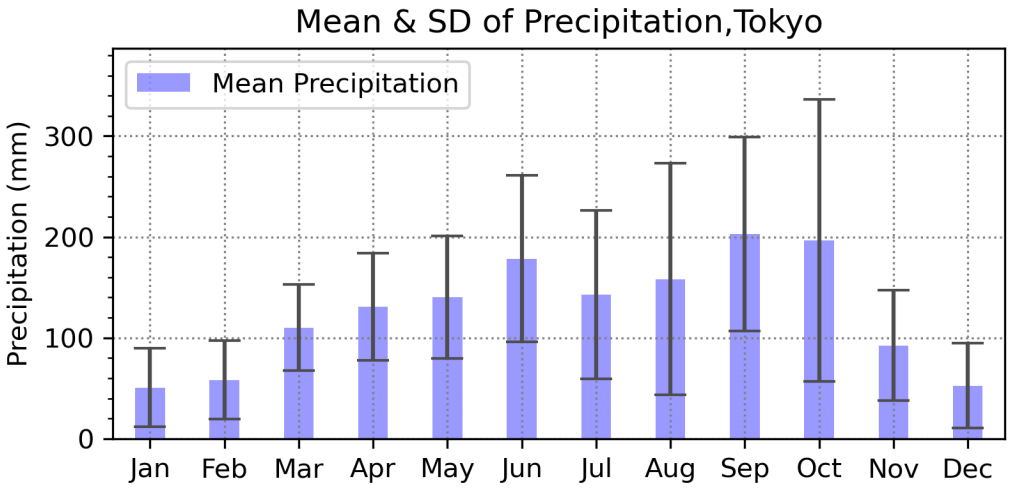


図 4 － 2 － 5　東京のアメダス地点における降水量の平均と標準偏差の季節変化（1960～2024 年）

4.2.6 各月に複数の棒グラフを並べる

これまでは 1960～2024 年の 1 期間だけでしたが、複数の期間平均を並べて降水量の平均と標準偏差を比較したいこともあるかと思います。そのような比較を行うため、異なる期間のデータから計算した棒グラフを横に並べる方法を考えます。このような作図を行う際には、期間平均の操作を複数回行う必要があり、その部分を分離した方がシンプルなプログラムになります。

その準備として、先ほどのプログラムの期間平均部分を関数にして外に出すことを考えます。amedas_prep_mean+sd_mon2.py は、先ほどと同じ図 4－2－5 を生成するプログラムですが、全体の制御と作図部分を main 関数、期間平均の操作を get_data 関数のように 2 つの関数を定義して処理を分割しました。

プログラム中で def **関数名():**で始まっているものは python の関数です。関数にした場合には、プログラム本体を実行した場合に実行される領域から外れるため、明示的に他から呼び出されるまでは実行されません。なお関数の前で定義されている syears などの変数はグローバル変数で、関数の内外問わず参照可能です。関数の中で定義したものはローカル変数で、関数の中だけで操作可能です。関数の外からは参照できないので、たとえ同じ名前の変数が関数外にあったとしても、関数外の値を変えることはできません。一方で、グローバル変数と同じ名前の変数は関数内でもグローバル変数になるため、ローカル変数のつもりで値を変更すると全体に影響を与えてしまいます。

main 関数はシンプルな記述の仕方で、引数は空で関数の戻り値はありません。プログラムの最後に main()としているので、その場所で main が呼び出され、main 関数内に記述されている処理が行われます。

python の関数は引数や戻り値の指定も可能です。get_data 関数では引数や戻り値を使ってローカル変数を別の関数に受け渡しています。引数が必要な場合は def **関数名（引数 1，引数 2，…）**のような表記を、戻り値が必要な場合は、**return（戻り値 1，戻り値 2，…）**のような表記を使います。get_data 関数は main 関数の中で呼び出します。get_data 関数では、def get_data(syear, eyear, prep_i):のように開始年（整数）、終了年（整数）、降水量データ（Pandas の DataFrame 形式）の 3 つを引数として渡すようにしました。get_data の戻り値は、prepm、prepsd、max_y、xlabel の 4 つのリストです。関数内では前節と同様の計算を行なっています。

```
def get_data(syear, eyear, prep_i):
    # prepm、prepsd、max_y、xlabel の部分は同じなので省略
    …
    return (prepm, prepsd, max_y, xlabel) #  結果を返却
```

プログラムの最初では、開始年と終了年をリスト形式で定義し、main 関数で用いるようにしました。

```
syears = [1960 ] #  開始年リスト
eyears = [2024 ] #  終了年リスト
```

get_data 関数を呼び出す部分では、次のように 3 つの引数を渡しています。1、2 番目の引数 syears[0]、eyears[0]は、それぞれ開始年リストの 1 番目、終了年リストの 1 番目の要素です。1 つの要素しか含まないリストなので、要素番号は 0 になっています。開始年、終了年の要素数を増やした場合には、[0]の部分を変えて、必要な要素を取り出すようにすれば良いでしょう。3 番目の引数 prep_i には retrieve_mon で取得してきた DataFrame をそのまま渡します。

関数の戻り値は、prepm、prepsd、max_y、xlabel にそれぞれ入力されます。get_data 関数の中でリストと定義されているので、これら 4 つは自動的にリストになります。今は関数の中と外で同じ名前の変数にしていますが、名前が異なっても問題ありません。

```
prepm, prepsd, max_y, xlabel = get_data(syears[0], eyears[0], prep_i)
```

関数の準備ができたところで、この get_data 関数を開始年と終了年の異なる複数の平均期間のループの中で呼び出すように書き換えます（amedas_prep_mean+sd_mon3.py）。このプログラムを使い、複数の期間平均の棒グラフを並べたものが図４－２－６です。

開始年と終了年のリストは、それぞれ 3 つの要素を含むようにしました。これらの要素をループの中で順に syear、eyear として用います。

```
syears = [1881, 1931, 1981] #  開始年リスト
eyears = [1910, 1960, 2010] #  終了年リスト
```

それぞれの平均期間に対応させ、作図オプションの色を変えています。

```
STYLES = [
    dict(color='b', alpha=0.4),
    dict(color='g', alpha=0.4),
    dict(color='r', alpha=0.4)
]
```

全ての棒グラフに共通のオプションとして、灰色でキャップサイズが 3 ポイントのヒゲグラフが付くように、error_config を設定しています。棒グラフの幅は x 軸上で 0.25 の長さに設定しています。index_i には、月に対応する x 軸上の座標（0、1、…、12）を格納しています。

```
index_i = np.arange(len(months)) + 1.0 # x 軸上の中心座標
error_config = {'ecolor': '0.3', 'capsize': 3}
bar_width = 0.25 # x 軸上の棒グラフの幅
```

棒グラフの幅と対応させて、3 つの棒グラフの位置を決めます。棒グラフの x 軸上の中心座標は、index = index_i + off_x で計算していて、off_x の初期値は -0.3 でループの最後に 0.3 毎足されていきます。月毎の目盛線の場所に 2 番目の棒グラフが配置され、その前後に 0.05 の隙間を開けて 1 番目と 3 番目の棒グラフが配置されます。

ループの部分では、for syear, eyear in zip(syears, eyears):を使っています。python に慣れていないと見たことがない記法かもしれませんが、このように zip(オブジェクト 1、オブジェクト 2、…)を使うと、syears[0]、eyears[0]の組み合わせ、syears[1]、eyears[1]の組み合わせ、…のように、同じ番号の要素をまとめて渡すことができます。このように取り出した、開始年、終了年の組み合わせを使い、bar_label に"開始年-終了年"の文字列を設定します。

データ取得部分（get_data 関数）の引数にも、syear、eyear を渡します。戻り値の記法は max_y_i だけ違います。これは全ての棒グラフで y 軸の最大値を計算するためで、ループの前に max_y を空のリストで定義し、データを取得する毎に max_y.append(max_y_i)で最大値を保存します。

棒グラフの作図部分では、先ほど定義した**STYLES[n]を渡します。n の値はループの前で 0 にして、ループの最後で 1 加えるようにしているので、開始年、終了年のペアが変わる毎に棒グラフの色が変わるようになります。

```
max_y = list()
off_x = -0.3
n = 0
for syear, eyear in zip(syears, eyears):
    bar_label = str(syear) + "-" + str(eyear) #  開始年-終了年
    #  データの取得
    prepm, prepsd, max_y_i, xlabel = get_data(syear, eyear, prep_i)
    max_y.append(max_y_i) # y 軸データ最大値
    index = index_i + off_x # x 軸の値
    #  棒グラフ
    plt.bar(index, prepm, yerr=prepsd, error_kw=error_config, ¥
            width=bar_width, label=bar_label, **STYLES[n])
    off_x += 0.3
    n += 1
```

x 軸の範囲設定は、これまで同様 0.5〜12.5 です。y 軸の最大値は、保存した max_y の 3 要素のうちの最大値を np.max で計算し、それに 50 mm 加えたものです。

```
plt.xlim([0.5, len(months) + 0.5])
plt.ylim([0, np.max(max_y) + 50])
```

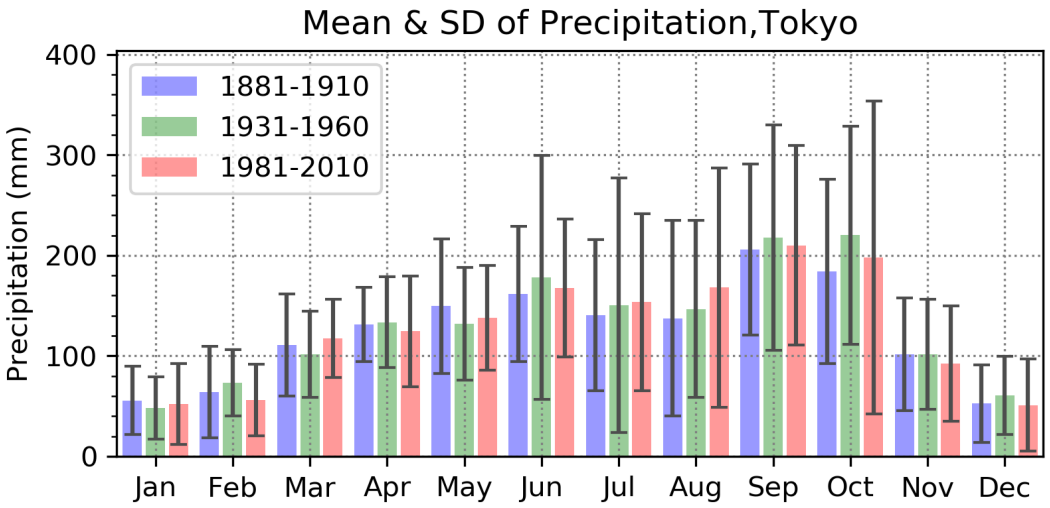


図４－２－６　東京のアメダス地点における降水量の平均と標準偏差の季節変化を 1881～1910 年、1931～1960 年、1981～2010 年の期間で並べた

4.2.7 棒グラフにハッチを付ける

これまでの棒グラフは、グラフの中を色で塗り潰していましたが、白黒にしてハッチのパターンで区別したいこともあるでしょう。先ほどの棒グラフにハッチを付けるプログラムが amedas_prep_mean+sd_mon_hatch.py です。

先ほどのプログラムから作図オプション（STYLES）のみ変更しています。棒グラフを描く際のオプションとして、ハッチを付ける hatch が追加されました。hatch='//'は片方の斜線、'xx'は両方の斜線、'oo'は丸のパターンです。ハッチを付ける場合は color を指定しても黒色になるので、color の指定は行いません。alpha=1.0 として不透明にしました。棒グラフを塗りつぶすオプション（fill）は None にしています（デフォルトの False が適用）。図４－２－７のように、棒グラフの中身がハッチに置き換わりました。

```
STYLES = [
    dict(alpha=1.0, hatch='//', fill=None),
    dict(alpha=1.0, hatch='xx', fill=None),
    dict(alpha=1.0, hatch='oo', fill=None)
]
```

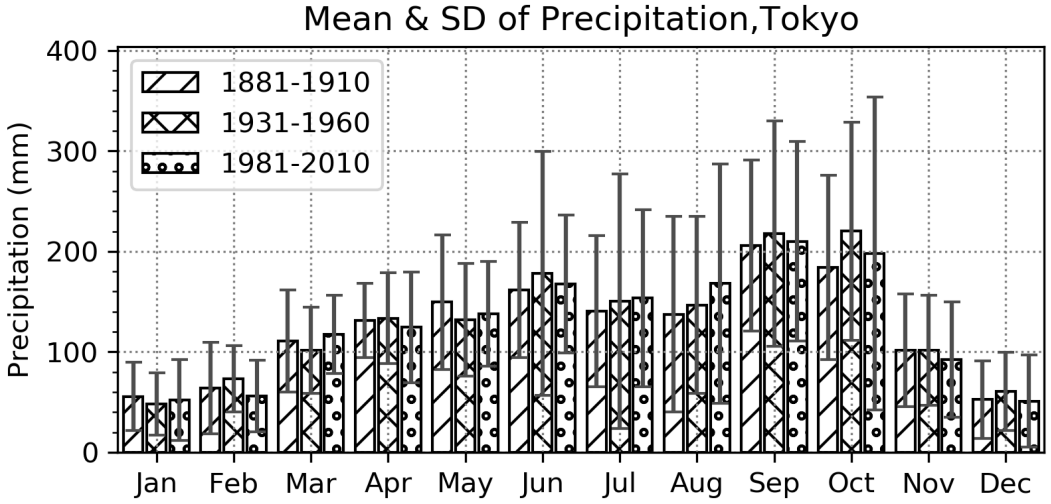


図４－２－７　図４－２－６の棒グラフをハッチで描いた

ハッチのパターンとして代表的なものを表４－２－３に示しておきます。

表４－２－３　ハッチのパターン

hatch='/'	
hatch='//'	hatch='..'
hatch='///'	hatch='...'
hatch='x'	hatch='oo'
hatch='xx'	hatch='OO'
hatch='xxx'	hatch='++'
hatch='--'	hatch='||'
hatch='---'	hatch='**'

ハッチを付けた棒グラフに色を付けることもできます。先ほどのプログラムの STYLES を変更して fill=True にします。color='b', alpha=0.4 のような色と不透明度の設定を戻します。

```
STYLES = [
    dict(color='b', alpha=0.4, hatch='//', fill=True),
    dict(color='g', alpha=0.4, hatch='xx', fill=True),
    dict(color='r', alpha=0.4, hatch='oo', fill=True)
]
```

作図を行うプログラムが、amedas_prep_mean+sd_mon_hatch2.py で、図４－２－８のように色付きの棒グラフに変わります。

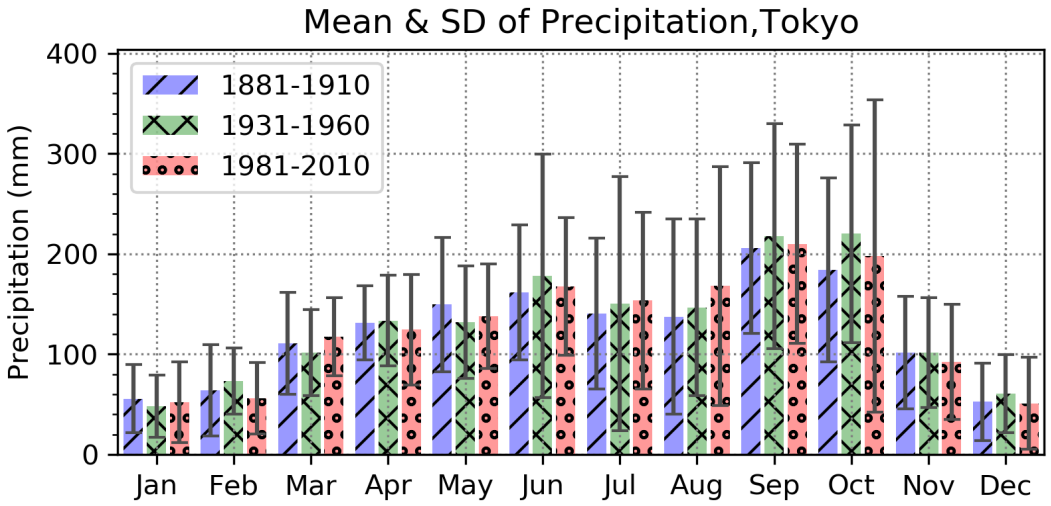


図４－２－８　図４－２－７の棒グラフに色を付けた

なお edgecolor を指定すると、棒の枠線に色を付けることができます（図４－２－９）。作図には、amedas_prep_mean+sd_mon_hatch3.py を使いました。枠線をつけるため、次のように edgecolor='k'のオプションを追加しました。edgecolor を有効にすると、なぜか alpha=0.4 がハッチにも適用されています。ハッチの色を黒にしておくには、alpha=1.0 にするしかないようです。

```
STYLES = [
    dict(color='b', alpha=0.4, hatch='//', fill=True, edgecolor='k'),
    dict(color='g', alpha=0.4, hatch='xx', fill=True, edgecolor='k'),
    dict(color='r', alpha=0.4, hatch='oo', fill=True, edgecolor='k')
]
```

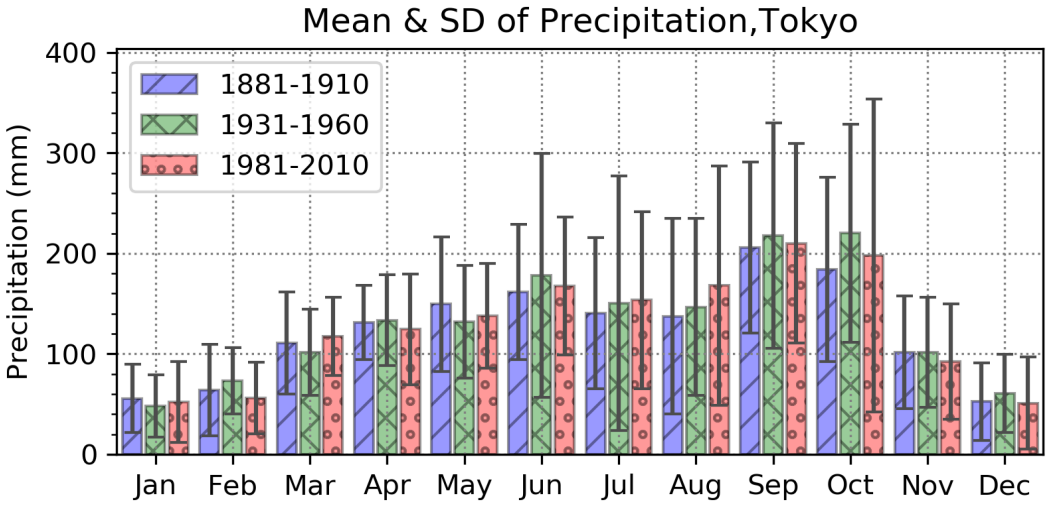


図４－２－９　棒グラフの枠線に色を付けた

最後に、棒グラフで使用できるオプション一覧を表にまとめておきます。これまで使ってきた plt.bar の代わりに plt.barh を使うと横棒を描くことが可能です。

表４－２－４　matplotlib の pyplot.bar で使用可能なオプション一覧

オプション	説明
x (必須)	x軸上の座標の配列
height or y (必須)	棒の高さ（y軸上の長さ）
width	棒の幅 、デフォルト値：0.8
bottom	下側の余白（y軸上の開始位置）
color	棒の色
edgecolor	棒の枠線の色
linewidth	棒の枠線の太さ
tick_label	x軸のラベル
xerr	x軸方向の誤差棒（誤差範囲）の数値または配列
yerr	y軸方向の誤差棒（誤差範囲）の数値または配列
ecolor	エラーバーの色を値または配列で指定
capsize	エラーバーの傘のサイズ
align	棒の位置、'edge'（垂直方向の場合:左端, 水平方向の場合:下端） 'center'（中央）、デフォルト値：'edge'
log	Trueで対数目盛り、デフォルト値：False
fill	塗り潰す色、デフォルト値：False
hatch	ハッチのパターン、デフォルト値：None
alpha	不透明度、デフォルト値：1.0
label	凡例を付ける場合、デフォルト値：None

使用方法：plt.bar(x,y)（縦棒）、plt.barh(x, y)（横棒）

ハッチを付ける別の方法もあるので紹介しておきます。パターンを貼り付ける matplotlib. patches を使い、矩形にハッチを付けたパターンを重ねます。それを行うプログラムが、amedas_prep_mean+sd_mon_patch.py です。まず matplotlib. patches をインポートします。

```
import matplotlib.patches as patches
```

ハッチ用のスタイルを定義しておきます。hatch='/'や'//'が斜線、'o'は丸のパターンです。全て黒色で描きます。

```
STYLESP = [
    dict(fill=False, hatch='/', color='k'),
    dict(fill=False, hatch='//', color='k'),
    dict(fill=False, hatch='o', color='k')
]
```

棒グラフを描いた後でハッチを付けたものが図４－２－10 です。ax.add_patch を使いパターンを追加します。patches.Rectangle が矩形のパターンで、最初の引数として開始位置の x 座標、y 座標を与えます。開始位置の x 座標（x1）は棒グラフの中心から半分の幅だけ引いた位置、開始位置の y 座標（y1）は 0 です。パッチの幅（width）は棒グラフの幅と同じ bar_width、高さ（height）は棒グラフの高さと同じ prepm の値です。patches.Rectangle は STYLESP[n]も引数に取っており、先ほど定義したパッチのパターンが左から順に適用されているのが分かります。(x1, y1)と括弧の中に入れているのは、この引数が python のタプルとして渡されるためです。タプルはリストや辞書とは違い、一度定義したら中身を変更できないオブジェクトです（リストのように中身を後から変更できるものをミュータブル、タプルのように変更できないものをイミュータブルと呼んでいます）。

```
for x, y in zip(index, prepm):
    x1 = x - bar_width/2
    y1 = 0
    ax.add_patch(patches.Rectangle((x1, y1), width=bar_width, ¥
                    height=y, **(STYLESP[n])))
```

この方法では、ハッチを描いても凡例に反映させることはできません。今後のアップデートが待たれるところです。

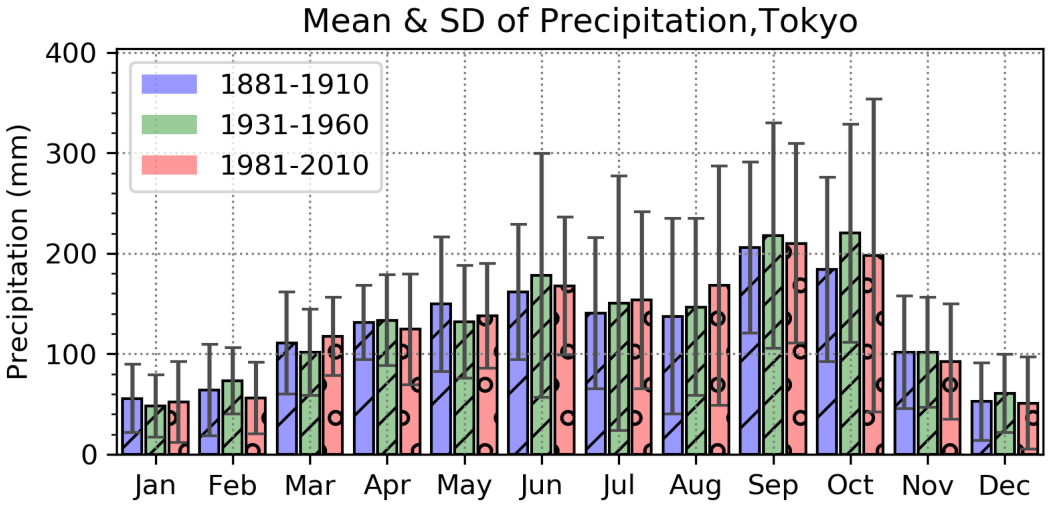


図４－２－10　図４－２－６に patches でハッチを付けたが、凡例には反映されない

4.3 2 軸グラフ

4.3.1 棒グラフと一緒に折れ線グラフを描く

これまでは棒グラフと折れ線グラフは単独で作成してきましたが、実用上は y 軸の片側を気温、片側を降水量にした 2 軸グラフを作成したいこともあるかと思います。2 軸グラフを作成する際、x 軸を共有する方法と凡例を適切に表示する方法にテクニックが必要ですので、ここで解説しておきます。これまでに使った月毎の降水量を作図する amedas_prep_mean+sd_mon.py に気温の折れ線グラフも加えることを考えます（amedas_prep+temp_mon.py)。このプログラムで作成した 2 軸グラフが図 4－3－1 です。Jupyter Notebook で作図すると、画面ではラベルが左上に集まることがありますが、保存されたファイルを開けると図 4－3－1 と同じになっています。

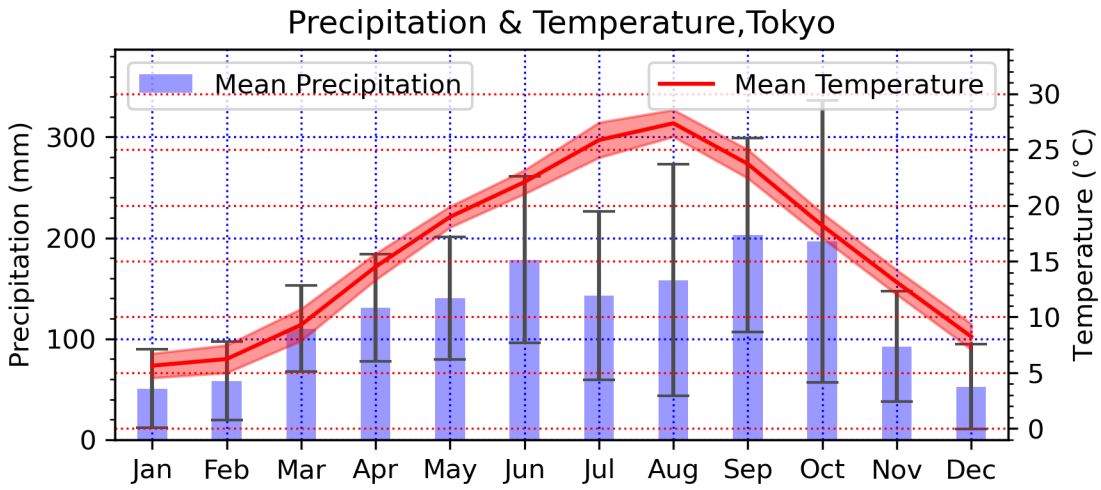


図 4 － 3 － 1　図 4 － 2 － 5 に東京のアメダス地点における平均気温を折れ線グラフで重ね、1 標準偏差の範囲を塗り潰した

降水量と平均気温のデータを取得して、それぞれ prep_i、tave_i に入力しておきます。

```
# AmedasStation.retrieve_mon メソッドを使い、降水量データを取得
prep_i = amedas.retrieve_mon("prep")
# AmedasStation.retrieve_mon メソッドを使い、平均気温データを取得
tave_i = amedas.retrieve_mon("tave")
```

降水量の平均と標準偏差 prepm、prepsd と x 軸のラベル xlabel の各月データを生成する部分はこれまでと同じです。気温については、平均気温（tempm）と平均から標準偏差を引いたもの（tmin）、平均に標準偏差を加えたもの（tmax）を気温データの塗り潰し範囲として計算しています。y 軸の範囲を設定するために降水量について y 軸上の最大値（max_yl）、気温について y 軸上の最小値（min_yr）と最大値（max_yr）を計算します。降水量の最小値は 0 に設定するので計算していません。

```
tempm = list() #  平均気温
tmin = list() #  塗り潰し範囲の下限
tmax = list() #  塗り潰し範囲の上限
max_yl = list() # y 軸上の最大値（左側、降水量）
max_yr = list() # y 軸上の最大値（右側、気温）
min_yr = list() # y 軸上の最小値（右側、気温）
for n in np.arange(len(months)):
    temp = tave_i.loc[syear:eyear,months[n]]
    tmin.append(temp.mean()-temp.std()) #  平均気温-標準偏差
    tmax.append(temp.mean()+temp.std()) #  平均気温+標準偏差
    max_yl.append(math.ceil(prep.mean()+prep.std())) #  平均降水量+標準偏差
    max_yr.append(math.ceil(temp.mean()+temp.std())) #  平均気温+標準偏差
    min_yr.append(math.floor(temp.mean()-temp.std())) #  平均気温-標準偏差
```

サブプロット（ax1）を作成して降水量の棒グラフを描きます。ここまでは棒グラフのみを作成する場合と同じです。y 軸上の最大値 max_yl に格納された最大の値よりも 50 mm 大きい値を y 軸の最大値としました。

```
ax1 = fig.add_subplot(1, 1, 1) #  サブプロットの作成
plt.xlim([0.5, len(months) + 0.5]) # x 軸の範囲（左側）
plt.ylim([0, np.max(max_yl) + 50])  # y 軸の範囲（左側）
error_config = {'ecolor': '0.3', 'capsize': 6}
plt.bar(index, prepm, yerr=prepsd, error_kw=error_config, ¥
        color='b', width=0.4, alpha=0.4, label='Mean Precipitation')
```

降水量については、グリッド線を青色で描きました。

```
plt.grid(color='b', ls=':') #  グリッド線を描く（青色）
```

ここでx軸を共有するテクニックが出てきます。棒グラフを描いてきたax1とx軸を共有するようなサブプロットax2をax1.twinx()メソッドを使って定義します。このように定義してあげると、右側のy軸に目盛りやラベルが付いたもう一つのグラフを描くことが可能になります。

```
ax2=ax1.twinx() # 2 つのプロットを関連付ける（x 軸の共有）
```

ここで新たなサブプロットを定義したので、この後でpltを使う場合にはax2に描画しようとします。そのため、次のy軸範囲の設定は右側の軸に対して適用されます。y軸上の最小値min_yrと最大値max_yrに格納された値から、それぞれ最小値、最大値を計算し、それよりも5度以下、5度以上までをy軸の範囲としました。

```
plt.ylim([np.min(min_yr)-5, np.max(max_yr)+5]) # y 軸の範囲（右側）
```

折れ線グラフの作成（plt.plot）と1標準偏差の範囲を塗り潰す部分（plt.fill_between）です。これらのpltは、ax2と書いても同じです。塗り潰しでは先ほど計算したtmin、tmaxを使います。plt.ylabelでも右側のy軸ラベルが設定されています。

```
plt.plot(index, tempm, color='r', ls='-', label='Mean Temperature') #折れ線
plt.fill_between(index, tmin, tmax, color='r', alpha=0.4) #  塗り潰し
plt.ylabel('Temperature ($^{¥circ}$C)') # y 軸のラベル
```

気温については、グリッド線を赤色で描き、降水量と区別できるようにしました。

```
plt.grid(color='b', ls=':') #  グリッド線を描く（青色）
```

最後に凡例を適切に表示するテクニックが出てきます。この方法では、降水量と気温の凡例を別々の場所に表示します。ax1 の legend メソッドを使うと棒グラフを描いた際の凡例を、ax2 の legend メソッドを使うと折れ線グラフを描いた際の凡例が表示されます。

```
ax1.legend(loc='best') #  降水量の凡例
ax2.legend(loc='best') #  気温の凡例
```

降水量の凡例と気温の凡例をまとめて付けることも可能です（図４－３－２）。amedas_prep_mean+sd_mon2.py はこのように凡例の付け方を変えたプログラムです。

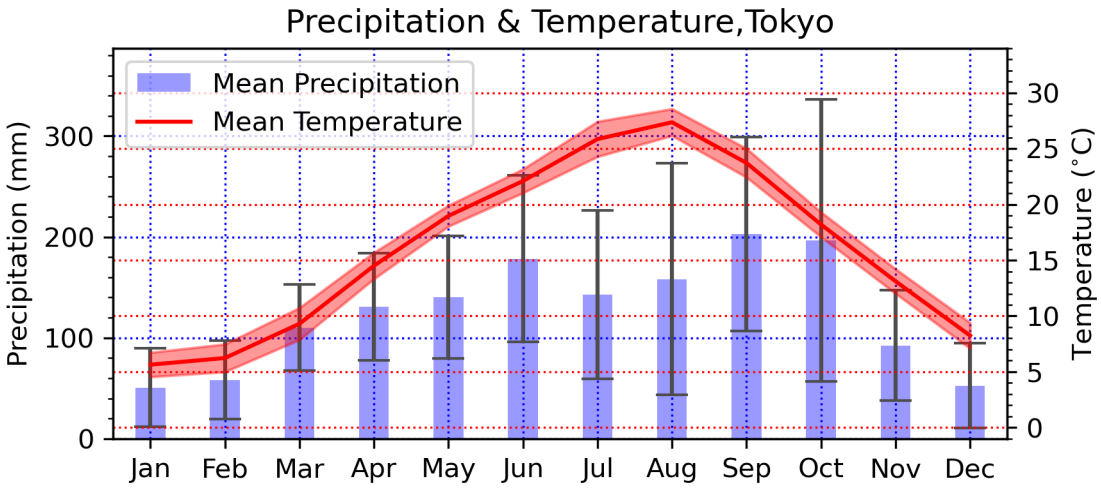


図４－３－２　図４－３－１の凡例の場所を１箇所にまとめた

まず大きな違いとしては、これまで使ったことがなかった描画関数の戻り値を利用するようにしたという部分です。棒グラフを作成する plt.bar の戻り値を p1 に、折れ線グラフを作成する plt.plot の戻り値を p2 に格納しています。この手法では関数の戻り値からラベルを設定することができないので、ラベルに表示したい文字列を l1、l2 に格納しました。

```
p1 = plt.bar(index, …) #  棒グラフ作成時の戻り値を p1 に格納
l1 = 'Mean Precipitation' #  棒グラフのラベル
p2 = plt.plot(index, …) #  折れ線グラフ作成時の戻り値を p2 に格納
l2 = 'Mean Temperature' #  折れ線グラフのラベル
```

plt.legend には、描画関数の戻り値の最初の要素（p1[0]、p2[0]）やラベルの文字列（l1、l2）を引数として渡すことが可能です。１番目の引数として(p1[0], p2[0])のタプル、２番目の引数として(l1, l2)のタプルを渡しています。p1[0]で棒グラフのマーク、l1 で棒グラフのラベルのように、１番目の引数、２番目の引数それぞれの最初の要素から順に凡例が付けられていきます。

```
plt.legend((p1[0], p2[0]), (l1, l2), loc='best') #  凡例を付ける
```

凡例を付ける際のオプションは他にもあるので表４－３－１に挙げておきます。詳細については、4.12.2 節で解説します。

表４－３－１　matplotlib の pyplot.legend の主要オプション一覧

オプション	説明
handles	作図の戻り値、例：(“戻り値１”, “戻り値２”)
labels	表示するラベル、例：(“ラベル１”, “ラベル２”)
loc	凡例の位置、{'best' | 'upper left' | 'upper center' | 'upper right' | 'lower left' | 'lower center' | 'lower right' | 'right' | 'center left' | 'center right' | 'center'}、デフォルト値：'best'
bbox_to_anchor	位置と幅、高さのタプル、(x, y, width, height)、例：loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5) 例：loc='upper right', bbox_to_anchor=(0.5, 0.5)
fontsize	文字のサイズ
fancybox	角を丸くする、{None | True | False}、デフォルト値：rcParams["legend.fancybox"]
shadow	影を付ける、{None | True | False}、デフォルト値：rcParams["legend.shadow"]
framealpha	枠の不透明度（０～１）、デフォルト値：rcParams["legend.framealpha"]
facecolor	凡例の色、デフォルト値：rcParams["legend.facecolor"]
edgecolor	凡例の枠の色、デフォルト値：rcParams["legend.edgecolor"]
borderaxespad	凡例の枠とアンカーの間の隙間、デフォルト値：rcParams["legend.borderaxespad"]
columnspacing	行間の隙間、デフォルト値：rcParams["legend.columnspacing"]

使用方法：plt.legend()、plt.legend(labels, オプション)、plt.legend(handles, labels, オプション)

4.3.2 グリッド線や目盛りなどの体裁を整える

x 軸の月のグリッドに関しては、後から描いた青色になってしまうのと、y 軸に 2 種類のグリッド線があり分かりにくいという問題が残っています。amedas_prep_mean+sd_mon3.py は、グリッド線の付け方を変えるプログラムです。y 軸の範囲を自動で決めてしまうと両側の軸の目盛りを合わせるのが難しいので、棒グラフの作図部分では、次のように 0〜390 mm の範囲を指定しました。y 軸の目盛りの設定は、手動で目盛り間隔の値を指定できる MultipleLocator を使い、大目盛りは 100 mm 毎、小目盛りは 20 mm 毎に付けています。

```
plt.ylim([0, 390]) # y 軸の範囲（左側）
ax1.yaxis.set_major_locator(ticker.MultipleLocator(100)) #  大目盛り
ax1.yaxis.set_minor_locator(ticker.MultipleLocator(20)) #  小目盛り
```

気温に関しては、0〜39 mm の範囲に設定し、大目盛りを 10 度、小目盛りを 2 度に設定することでグリッド線の場所を合わせます。

```
plt.ylim([0, 39]) # y 軸の範囲（左側）
ax2.yaxis.set_major_locator(ticker.MultipleLocator(10)) #  大目盛り
ax2.yaxis.set_minor_locator(ticker.MultipleLocator(2)) #  小目盛り
```

グリッド線は灰色で描きます。このようにして作図したものが図 4 – 3 – 3 です。グリッド線がスッキリしたと思います。

```
plt.grid(color='gray', ls=':') #  グリッド線を描く
```

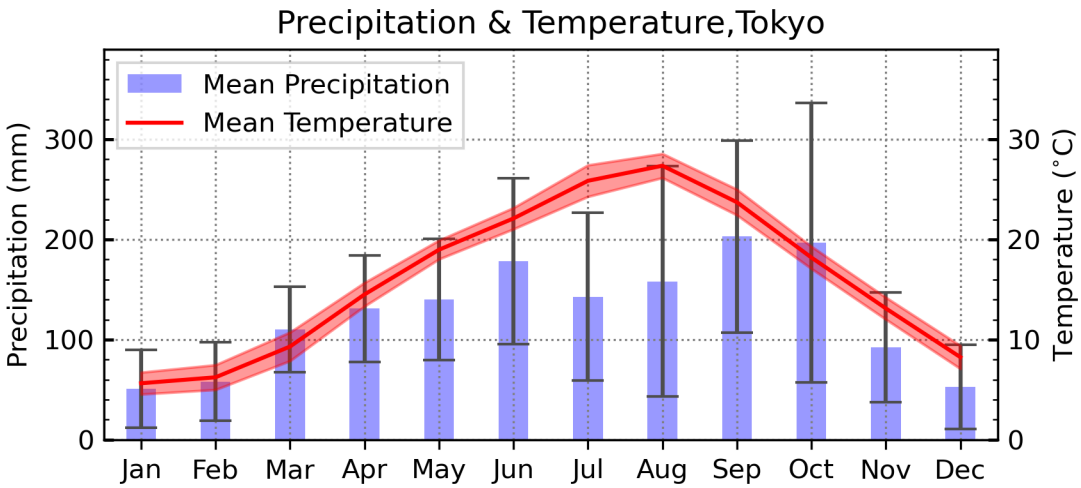


図４－３－３　グリッド線をシンプルに、目盛り線を内側にして大目盛り線を少し太く、凡例の枠を角張った四角形にした

これまでの図と違い、目盛り線の向きが外側から内側に変わっているのに気が付いたでしょうか。図を作る前に、デフォルトの描画パラメータを変更するおまじないを行っています。plt.rcParam がデフォルトパラメータの設定で、そのうちの ytick.direction が y 軸の目盛り線の向き、ytick.major.width が y 軸の大目盛り線の太さを意味しています。

```
plt.rcParams['ytick.direction'] = 'in' # y 軸目盛り線の向き
plt.rcParams['ytick.major.width'] = 1.2 # y 軸大目盛り線の太さ
```

もう 1 箇所変わった部分があります。凡例の枠が丸みを帯びた形から角張った四角形に変わっています。legend.fancybox を False に変えたためです。

```
plt.rcParams["legend.fancybox"] = False #  凡例の枠を角張った四角形
```

他にも plt. rcParam で変更可能な目盛り線や凡例の書式があるので、表４－３－２と表４－３－３にまとめておきます。また plt.rcParam で図のフォントを変更することも可能で、それらを表４－３－４にまとめました。

表４－３－２　目盛り線の書式

plt.rcParams['項目']	説明
xtick.direction	x軸の目盛線の向きをinかoutで指定、デフォルト値：out
ytick.direction	y軸の目盛線の向きをinかoutで指定、デフォルト値：out
xtick.major.width	x軸の主目盛線の太さを指定、デフォルト値：0.75、推奨値：1.2
ytick.major.width	y軸の主目盛線の太さを指定、デフォルト値：0.75、推奨値：1.2
xtick.minor.width	x軸の副目盛線の太さを指定、デフォルト値：0.75、推奨値：0.75
ytick.minor.width	y軸の副目盛線の太さを指定、デフォルト値：0.75、推奨値：0.75

表４－３－３　凡例の書式

plt.rcParams['項目']	説明
legend.fancybox	枠線の角を丸くするかどうか、デフォルト値：True（丸）、角：False
legend.framealpha	凡例の不透明度、0～1の範囲（1で不透明）
legend.edgecolor	枠線の色、デフォルト値：gray
legend.markerscale	マーカーの大きさ、デフォルト値：2、推奨値：2

表４－３－４　フォントの書式

plt.rcParams['項目']	説明
font.size	フォントサイズ、デフォルト値：10
font.family	飾り付き（serif）か飾り無し（sans-serif）、デフォルト値：'sans-serif'
font.sans-serif	sans-serifのフォント名、例：['Arial']

4.4 ヒストグラムの作成

ここからはヒストグラムの作図に移ります。最初はランダムなデータを使って作図方法を学び、その後で実際の気象データを使った作図を行います。

4.4.1 基本的なヒストグラム

まずはランダムなデータを使って作図します（hist_rand.py）。平均 0、標準偏差 1 の正規分布に従うランダムデータを生成するツールが Numpy に入っており、np.random.randn(**データの個数**)のように呼び出します。このツールで生成されるランダムデータは毎回変わりますが、解説の際にはデータに再現性を持たせたいので、np.random.seed(**整数**)というツールを使い、生成されるデータを固定するテクニックを用いました。作成されたものが図４－４－１です。

```
# 再現性を保つため、random state を固定
np.random.seed(1900)
hist_y = np.random.randn(1000) # ランダムデータの準備
```

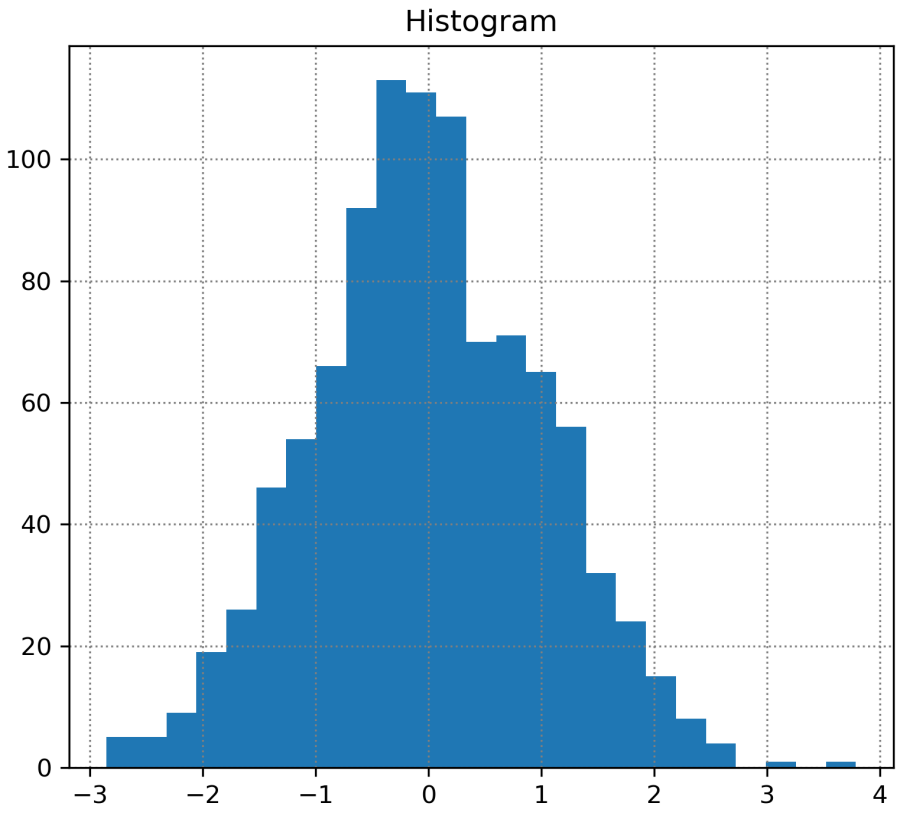


図４－４－１　1000 個のランダムデータから作成したヒストグラム

棒グラフや折れ線グラフの時のように横長の図にはしたくなかったので、プロット範囲を(6, 6)にしました。ヒストグラムの作成は plt.hist で行います。サブプロット作成時に作成した ax を使い、ax.hist としても同じです。1 番目の引数がヒストグラムを作成するための 1 次元データです。まずは、ビン数を自動設定する bins='auto'で作図しました。

```
fig = plt.figure(figsize=(6,6)) #  プロット範囲（6x6）
ax = fig.add_subplot(1,1,1) #  サブプロット作成
plt.hist(hist_y, bins='auto') #  ヒストグラムを描く
```

ちなみに ax の属性は、<class 'matplotlib.axes._subplots.AxesSubplot'>となっていて、python のクラスを参照するようなオブジェクトです（インスタンス）。そのため、ax.hist でヒストグラムを描くといった、インスタンスメソッドを使うことが可能です。

4.4.2 ヒストグラムのスタイルを変える

まだデフォルトの体裁のままで見栄えが良くないので、グラフの体裁を変えていきましょう。まずは、グラフの色や不透明度、枠線の色を変えてみます（図４－４－２）。作図に用いたのが hist_rand2.py です。プログラムの最初で plt.hist に渡す STYLES を定義しておきます。左上が灰色で半透明、右上が青色で半透明になりました。ここでも棒グラフ同様に edgecolor が使えて、左下のように黒の枠線が付きます。不透明にしたものが右下です。

```
STYLES = [
    dict(color='gray', alpha=0.4), #  灰色、半透明
    dict(color='b', alpha=0.4), #  青、半透明
    dict(color='b', alpha=0.4, edgecolor='k'), #  青、半透明、枠線を黒
    dict(color='r', alpha=1.0, edgecolor='k') #  赤、不透明、枠線を黒
]
```

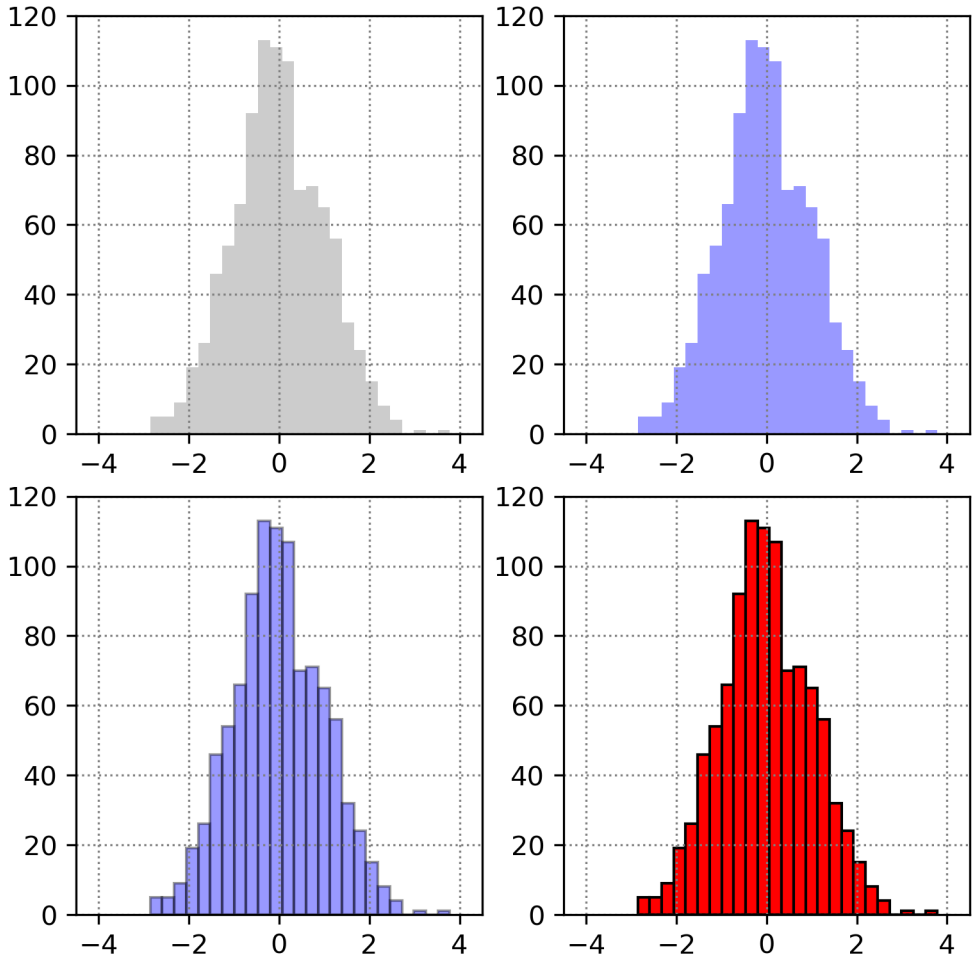


図４－４－２　４つのサブプロットに作図オプションを変えてヒストグラムをプロット

縦と横に２つ並べたサブプロットを計４つ生成し、STYLES の要素番号を変えて、それぞれのサブプロットで別々に作図していくことを考えます。まず np.arange(4)で 0～4 までの整数を順に生成してループ中で n の値を更新していきます。その値を使い、サブプロットの生成とヒストグラムの作図、グリッド線の描画を行います。縦に 2、横に 2 の計４つのサブプロットを生成するため、fig.add_subplot(2, 2, n+1)を行っています。番号を n+1 としたのは、サブプロットの番号を 1～4 にする必要があるためです。生成したサブプロットを ax.append で ax に追加していきます。ヒストグラムの作図を行う ax[n].hist、グリッド線の描画を行う ax[n].grid は、ax というリストの要素である ax[0]、ax[1]、ax[2]、ax[3]にヒストグラムやグリッド線を描くことを意味しています。

少し複雑ですが、ax はリストとして定義して、リストの要素としてサブプ

ロットを追加する操作を行いました。このように、リストには文字列や数字だけではなく様々な属性のオブジェクトを追加できるようになっています。

今のプログラムでは ax をループの外で参照していないので、実は毎回 ax=fig.add_subplot(…)でも作図できます。サブプロットをリストにする方法は、今後複雑な作図をする場合には必要なテクニックなので、こういう方法もあるという程度に思っておいて下さい。

```
ax = list()
for n in np.arange(4):
    # サブプロット作成
    ax.append(fig.add_subplot(2,2,n+1))
    # ヒストグラム
    ax[n].hist(hist_y, bins='auto', **STYLES[n])
    # グリッド線を描く
    ax[n].grid(color='gray', ls=':')
    # x 軸、y 軸の範囲を固定
    plt.xlim([-4.5, 4.5])
    plt.ylim([0, 120])
```

棒グラフの時のようにハッチを付けることも可能です（図４－４－３）。ハッチが付いたのに加え、先ほどと違い、４つのグラフで分布が異なるのに気が付いたでしょうか。乱数の発生の際に random state を固定していないので、作図される毎に結果が異なります。作図に用いたプログラムは hist_rand3.py です。右上の図だけ fill=True、color='b'で青塗りした上にハッチをかけ、他の図はハッチのみで作図しました。

```
STYLES = [
    dict(alpha=0.4, fill=False, hatch='//'), #  半透明
    dict(color='b', alpha=0.4, fill=True, hatch='++'), #  半透明、青塗り
    dict(alpha=0.4, edgecolor='k', fill=False, hatch='xx'), #  半透明
    dict(alpha=1.0, edgecolor='k', fill=False, hatch='oo') #  不透明
]
```

作図に必要な設定をプログラムの最初に持ってきています。

```
size_of_sample = 1000 #  サンプルサイズを設定
num_bins = 20 #  ビン数の設定
#  作図範囲の設定
xmin = -4.5 # x 軸下限
xmax = 4.5 # x 軸上限
ymin = 0 # y 軸下限
ymax = 200 # y 軸上限
```

サンプルサイズの設定は、np.random.randn(size_of_sample)で生成させるサンプル数を、xmin と xmax は plt.xlim で設定される x 軸の範囲を、ymin、ymax は plt.ylim で設定される y 軸の範囲をそれぞれ変更します。ちなみに軸の範囲を自動設定したい時には、xmax=None のようにするだけです。ヒストグラムの作成では、これまでの bins='auto'の代わりに bins=num_bins で手動設定しています。

```
hist_y.append(np.random.randn(size_of_sample)) #  ランダムデータ準備
ax[n].hist(hist_y[n], bins=num_bins, **STYLES[n]) #  ヒストグラム作成
plt.xlim([xmin, xmax]) # x 軸の範囲
plt.ylim([ymin, ymax]) # y 軸の範囲
```

x 軸の大目盛と小目盛を自動設定する文も加えています。ax[n]を使っている部分を除けば、棒グラフの所までに行ってきた方法と同じです。

```
ax[n].xaxis.set_major_locator(ticker.AutoLocator()) #  大目盛
ax[n].xaxis.set_minor_locator(ticker.AutoMinorLocator()) #  小目盛
```

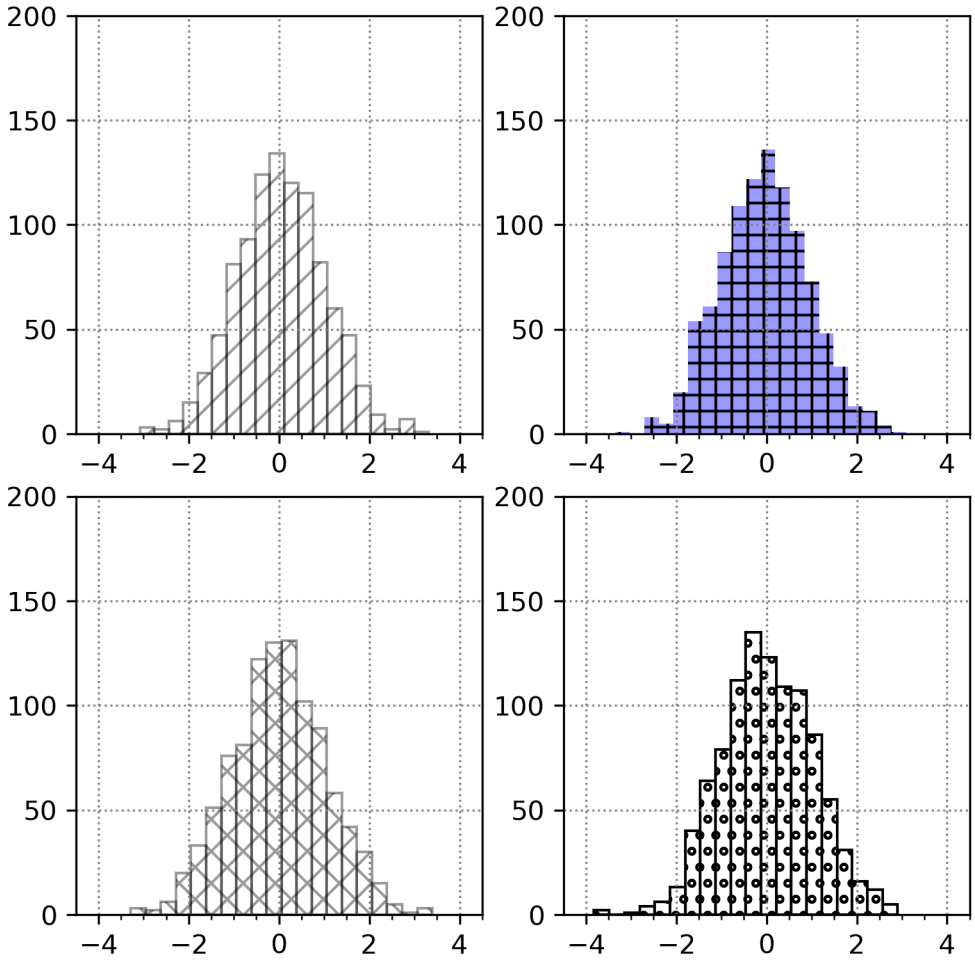


図４－４－３　ヒストグラムにハッチを付けた

4.4.3 ビンの間隔を手動で設定する

これまでの作図では、ビンの間隔を設定する方法は自動（bins='auto'）かビン数の指定（bins=ビン数）だけでした。これらの方法では、データの最小値と最大値から勝手にビンの幅が決まってしまいます。実際にデータを扱う場合には、図４－４－３のようなｘ軸のσの値とビンの端がずれている図では不便なこともあります。そのような場合には、ビンの端点の値をリスト（edges=[端点 1，端点 2，…，端点 n]）にして、bins のパラメターで bins=edges のように指定する方法が便利です。図４－４－４は、そのようにビンの端点を固定したものです。作図には hist_rand4.py を使いました。ビンの端を見やすくするため、ハッチを塗り潰さないようにしています。

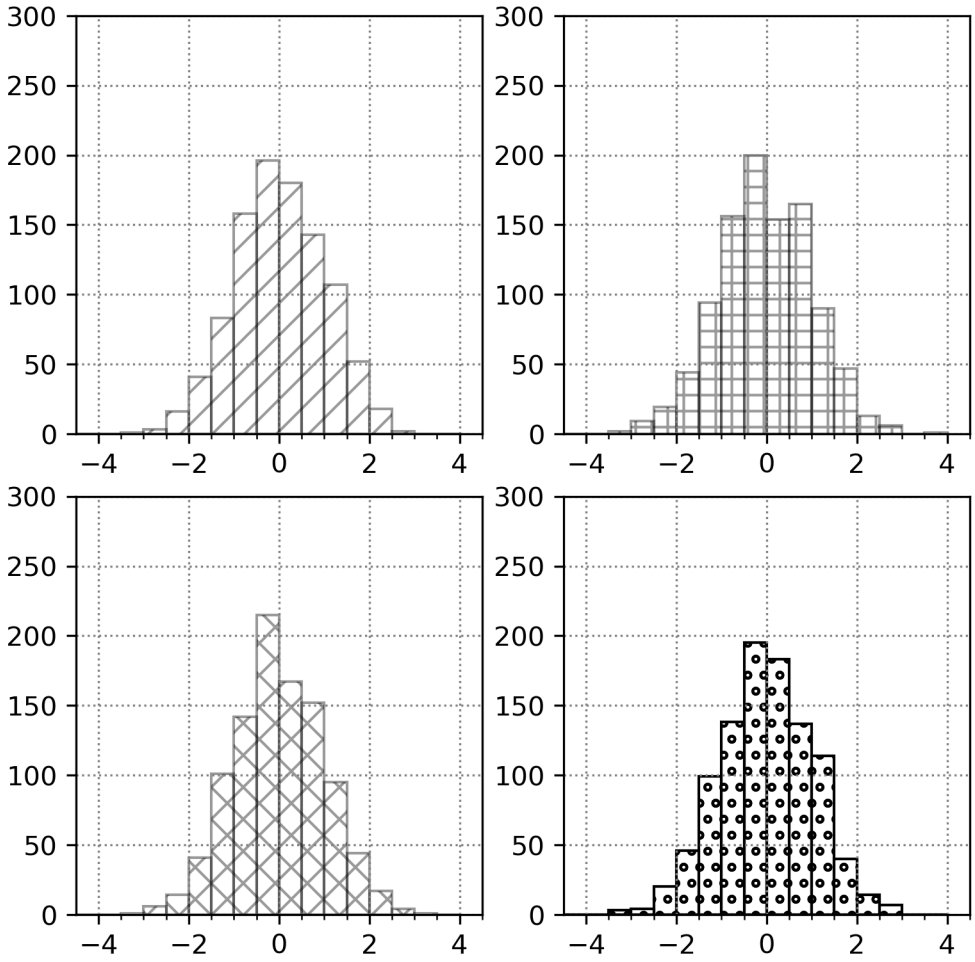


図４－４－４　ビンの端点を指定した

ヒストグラムを作成する際の端点を決めるため、edges を定義しています。np.arange(start, end, step)のように３つの引数を与えると、等間隔に値が入った配列（ndarray）を生成します。この場合には、-4.5、-3.5、…、3.5、4.5になります。ax[n].hist で bins=edges を渡しています。

```
edges = np.arange(-4.5, 4.6, 0.5) #  ビンの端点の設定
…
ax[n].hist(hist_y[n], bins=edges, **STYLES[n]) #  ヒストグラム作成
```

プログラムを実行した際に、算術平均（mean)、標準偏差（standard deviation)、中央値（median)、歪度（skewness)、尖度（kurtosis）といった

統計量の値が表示されるようになったのに気が付いたでしょうか。作図と一緒に次のような計算を追加しています。mean()と std()は Numpy の ndarray のメソッドを使いました。Numpy には他の 3 種類の統計量を計算するメソッドは含まれていないので、Pandas の Series に直し、Series に含まれている median()、skew()、kurtosis()メソッドを使いました。このようなテクニックを使うため、プログラムの最初で Series を import しています。

```
from pandas import Series
…
print("mean = ", Series(hist_y[n]).mean()) #  算術平均
print("SD = ", Series(hist_y[n]).std()) #  標準偏差
print("median = ", Series(hist_y[n]).median()) #  中央値
print("skewness = ", Series(hist_y[n]).skew()) #  歪度
print("kurtosis = ", Series(hist_y[n]).kurtosis()) #  尖度
```

4.4.4 ヒストグラムと同時に統計量を表示

これらの統計量をヒストグラムに表示したいこともあると思います。そのようなプログラムが hist_rand_size.py です。mean = "mean = %3.2f" % 数値、のような表記で、小数点以下 2 桁まで有効にした浮動小数点数を%3.2f の部分に代入した表記の文字列を mean に入力しています。入力された文字列を plt.text を使って指定した場所（xmin+0.5、ymax-0.05）にプロットします。

```
mean = ("mean = %3.2f" % Series(hist_y[n]).mean())
std = ("SD = %3.2f" % Series(hist_y[n]).std())
med = ("median = %3.2f" % Series(hist_y[n]).median())
skew = ("skewness = %3.2f" % Series(hist_y[n]).skew())
kurt = ("kurtosis = %3.2f" % Series(hist_y[n]).kurtosis())
plt.text(xmin + 0.5, ymax - 0.05, mean, fontsize=8)
plt.text(xmin + 0.5, ymax - 0.1, std, fontsize=8)
plt.text(xmin + 0.5, ymax - 0.15, med, fontsize=8)
plt.text(xmin + 0.5, ymax - 0.2, skew, fontsize=8)
plt.text(xmin + 0.5, ymax - 0.25, kurt, fontsize=8)
```

このプログラムでは、ヒストグラムを描く際に合計が 1 となるように、density=True を行っています。古いリファレンスでは normed=True のオプションになっていますが、今は density オプションに置き換えられましたので、こちらを使った方が良いです。

```
ax[n].hist(hist_y[n], density=True, bins=edges, **STYLES[n])
```

このプログラムでは全体のタイトルとして、サンプルサイズの表示を付けています。サブプロットを定義する前にタイトルを付けると、不要な枠線が出てしまうので、plt.axis('off')を行って枠線が表示されないように工夫しています。この設定はその後には反映されないので、サブプロットの定義の前に変更する必要はありません。

```
plt.axis('off')
plt.title("n = " + str(size_of_sample))
```

ランダムなサンプル数を 10、100、1000、10000、100000 と変えて、それぞれ 4 つのプロットを行ったものが図 4－4－5 です。np.random.randn では N(0.0, 1.0)に従う正規分布を生成しますが、サンプル数が 10 ではバラバラな分布です。平均、標準偏差もバラバラです。ランダムというのはこういうものかもしれません。サンプル数が 100 となると、平均は 0.0、標準偏差は 1.0 に近付きますが、歪度や尖度はバラバラで、とても正規分布とは言えません。正規分布に従うランダムなサンプルですらこのようになるので、サンプル数 100 程度では分布関数の形を正確に決めることはできないことが分かります。サンプル数が 1000 になると、見た目はほぼ正規分布になり平均、標準偏差に加えて歪度が 0 に近づいてきますが、まだ尖度はバラバラです。サンプル数が 10000 になると、歪度、尖度共に 0 に近づいてきます。横の 4 つの分布関数の形が互いに似ていますが、まだ微妙な違いが見られます。サンプル数が 100000 になれば、分布関数の形は横の 4 つの並びでほぼ同じです。これらを見ていくと、分布関数の形を決めるには、平均、標準偏差、歪度、尖度が正規分布のものになる 1000～10000 程度のサンプル数が必要と考えられます。

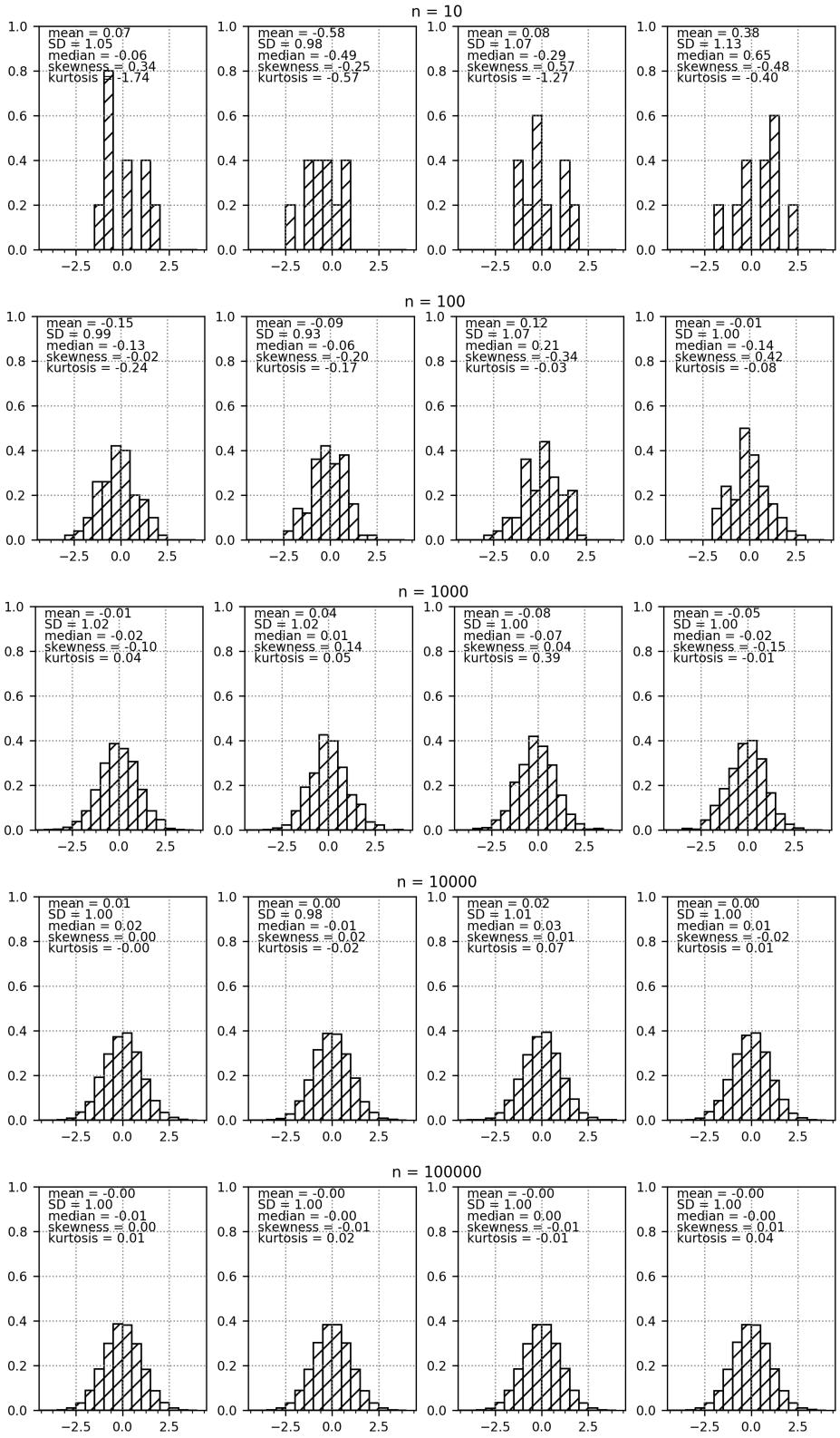


図４－４－５　サンプル数を変えたヒストグラム

4.4.5 2 つのヒストグラムを同時に描く

2 つのヒストグラムを 1 枚の図に描きたいこともあると思います。そのようなプログラムが hist_rand_double.py です。最初にサンプルサイズやビンの端点、作図範囲の設定を行っています。今回は平均 50、標準偏差 10 と平均 30、標準偏差 4 のヒストグラムを描くので、x 軸の範囲は 0〜100 にしました。
mu1=50、sigma1=10 として、
hist_y1 = mu1 + sigma1 * np.random.randn(size_of_sample)
のように、平均 50、標準偏差 10 の正規分布に従うようなランダムデータを生成します。平均 30、標準偏差 4 のサンプルについても同様です。

```
size_of_sample = 1000 #  サンプルサイズを設定
edges = np.arange(0, 100, 5) #  ビンの端点の設定

#  作図範囲の設定
xmin = 0
xmax = 100
ymin = 0
ymax = 0.1

#  平均、標準偏差の指定
mu1, sigma1 = 50, 10
mu2, sigma2 = 30, 4
#  ランダムデータの準備
hist_y1 = mu1 + sigma1 * np.random.randn(size_of_sample)
hist_y2 = mu2 + sigma2 * np.random.randn(size_of_sample)
```

2 つのサンプルのヒストグラムを描く際のオプションを STYLES で準備しておきます。1 つ目のサンプルを青、2 つ目のサンプルを赤で表します。ヒストグラムでもラベルオプション（label）が使えるので、サンプルの中身をラベルにしています。

```
STYLES = [
    dict(color='b', alpha=1.0, edgecolor='k', label='$¥mu=50,¥ ¥sigma=10$'),
    dict(color='r', alpha=1.0, edgecolor='k', label='$¥mu=30,¥ ¥sigma=4$')
]
```

ヒストグラムでも軸のラベルを付けることが可能です。

```
plt.xlabel('x') # x 軸のラベル
plt.ylabel('Frequency') # y 軸のラベル
```

作図部分です。単に plt.hist を 2 回呼び出しているだけです。最初に描いたグラフが次に描いたグラフに隠されています（図４－４－６）。

```
plt.hist(hist_y1, density=True, bins=edges, **STYLES[0])
plt.hist(hist_y2, density=True, bins=edges, **STYLES[1])
```

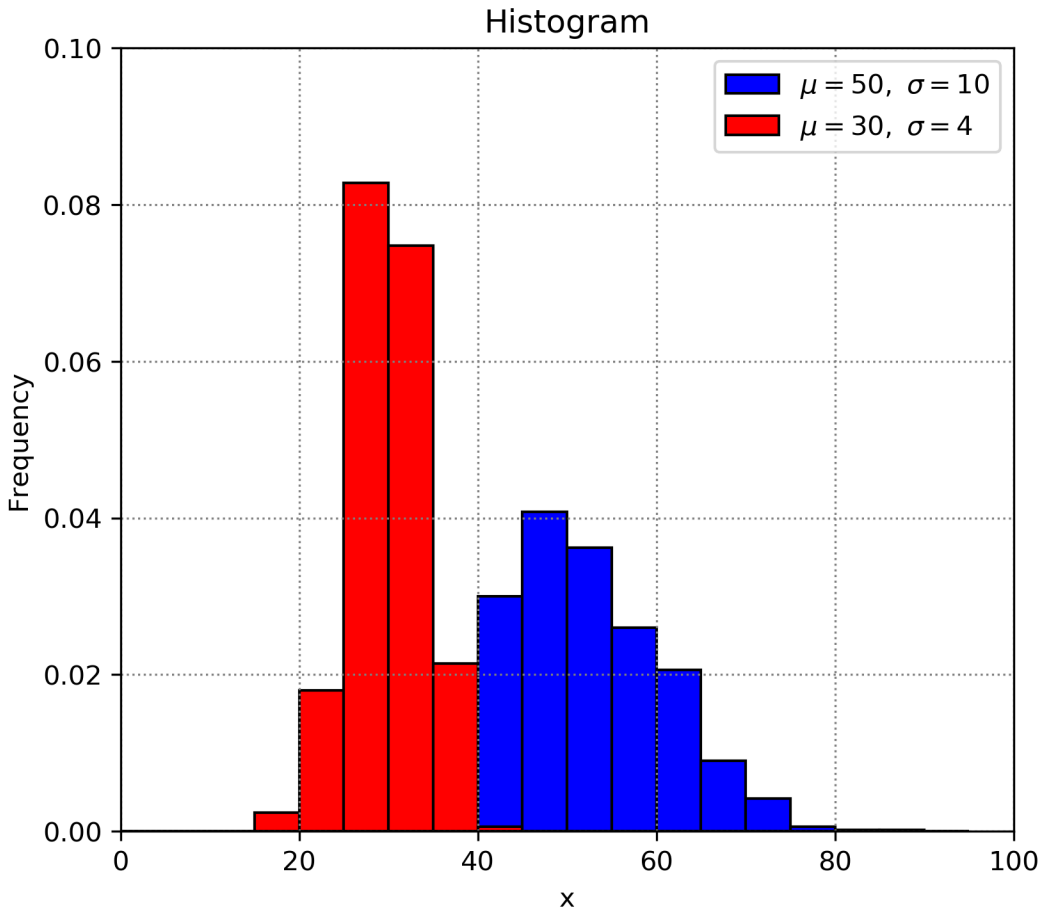


図４－４－６　２つのヒストグラムを１枚のグラフに表示

グラフを半透明にしておくと、２つのグラフが見えるようになります（図４－４－７）。作図には hist_rand_double2.py を用いました。プログラム中では、次のように STYLES で alpha=0.4 と不透明度を変えています。

```
STYLES = [
    dict(color='b', alpha=0.4, edgecolor='k', label='$¥mu=50,¥ ¥sigma=10$'),
    dict(color='r', alpha=0.4, edgecolor='k', label='$¥mu=30,¥ ¥sigma=4$')
]
```

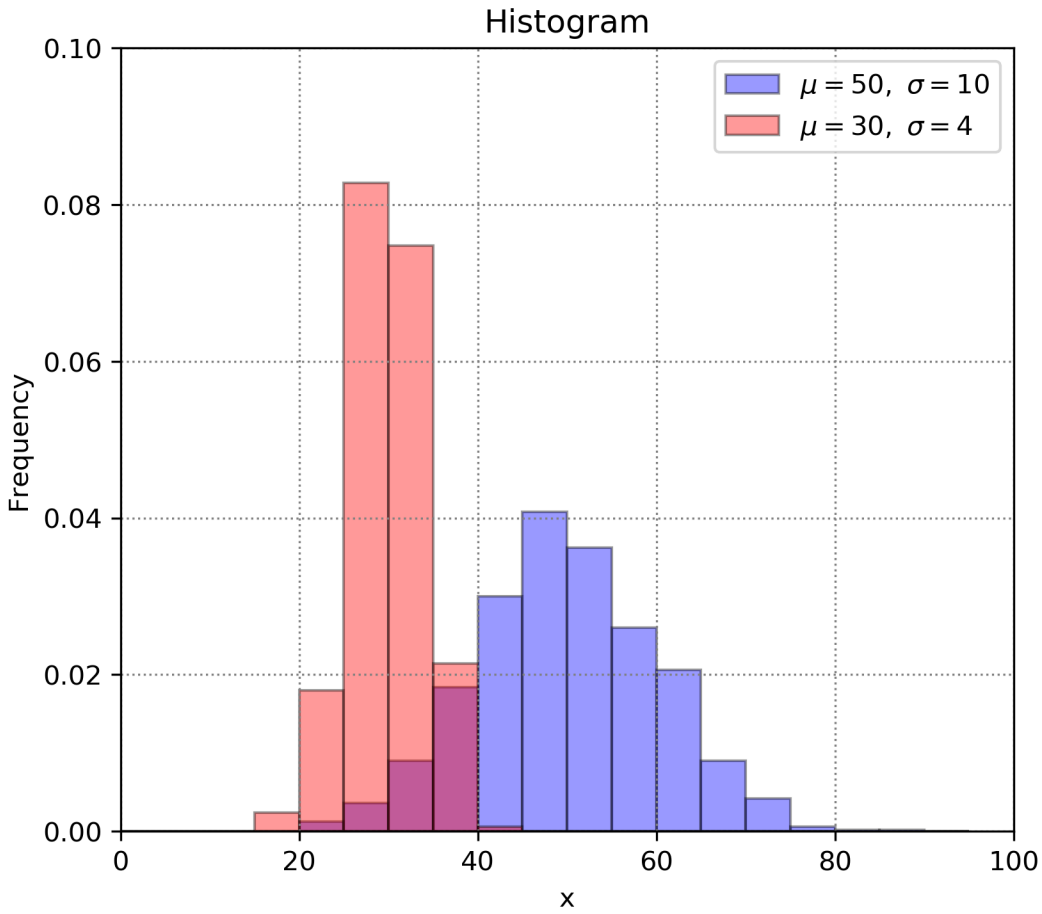


図４－４－７　ヒストグラムを半透明にした

4.4.6 降水量データのヒストグラム

それでは、実際の降水量データを使って作図してみます。図４－４－８のように東京のアメダス地点の 7 月積算降水量の上限は 400 mm に達しないので、0～400 mm の範囲でヒストグラムを作図します（amedas_prep_hist_jul.py)。20 mm 毎にビンを区切るため、np.arange(0, 401, 20)の結果を edges に入力しておきます。ymax は自動設定にするため None としました。

```
edges = np.arange(0, 401, 20) #  ビンの端点の設定
#  作図範囲の設定
xmin = 0
xmax = 400
ymin = 0
ymax = None
```

作図のスタイルも設定しておきます。ラベルには Jul など 1 文字目が大文字となった文字列が入るようにします。

```
STYLES = [
    dict(color='b', alpha=0.4, edgecolor='k', label=str(month).capitalize())
]
```

データの取り出し方は棒グラフの時と同じです。

```
amedas = AmedasStation(sta) # AmedasStation Class の初期化
prep_i = amedas.retrieve_mon("prep") #  降水量データ取得
prep = prep_i.loc[syear:eyear,month] #  降水量データの取り出し
```

取り出した東京のアメダス地点 7 月の積算降水量データからヒストグラムを作成します。density=True としたので、ヒストグラムの総和が 1 になります。

```
plt.hist(prep, density=True, bins=edges, **STYLES[0]) #  ヒストグラム作成
```

大目盛線、小目盛線の間隔も設定しておきます。

```
ax.xaxis.set_major_locator(ticker.AutoLocator()) # x 軸大目盛
ax.xaxis.set_minor_locator(ticker.AutoMinorLocator()) # x 軸小目盛
ax.yaxis.set_major_locator(ticker.AutoLocator()) # y 軸大目盛
ax.yaxis.set_minor_locator(ticker.AutoMinorLocator()) # y 軸小目盛
```

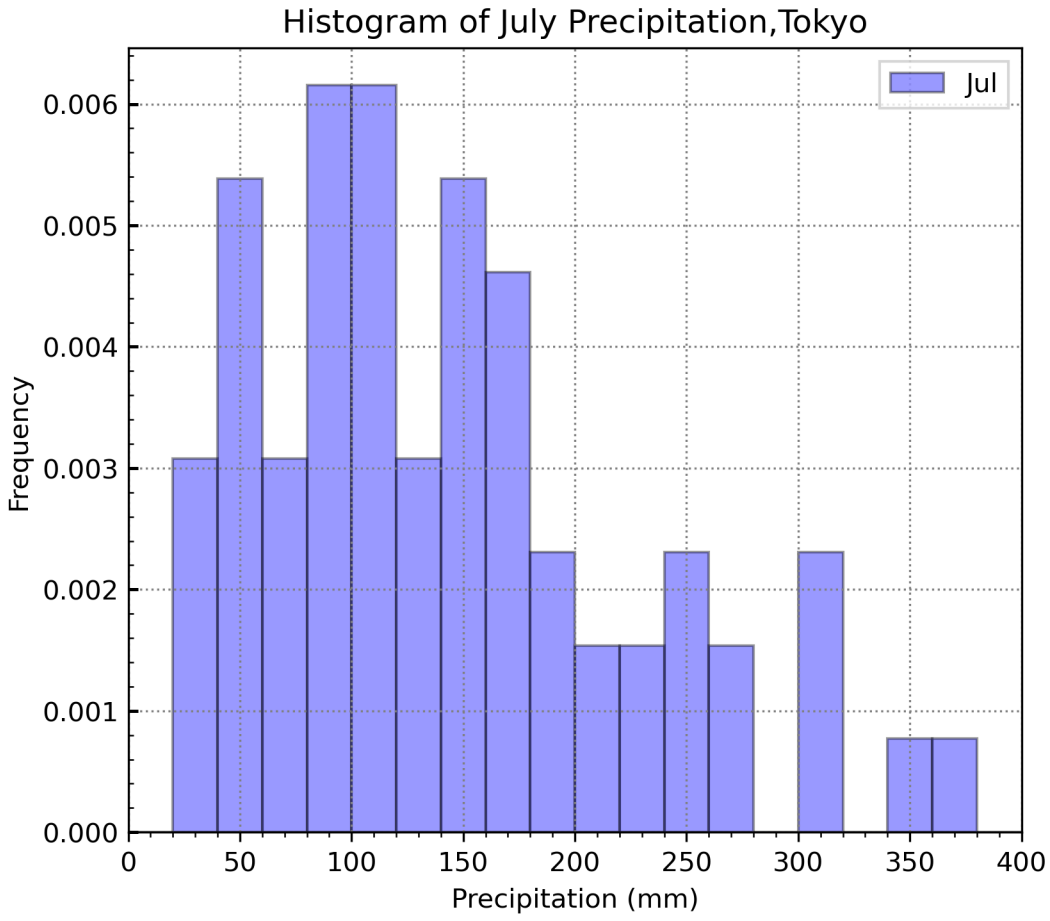


図４－４－８　東京のアメダス地点における 7 月積算降水量から作成したヒストグラム（1960～2024 年）

4.4.7　3つのヒストグラムを横に並べる

夏季（6～8月）3ヶ月分の降水量データのヒストグラムを横に並べて比較してみます（amedas_prep_hist_3mon.py）。図4－2－2を見ると上限が 500 mm を少し超えているので、0～540 mm の範囲で作図しています。

```
edges = np.arange(0, 541, 20) #  ビンの端点の設定
#  作図範囲の設定
xmin = 0
xmax = 540
ymin = 0
ymax = None

months = ["jun", "jul", "aug"]
```

作図の際のスタイルは次のように設定します。

```
STYLES = [
    dict(color=['g', 'b', 'aqua'], alpha=0.4, edgecolor='k', ¥
        label=[str(months[0]).capitalize(),¥
        str(months[1]).capitalize(), str(months[2]).capitalize()])
]
```

6～8月のデータ作成は、降水量の棒グラフを作成した時と同じで、prep に3ヶ月分のデータを追加しています。ヒストグラムを作成する部分では、その3ヶ月分のデータを渡します。plt.hist では、最初の引数として配列が要素になっているリストを渡すと、x 軸上で横に並べたグラフを描こうとします。ここでは[prep[0], prep[1], prep[2]]のリストを渡しているので、20 mm 間隔で区切ったビン毎に prep[0]から順にヒストグラムの棒が並ぶような図になります（図4－4－9）。

```
plt.hist([prep[0], prep[1], prep[2]], density=True, bins=edges, **STYLES[0])
```

ここで先に定義しておいた STYLES を使っており、color=['g', 'b', 'aqua']のように要素に色の名前を取るリストを与えたので、6 月から順に緑、青、水色に塗られます（alpha=0.4 なので半透明です）。凡例のラベルについても同様のリストを渡すことができ、label=["jun", "jul", "aug"]のようなリストを渡しています。リストの要素は先ほどと同様に、str(months[0]).capitalize()を使い自動設定しました。

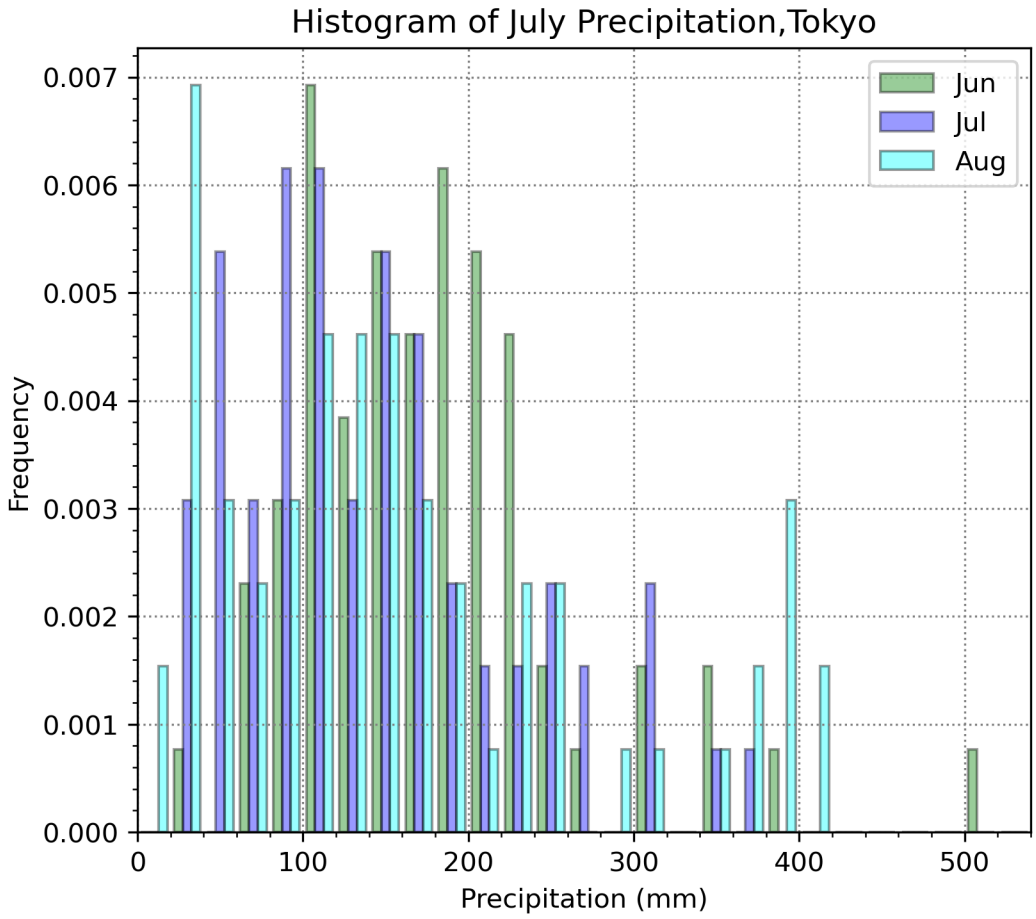


図４－４－９　東京のアメダス地点における 6〜8 月の積算降水量のヒストグラムを横に並べた（1960〜2024 年）。ヒストグラムの総和が、いずれも 1

4.4.8 積み上げヒストグラム

３ヶ月分の降水量データを縦に積み上げることも可能です（図４－４－10）。作図に用いた amedas_prep_hist_cum3mon.py は、前節のプログラムの plt.hist の引数に stacked=True のオプションを追加しただけです。このオプションを density=True と共に使うと、積み上げたものに対する総和が１となります。

```
plt.hist([prep[0], prep[1], prep[2]], density=True, stacked=True, ¥
        bins=edges, **STYLES[0])
```

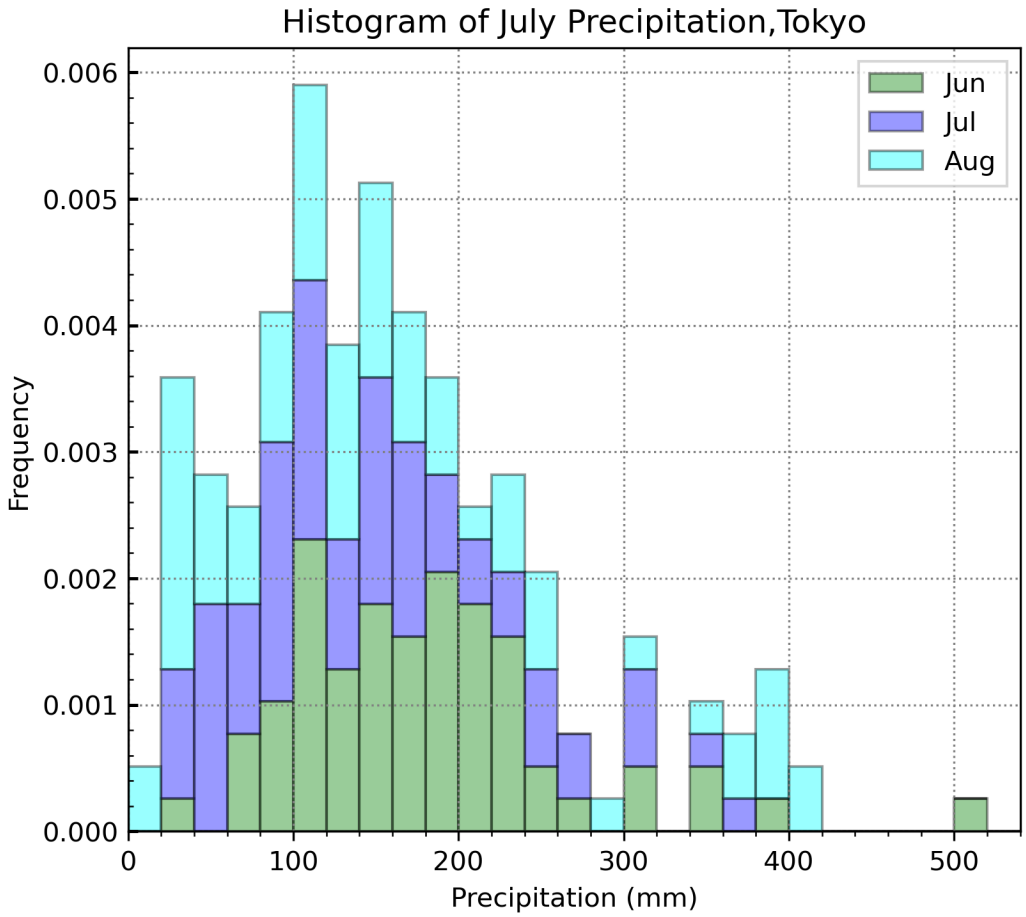


図４－４－10　積み上げヒストグラムにした場合。積み上げたものの総和が１

4.4.9 ヒストグラムとカーネル密度推定

図４－４－９にカーネル密度推定を重ねてみます（図４－４－11）。作図に用いたプログラムは、amedas_prep_hist+kde_3mon.py です。６月や８月が２山の分布になっているように見えます。６月に関しては、１つのデータで１山ができているので、２山の分布とは言い切れないでしょう。

次のように、ヒストグラムを描いた後、prep[n].plot(kind='kde')でカーネル密度推定を行なった結果が描かれます。なお prep[n]は Pandas の Series か DataFrame にしておく必要があり、Pandas の機能を利用してカーネル密度推定を行い作図する。

```
plt.hist([prep[0], prep[1], prep[2]], density=True, bins=edges, **STYLES[0])
for n in np.arange(3):
    prep[n].plot(kind='kde', **STYLESK[n]) #  カーネル密度推定
```

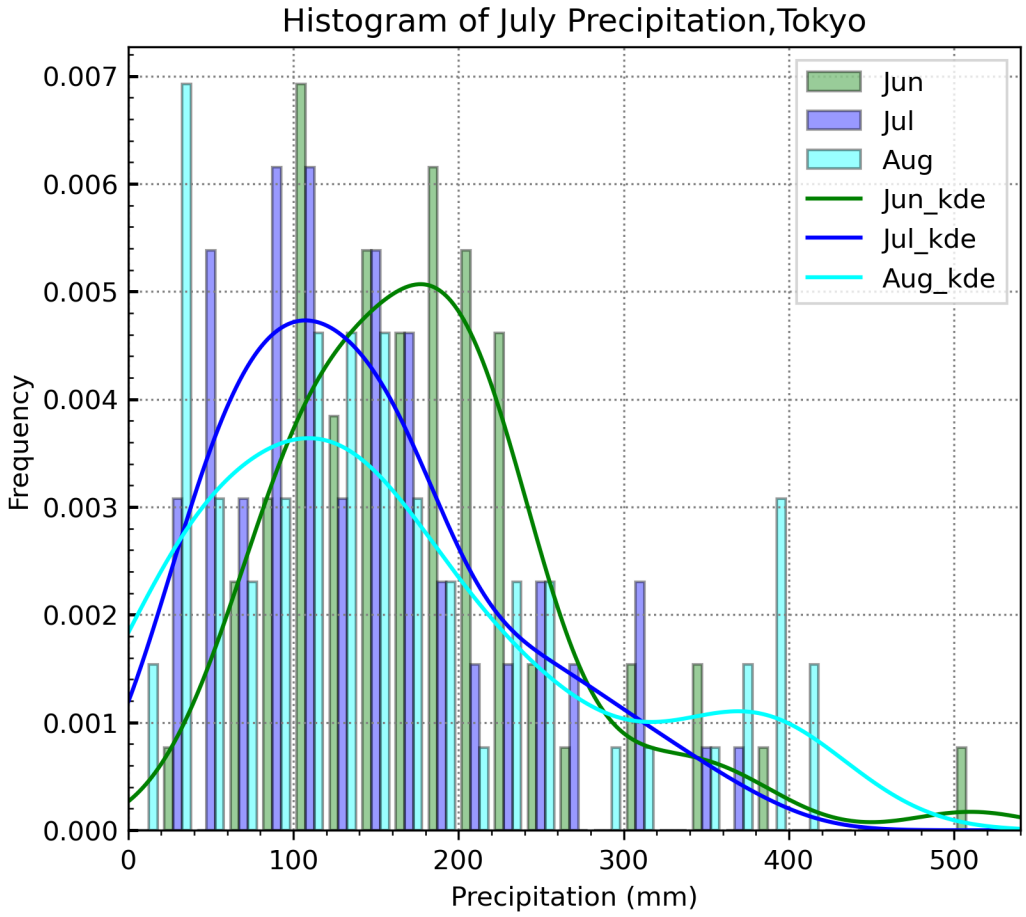


図４－４－11　図４－４－９にカーネル密度推定を重ねた

4.4.10 ヒストグラムのオプション

最後にヒストグラムで使用できるオプション一覧を表４－４－１にまとめておきます。オプションに histtype があり、'bar'（棒状)、'barstacked'（棒状、積み上げ)、'step'（階段状)、 'stepfilled'（階段状、塗り潰し）が可能です。そのままでは分かりにくいので、これまでの作図に用いてきた 7 月の積算降水量データを使い、histtype オプションを変えた４つの図にしました（図４－４－12)。作図に用いたプログラムは、amedas_prep_histtype_jul.py です。histtype オプションはデフォルトで histtype='bar'ですので、図４－４－８で作成した時と同様、棒状のヒストグラムが左上に作成されました。右上の図は histtype='barstacked'で作成しました。このオプションは積み上げを意味しますが、積み上げるデータが１つなので左上の図と同じです。左下は histtype='step'としたもので、枠線で階段状のヒストグラムが作成されます。塗り潰しの指定は無視されています。右下は histtype='stepfilled'とした場合で、階段状ヒストグラムに色が付きます。

表４－４－１　matplotlib の pyplot.hist で使用可能なオプション一覧

オプション	説明
x（必須）	配列、または複数配列のリスト
bins	文字列'auto'、ビンの個数（整数）、ビンの端点の値（配列）、デフォルト値：'auto'
range	下限と上限（タプル、端点指定で無効）、デフォルト値：None（(x.min(), x.max())を使用）
color	ヒストグラムの色、デフォルト値：None
density	Trueでヒストグラムの総和を1にする、デフォルト値：False densityと共に使うと積み上げの総和を1
stacked	Trueで複数データを積み上げ、デフォルト値：False
weights	データの重み（配列）、デフォルト値：None
cumulative	Trueでビンの累積和を用いる、デフォルト値：False
bottom	下側の余白（y軸上の開始位置）、デフォルト値：None（0を使用）
histtype	ヒストグラムの形式、{'bar', 'barstacked', 'step', 'stepfilled'}、デフォルト値：'bar'
align	ヒストグラムを描く位置、{'left', 'mid', 'right'}、デフォルト値：'mid' 'left'：ビンの左端、 'mid'：ビンの中央、'right'：ビンの右端
orientation	ヒストグラムの向き、{'horizontal', 'vertical'}、デフォルト値：'horizontal'
rwidth	ビンの幅、デフォルト値：None（自動設定）
log	Trueでログスケール、デフォルト値：False
color	ヒストグラムの色、デフォルト値：None
edgecolor	枠線の色、デフォルト値：None
fill	塗り潰す色、デフォルト値：False
hatch	ハッチのパターン、デフォルト値：None
alpha	不透明度、デフォルト値：1.0
label	凡例を付ける場合、デフォルト値：None

使用方法：plt.hist(x)、plt.hist(x, オプション)

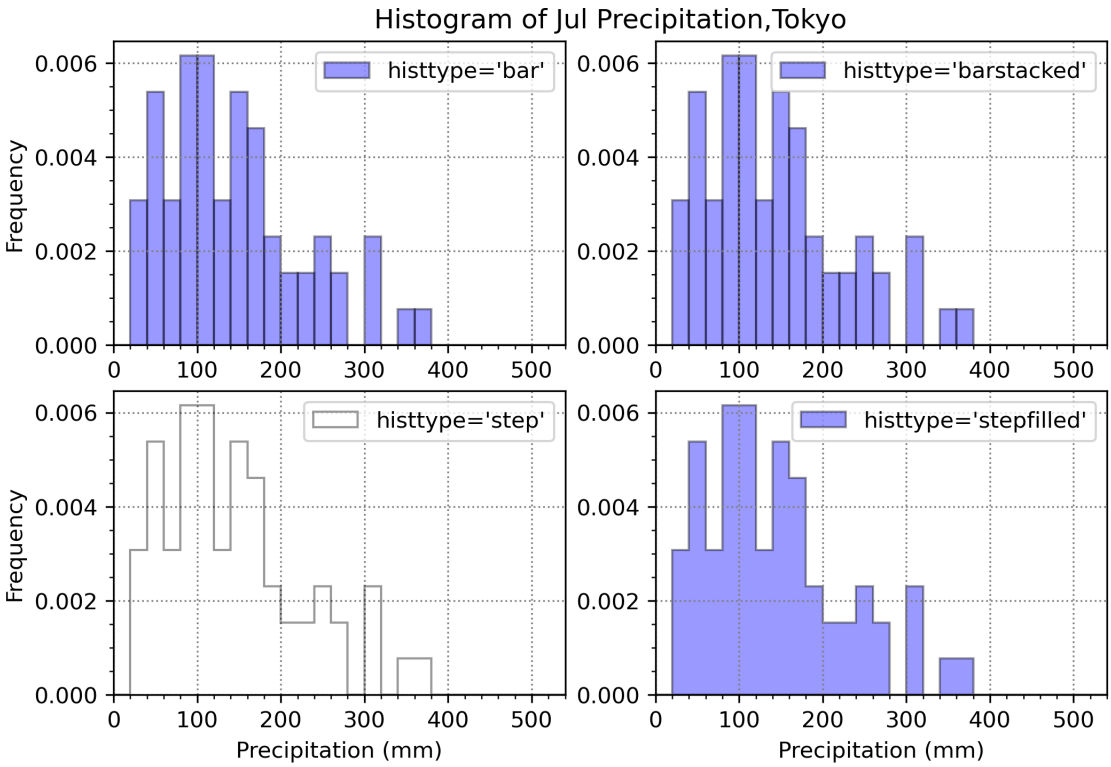


図４－４－12　東京のアメダス地点における７月の積算降水量を使い、histtype オプションだけを変えた

次のように STYLES で histtype のオプションを変え、ラベルにもオプションが表示されるようにしています。

```
STYLES = [
    dict(color='b', alpha=0.4, edgecolor='k', ¥
        histtype='bar', label="histtype='bar'"), #  棒状
    dict(color='b', alpha=0.4, edgecolor='k', ¥
        histtype='barstacked', label="histtype='barstacked'"), #  棒状積み上げ
    dict(color='b', alpha=0.4, edgecolor='k', ¥
        histtype='step', label="histtype='step'"), #  階段状
    dict(color='b', alpha=0.4, edgecolor='k', ¥
        histtype='stepfilled', label="histtype='stepfilled'") #  階段状塗り潰し
]
```

プロットエリアを定義してからサブプロットを作成する前に、全体のタイトルを付けています。このようにしておかないと、個別のサブプロットの方にタイトルが付いてしまいます。しかしデフォルトでは、プロットエリア全体で1つのグラフを作成し、その上にタイトルを描く設定になっており、全体のタイトルが付くと同時に、4枚のサブプロットに隣接するようにグラフ全体を囲うような枠線を描いてしまいます。それを防ぐテクニックとして、plt.axis('off')として枠線を描かないようにしています。どうなるか気になる場合は、plt.axis('off')をコメントアウトした上で作図を試してみましょう。

```
fig = plt.figure(figsize=(8, 6)) #  プロットエリアの定義（8x6）
plt.axis('off') #  枠線を消す
plt.title(title + ',' + sta) #  タイトルを付ける
```

4枚のサブプロットを2×2に並べるため、add_subplot(2,2,n+1)のように呼び出し、先ほどの作図と同様にループを回して add_subplot の3番目の引数に与える番号を変えています。ここでは ax=fig.add_subplot としているので、ループ毎に ax が更新され、ax.hist でヒストグラムを作図する際に、異なるサブプロットを参照することになっています。ヒストグラムを作成した後、x 軸のラベルを下側のサブプロットのみに、y 軸のラベルを左側のサブプロットのみに付けています。そのために、x 軸では n=2,3 で、y 軸では n=0,2 でラベルを付けるように、次のような if 文を使いました。y 軸の if not n % 2:は少し分かりにくいですが、0 は False、1 は True を意味するので、n % 2 が 0（偶数）なら False です。python の真偽値判定は分かりにくいので、表4－4－2にまとめておきます。

```
for n in np.arange(4):
    ax=fig.add_subplot(2,2,n+1) # 2x2 のサブプロット作成
    ax.hist(prep, density=True, bins=edges, **STYLES[n]) #  ヒストグラム
    …
    if n >= 2:
        plt.xlabel(xlabel) # x 軸のラベル（n=2,3 で下側のサブプロット）
    if not n % 2: # 0 で False、1 で True
        plt.ylabel(ylabel) # y 軸のラベル（n=0,2 で左側のサブプロット）
```

表４－４－２　python の真偽値一覧

型	真と判定	偽と判定
ブール値	True	False
文字列型	空以外	空文字列 '' や ""
数値型	0 や 0.0 以外	0 や 0.0
リスト型	空以外	空のリスト []
タプル型	空以外	空のタプル ()
辞書型	空以外	空の辞書{}
集合型	空以外	空の集合set()

次に２つのデータを用いて、同時にヒストグラムを作成してみます（図４－４－13）。６月と７月の積算降水量データを使いました。作図に用いたプログラムは、amedas_prep_histtype_JJ.py です。左上の図のように、histtype='bar'とした場合には同じビンの中でヒストグラムが横に並びます。先ほどとは違い２つのデータがあるので、histtype='barstacked'にした場合には右上のように積み上げ棒グラフになります。なお histtype='bar'であっても、stacked=True とした場合には、'barstacked'と同じ結果になります。左下は histtype='step'を用いて階段状のヒストグラム２つを同じビンに重ねて描いた場合ですが、枠線しか付かないので２つのヒストグラムの区別が困難です。このオプションを使う場合には、1 つのヒストグラムのみにした方が良いでしょう。histtype='stepfilled'で

階段状のヒストグラムに色を付けた場合、右下のように、2つのヒストグラムが重なると両方の色をブレンドしたような色として表示されるようです。配色を考えて作図すれば色の区別は付きますが、3つのヒストグラムと勘違いされやすいので避けた方が良さそうです。

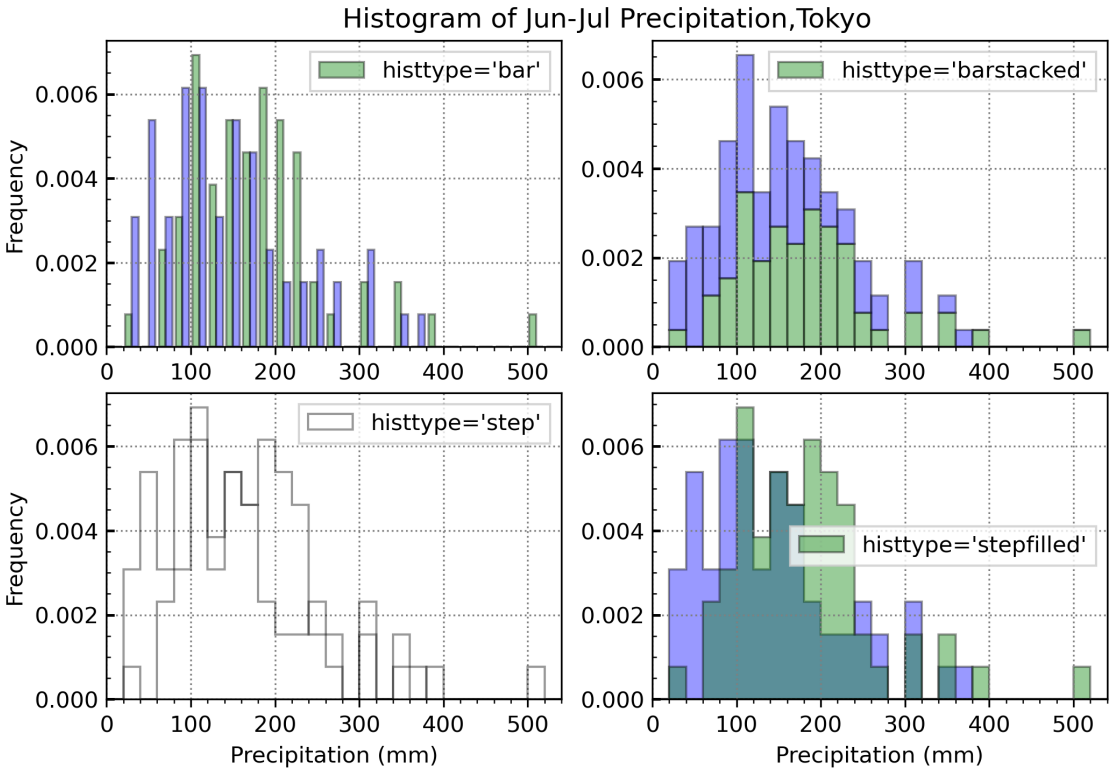


図4－4－13　東京のアメダス地点における6〜7月の積算降水量を使い、histtypeオプションだけを変えて2つのヒストグラムを重ねた

プログラム中では、month1 = "jun"、month2 = "jul"と定義し、6〜7月データをprep_iから取り出しています。

```
month1 = "jun"
month2 = "jul"
…
prep1 = prep_i.loc[syear:eyear,month1] # 6月降水量データ取り出し
prep2 = prep_i.loc[syear:eyear,month2] # 7月降水量データ取り出し
```

STYLES の定義では、color=['g', 'b']のようにリストで記述しておき、6 月と 7 月で色を変えられるようにしました。ラベルは 1 つの値だけなので、凡例になった時も 1 つしか表示されません。最初に描いた 6 月のグラフのマーカーが、凡例のマーカーとして利用されるようです。

```
STYLES = [
    dict(color=['g', 'b'], alpha=0.4, edgecolor='k', ¥
        histtype='bar', label="histtype='bar'"), #  棒状
    dict(color=['g', 'b'], alpha=0.4, edgecolor='k', ¥
        histtype='barstacked', label="histtype='barstacked'"), #  棒状積み上げ
    dict(color=['g', 'b'], alpha=0.4, edgecolor='k', ¥
        histtype='step', label="histtype='step'"), #  階段状
    dict(color=['g', 'b'], alpha=0.4, edgecolor='k', ¥
        histtype='stepfilled', label="histtype='stepfilled'") #  階段状塗り潰し
]
```

ヒストグラムを作図する ax.hist では、1 番目の引数で[prep1, prep2]のリストを渡して 2 つのデータを処理させています。

```
for n in np.arange(4):
    ax=fig.add_subplot(2,2,n+1) # 2x2 のサブプロット作成
    #  ヒストグラム
    ax.hist([prep1, prep2], density=True, bins=edges, **STYLES[n])
    …
```

4.5 散布図の作成

ここからは散布図の作図に移ります。まずはランダムなデータで作図方法を学び、後で気象データを使った作図を行います。

4.5.1 基本的な散布図

まずはランダムなデータを使って作図します（scat_rand.py）。data_x と data_y に 1000 個のデータをそれぞれ準備します。

```
size_of_sample = 1000 # サンプルサイズを設定
np.random.seed(1900) # 再現性を保つため、random state を固定
data_x = np.random.randn(size_of_sample) # x 軸のランダムデータ
data_y = np.random.randn(size_of_sample) # y 軸のランダムデータ
```

散布図の作成は plt.scatter(引数１, 引数２)で行います。1 番目の引数が x 軸のデータ、2 番目の引数が y 軸のデータに当たります。図４－５－１のような散布図が作成されます。

```
fig=plt.figure(figsize=(6, 6)) # プロット範囲（6x6）
ax=fig.add_subplot(1,1,1) # サブプロット作成
plt.scatter(data_x, data_y) # 散布図を描く
```

図４－５－１では、マーカーサイズが大きすぎて散布図の中心付近のマーカーが潰れてしまっています。

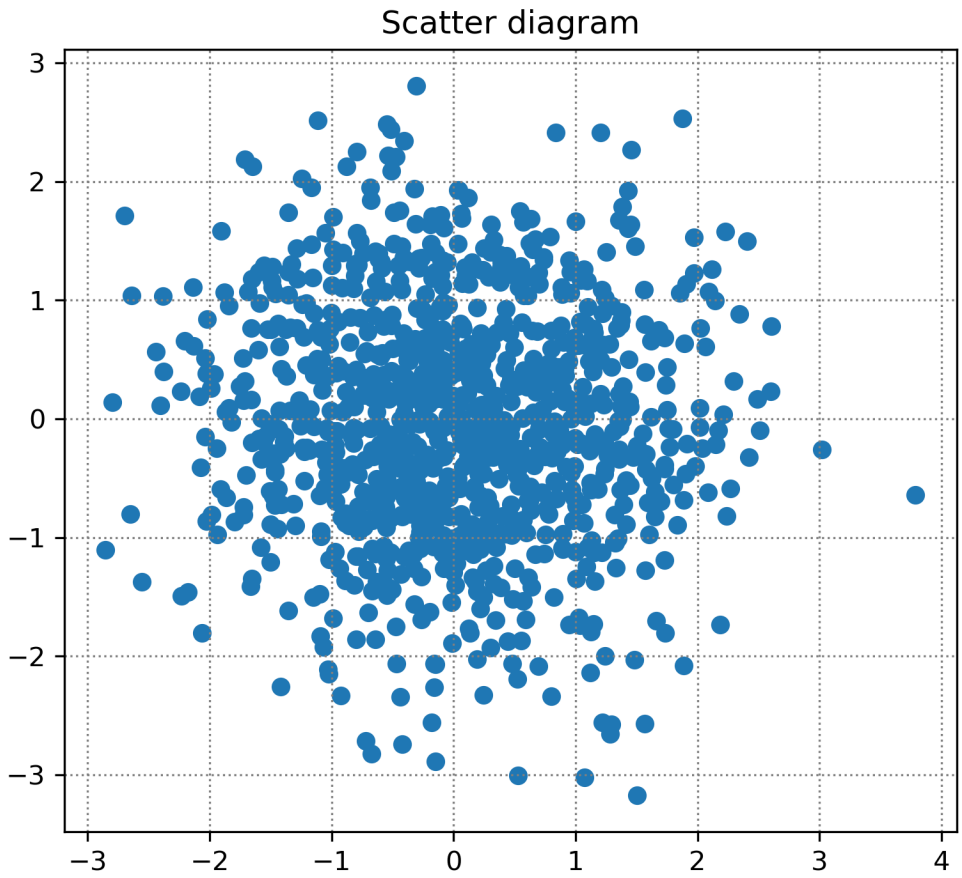


図４－５－１　1000 個のランダムデータから作成した散布図

4.5.2 マーカーの変更

マーカーサイズを変えるには、s オプションを使います（scat_rand2.py）。デフォルトのサイズは s=36 です。s=6 のように値を設定した場合、図４－５－２のようになり、マーカーを判別しやすくなりました。さらに、マーカーの色を color='k'（c='k'でも同じ）で黒にしています。マーカーの種類を marker='o'（円形）で指定しましたが、デフォルトが円形なので変化はありません。

```
plt.scatter(data_x, data_y, color='k', marker='o', s=6)
```

なお matplotlib の内部では、s オプションのデフォルト値はデフォルトパラメータの設定を参照しており、s=rcParams['lines.markersize']**2 のように計算されています。rcParams['lines.markersize']=6 なので、デフォルト値は s=36 です。

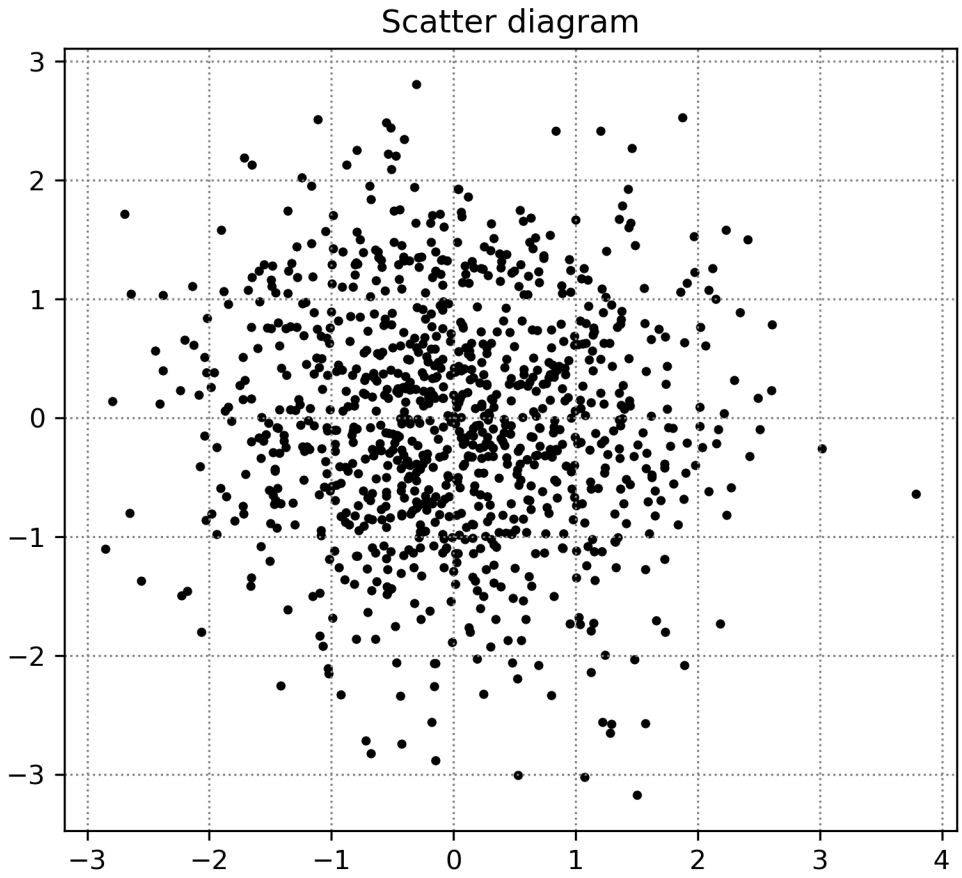


図４－５－２　マーカーの大きさと色を変更した

次にマーカーの種類を変更してみます（図４－５－３）。作図に用いたプログラムは scat_rand3.py です。マーカーを×印にするため、marker='x'としました。このままではマーカーサイズが若干小さいので、s=12 に変更します。

```
plt.scatter(data_x, data_y, color='k', marker='x', s=12)
```

他の図と同様、x 軸、y 軸の名前も設定可能です。

```
plt.xlabel(xlabel) # x 軸のラベル
plt.ylabel(ylabel) # y 軸のラベル
```

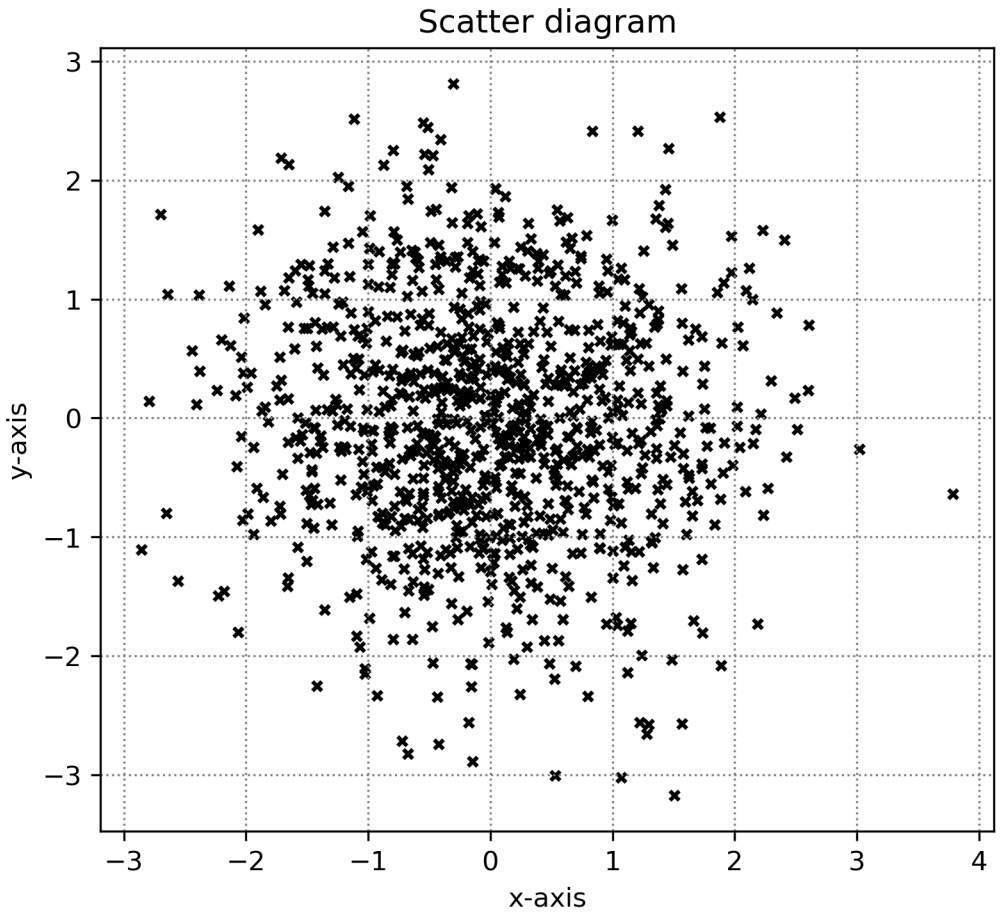


図４－５－３　マーカーの種類を変更

4.5.3 マーカーの色で値を表現

これまではx、y軸の2次元で散布図を描いていましたが、(x, y, z) の3次元データを使い、x、y軸にプロットしたデータのz軸上の値を色で表現したいこともあるでしょう。これまでのようにサンプル数が 1000 もあるデータをプロットすると、マーカー同士が重なって z 軸の値を判別しにくいため、ランダムに生成するサンプル数を 100 に減らした上で、z データを色で表現したものが図4－5－4です。作図に用いたプログラムは scat_rand4.py です。これまでのx、y軸データに加えて、z軸データもランダムに生成しました。

```
size_of_sample = 100 #  サンプルサイズを設定
…
data_x = np.random.randn(size_of_sample) # x 軸のランダムデータ
data_y = np.random.randn(size_of_sample) # y 軸のランダムデータ
data_z = np.random.randn(size_of_sample) # z 軸のランダムデータ
```

データを描く範囲を−4〜4までに限定しています。

```
xmin = -4 # x 軸範囲の下限
xmax = 4 # x 軸範囲の上限
ymin = -4 # y 軸範囲の下限
ymax = 4 # y 軸範囲の上限
```

散布図を作図する部分です。z軸のデータを c=data_z で渡します。c オプションは色の指定ですが、連続した色の値を ndarray で渡すこともできて、その時にはマーカーに色を付けるようになっています。色の付け方は cmap で指定します。cmap='Blues'で青系の色が付きます。

```
plt.scatter(data_x, data_y, marker='o', s=24, c=data_z, cmap='Blues')
```

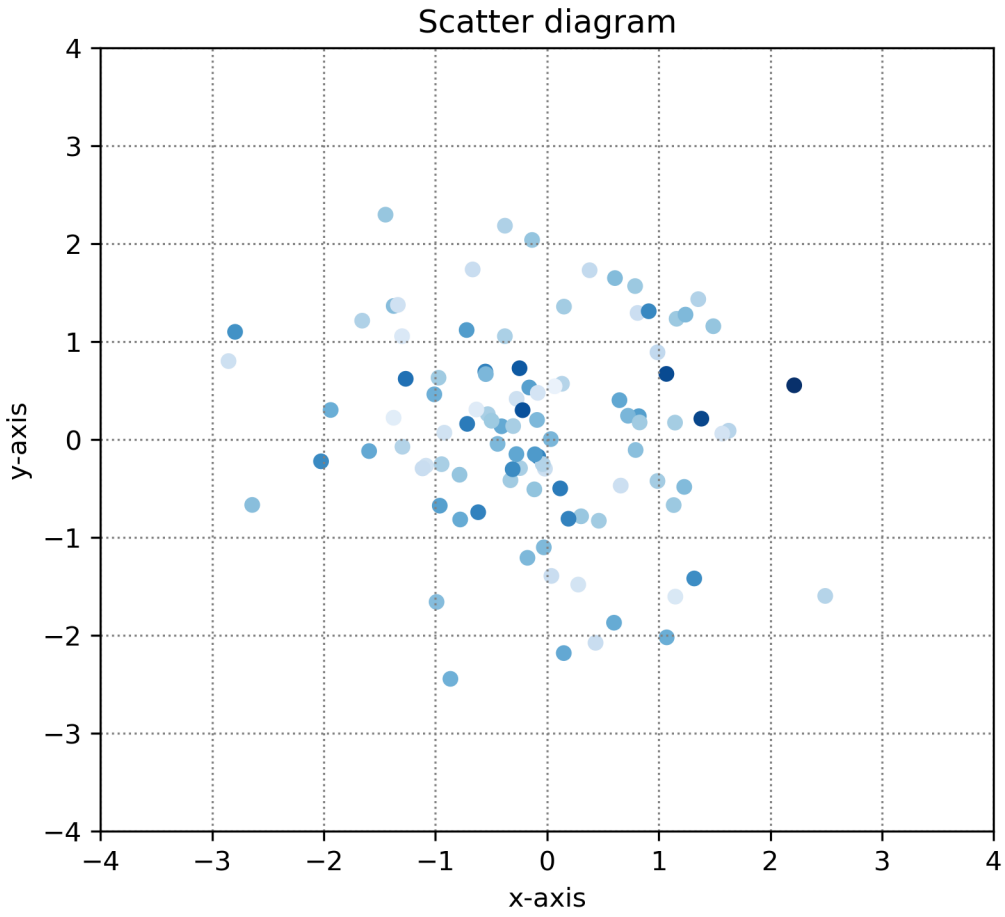


図４－５－４　ｚ軸のデータを色で表現した

4.5.4 図の体裁を整える

もう少し見やすい図にするため、マーカーの枠線を付け、カラーバーを追加します（図４－５－５）。さらに軸のラベルやタイトルを大きくし、x=0、y=0 の線を付け、目盛線を内側にして小目盛を付けます。作図に用いたプログラムは scat_rand5.py です。マーカーの枠線を付けるオプションが edgecolor で、edgecolor='k'（黒）のように枠線の色を指定します。ここでは使っていませんが、枠線の幅を指定する linewidth（lw でも同じ）を指定することもできます。デフォルトでは現在と同じ lw=1 なので、枠線を太くしたい場合に使います。

```
cs = plt.scatter(data_x, data_y, marker='o', s=24, c=data_z, ¥
                    cmap = 'Blues', edgecolor='k')
```

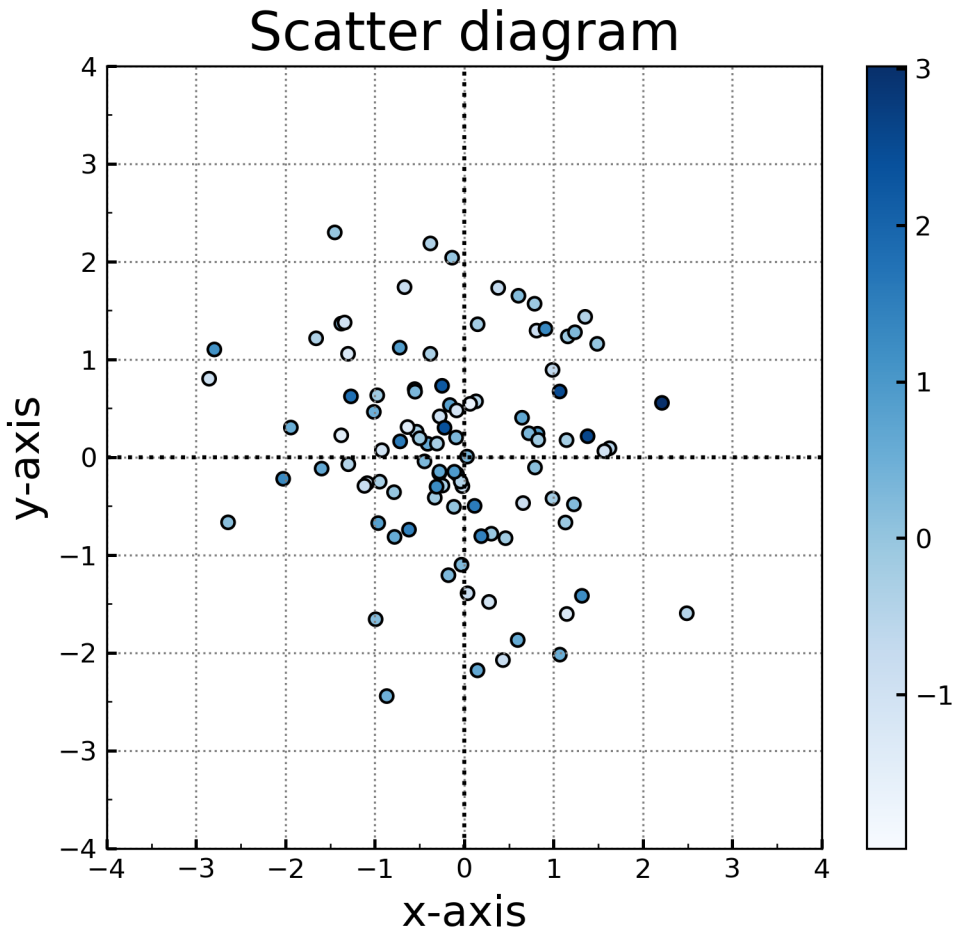


図４－５－５　図の体裁を整えカラーバーも付けた

plt.scatter の戻り値 cs は、カラーバーを付ける時に使います。カラーバーを付けるのは fig.colorbar です。デフォルトでは図の右側に縦棒が付きます。

```
fig.colorbar(cs)
```

3.5 節で紹介した次の方法で軸のラベルやタイトルを大きくすると、バランスの良い図になったように見えます。

```
plt.title(title, fontsize=20) #  タイトルを付ける
plt.xlabel(xlabel, fontsize=16) # x 軸のラベルを付ける
plt.ylabel(ylabel, fontsize=16) # y 軸のラベルを付ける
```

x=0、y=0 の場所が分かりにくいので、線をつけておきます。

```
plt.axvline(x=0, color='k', ls=':') # x=0 の線を付ける
plt.axhline(y=0, color='k', ls=':') # y=0 の線を付ける
```

目盛線の付け方は好みの問題ですが、4.3.2 節で行った方法で内側につけています。

```
plt.rcParams['xtick.direction'] = 'in' # x 軸目盛線を内側
plt.rcParams['xtick.major.width'] = 1.2 # x 軸大目盛線の長さ
plt.rcParams['ytick.direction'] = 'in' # y 軸目盛線を内側
plt.rcParams['ytick.major.width'] = 1.2 # y 軸大目盛線の長さ
```

大目盛線の間隔を 1.0、小目盛線の間隔を 0.5 として目盛線をつけます。

```
ax.xaxis.set_major_locator(ticker.MultipleLocator(1.0)) # x 軸大目盛の間隔
ax.xaxis.set_minor_locator(ticker.MultipleLocator(0.5)) # x 軸小目盛の間隔
ax.yaxis.set_major_locator(ticker.MultipleLocator(1.0)) # y 軸大目盛の間隔
ax.yaxis.set_minor_locator(ticker.MultipleLocator(0.5)) # y 軸小目盛の間隔
```

4.5.5 データ範囲の指定

ここまでは、色を自動で付けており、表示する最小値から最大値までの範囲指定は行いませんでした。作図の際には、プラスとマイナスで色を変え色の濃さで値の違いを表現するといった、データ範囲の指定が必要な場合もあると思います。scat_rand6.py は、そのような作図を行うプログラムで、図４－５－６のように 0 付近が白でマイナス側が青系の色、プラス側が赤系の色で表現されます。色は cmap オプションで指定し、cmap='bwr'で青～白～赤のように変化するような色テーブルを利用します。色を描く範囲を指定するオプションが vmin、vmax オプションで、−4～4 の範囲にしたので負の側が青、0 付近が 0、正の側が赤になりました。他にも様々な色テーブルが利用可能で、図４－５－７に一覧を載せています。色テーブルの名前は一覧の左に記述されており、色テーブルは値の小さい方から大きい方に向け左側から右側に色が変化していくことを表しています。なお、色テーブルの名前に_r を付けると色の変化を逆にすることができ、例えば cmap='bwr_r'にした場合、赤～白～青のように変化します。

```
zmin=-4 #  データ範囲の下限
zmax=4 #  データ範囲の上限
…
cs = plt.scatter(data_x, data_y, marker='o', s=24, c=data_z, cmap='bwr', ¥
                    edgecolor='k', vmin=zmin, vmax=zmax)
```

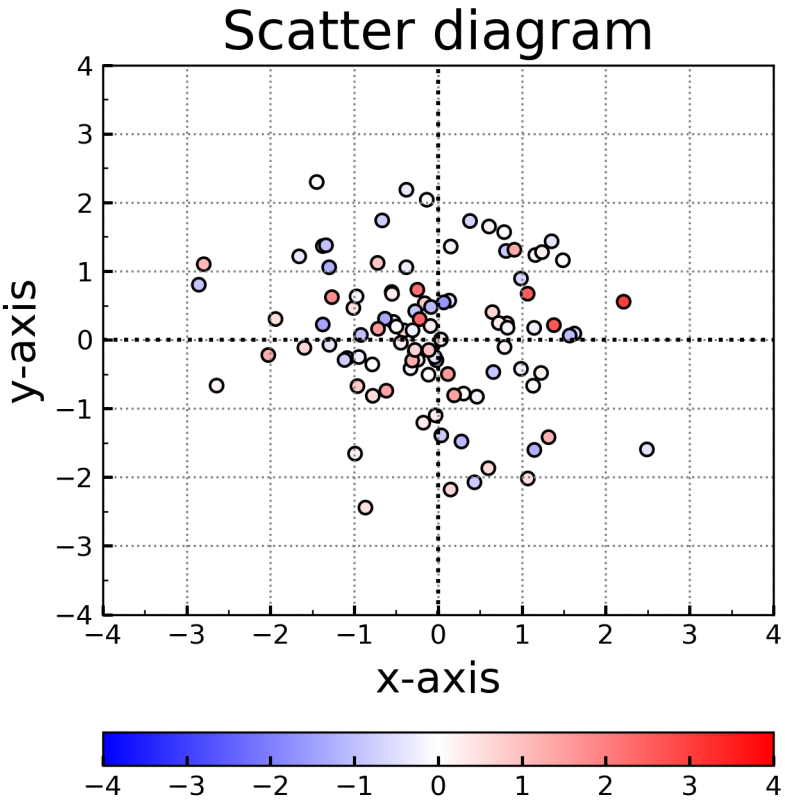
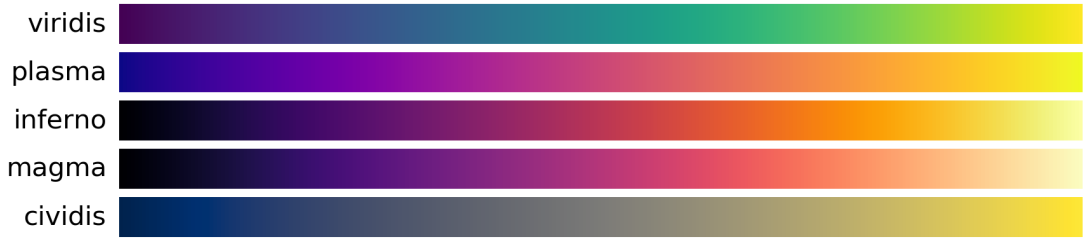


図４－５－６　データ範囲を指定しマイナスを青系、プラスを赤系の色で表現した

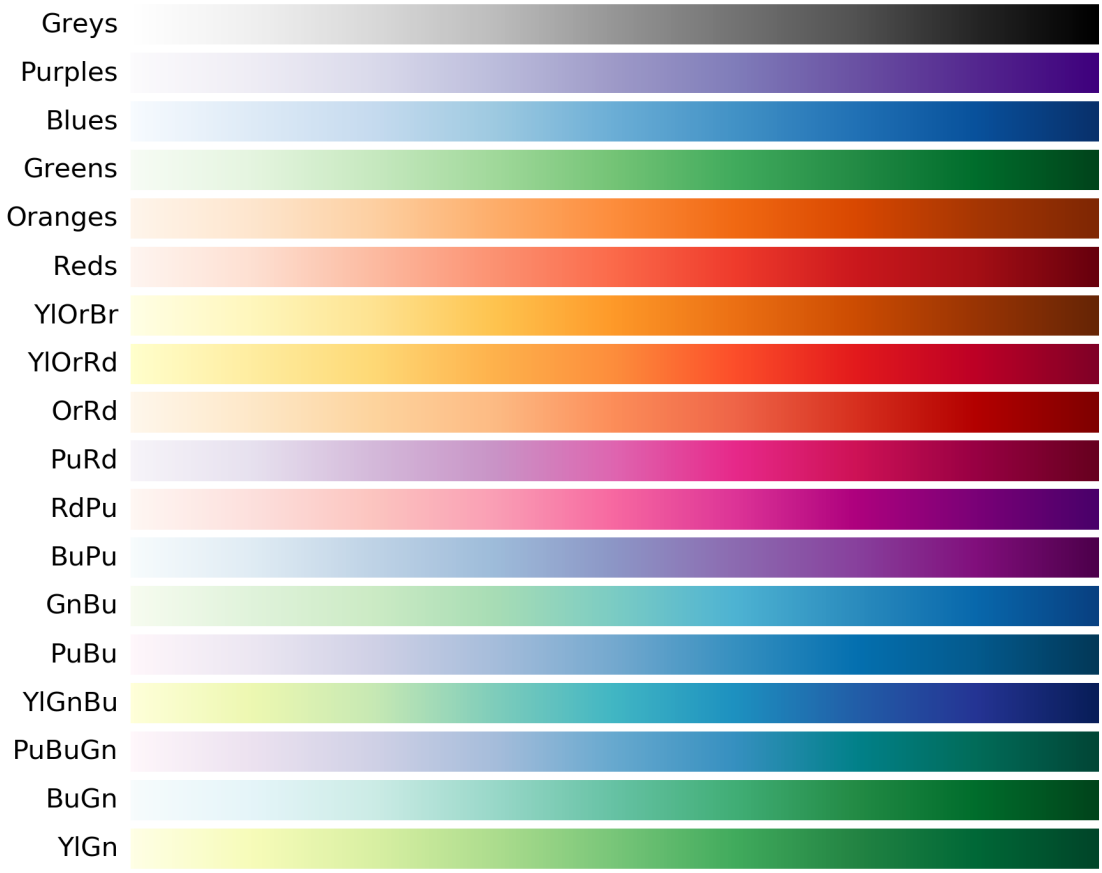
凡例の場所が下側になり、横棒になっているのに気が付いたでしょうか。今回は fig.colorbar のオプションに orientation='horizontal'を指定していたので、このように凡例の方向が横になりました。

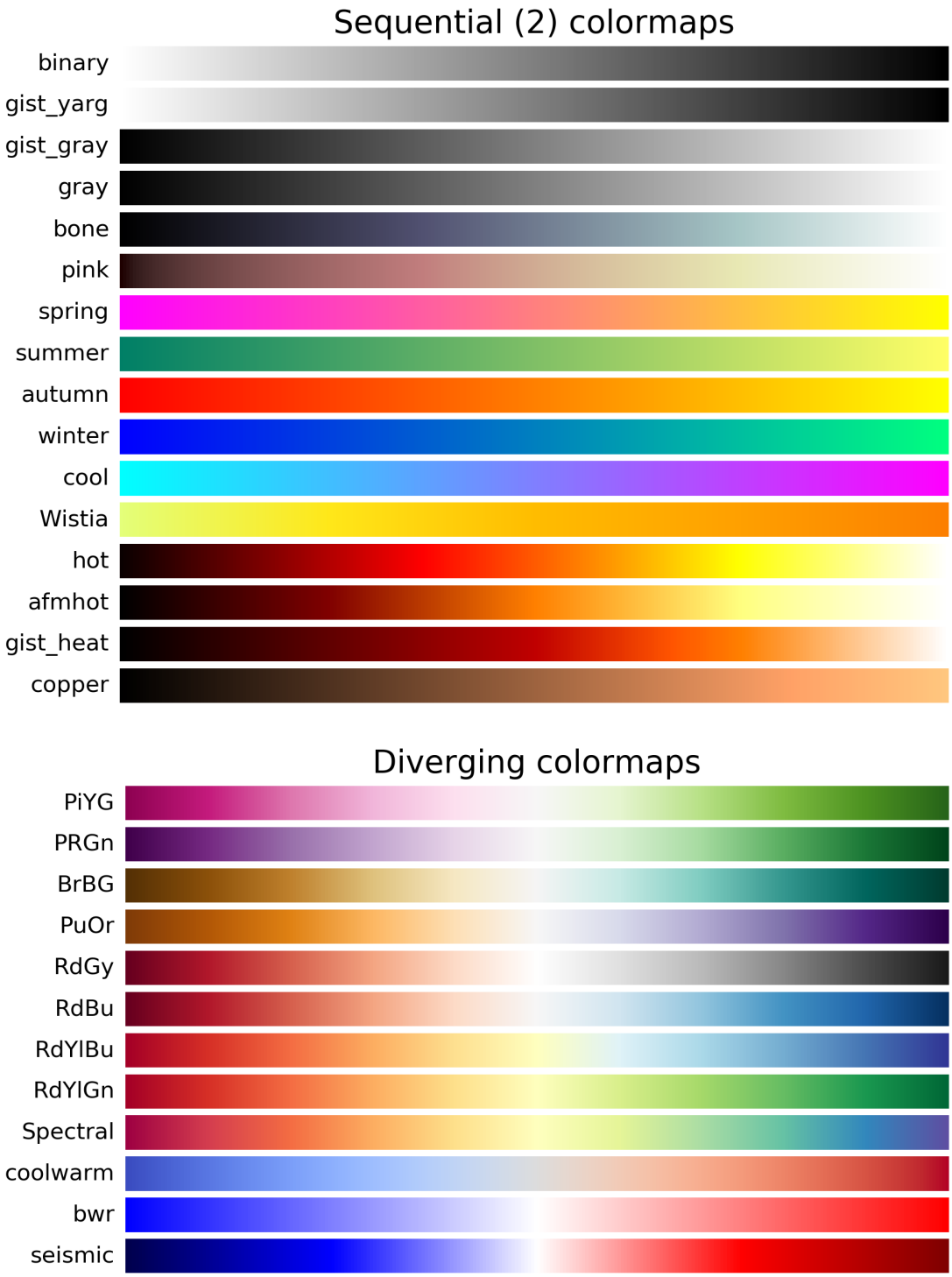
```
fig.colorbar(cs, orientation='horizontal')
```

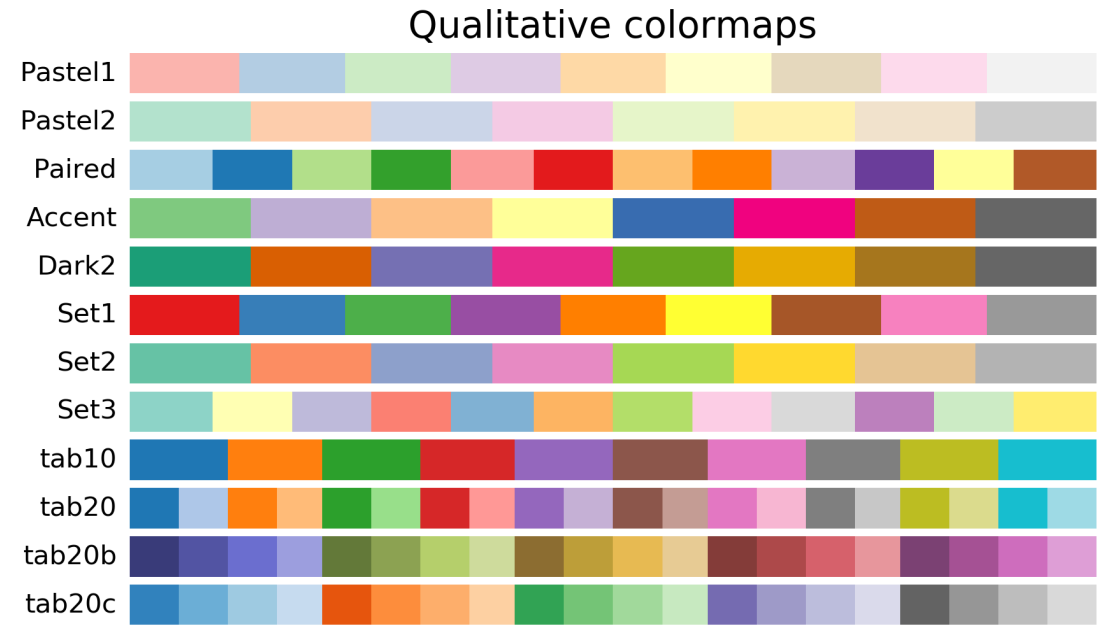
Perceptually Uniform Sequential colormaps



Sequential colormaps




Sequential (2) colormaps
binary
gist_yarg
gist_gray
gray
bone
pink
spring
summer
autumn
winter
cool
Wistia
hot
afmhot
gist_heat
copper
Diverging colormaps
PiYG
PRGn
BrBG
PuOr
RdGy
RdBu
RdYlBu
RdYlGn
Spectral
coolwarm
bwr
seismic



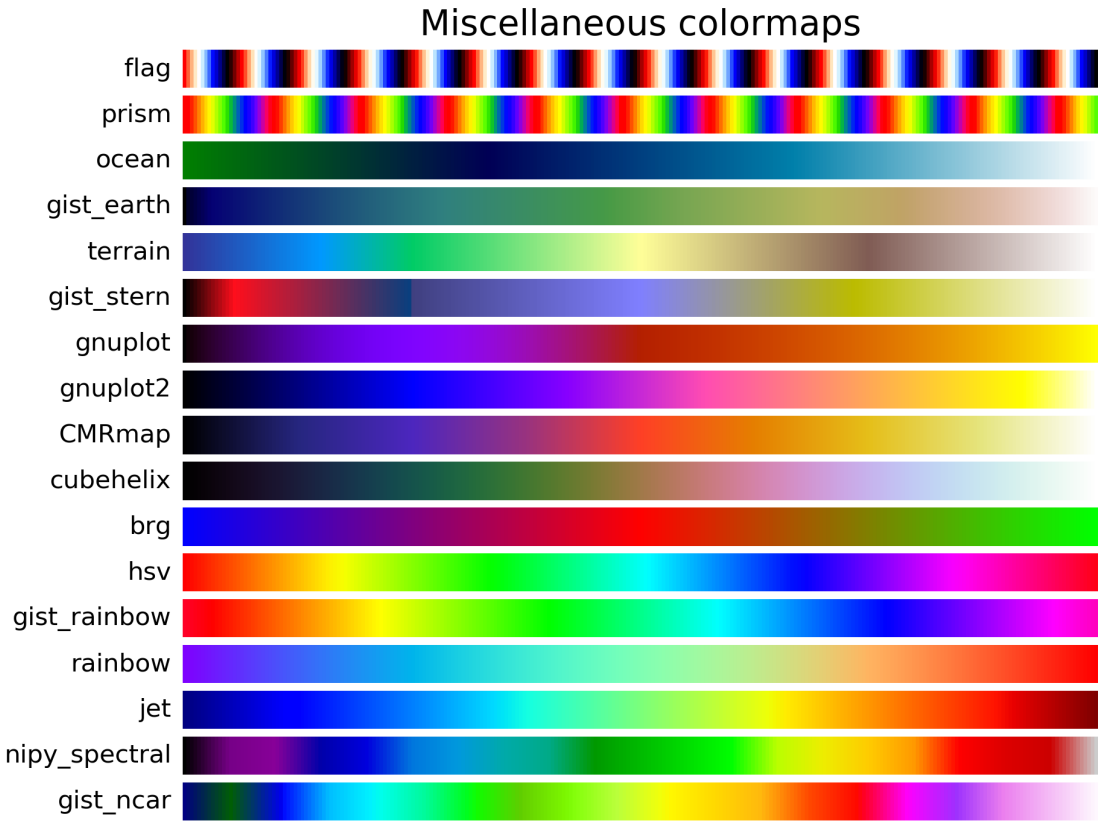


図４－５－７　色テーブルの一覧

4.5.6 複数のデータを同時に描く

観測地点や季節、期間などが異なる複数のデータがありデータ間の違いを比較する場合など、データ毎にマーカーの色を変えて描きたいこともあると思います。scat_double_rand.py は、平均と標準偏差の異なる 2 つのランダムデータを使いマーカーの色を変えて同時に描くプログラムです。用いた data-1 は x 軸、y 軸共に平均 50、標準偏差 10 のデータで、data-2 は平均 30、分散 4 のデータです。赤色のマーカーで表した data-1 が右上に、青色のマーカーで表した data-2 が左下に分布しているのが分かると思います（図４－５－８）。

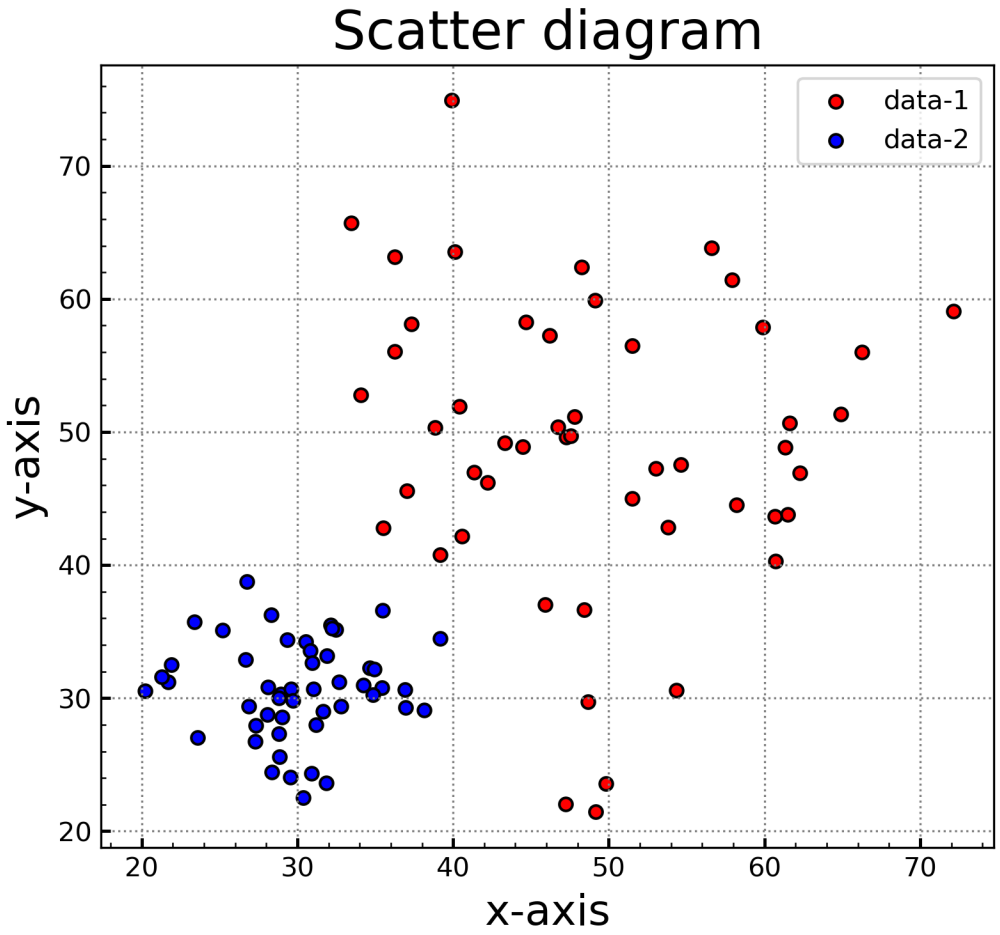


図４－５－８　２つのデータを同時に描く

データの生成は次のように行なっています。np.random.randn で生成される乱数が平均 0、標準偏差 1 であることを利用し、生成したい標準偏差の値を乱数

に掛け平均値を足しています。生成された data-1 に対応するのが data_x1 と data_y1、data-2 に対応するのが data_x2 と data_y2 です。

```
size_of_sample = 50 #  サンプルサイズを設定
mu1, sigma1 = 50, 10 # data-1 の平均、標準偏差の指定
mu2, sigma2 = 30, 4 # data-2 の平均、標準偏差の指定
…
#  ランダムデータ作成
data_x1 = mu1 + sigma1 * np.random.randn(size_of_sample) # data-1 (x)
data_y1 = mu1 + sigma1 * np.random.randn(size_of_sample) # data-1 (y)
data_x2 = mu2 + sigma2 * np.random.randn(size_of_sample) # data-2 (x)
data_y2 = mu2 + sigma2 * np.random.randn(size_of_sample) # data-2 (y)
```

散布図の作成部分では、plt.scatter に渡す x 軸、y 軸のデータを変えて、data-1 の散布図を描く処理と data-2 の散布図を描く処理を別々に行います。異なった色で塗り潰すため、オプションとして data-1 に対しては color='r'（赤色）、data-2 に対しては color='b'（青色）を指定しました。マーカーについては、いずれも丸マーカーで黒色の枠線を付けるようにしています。

```
plt.scatter(data_x1, data_y1, color='r', edgecolor='k', marker='o', ¥
            s=24, label='data-1')
plt.scatter(data_x2, data_y2, color='b', edgecolor='k', marker='o', ¥
            s=24, label='data-2')
```

作図の際に label オプションを指定したので、plt.legend で凡例が生成されています。

```
plt.legend(loc='best') #  凡例を描く
```

4.5.7 相関係数を表示する

散布図と一緒に相関係数を表示したいこともあるかと思います。簡単にするため、1 つのランダムデータだけで作図した散布図に相関係数を表示します（図４－５－９）。ここでは、x 軸方向に平均 50、標準偏差 2、y 軸方向に平均 30、標準偏差 5 の正規分布となるような標本を使いました。x 軸、y 軸共にランダムデータなので、相関係数 r=-0.097 とほぼ無相関です。作図に用いたプログラムは、scat_rand+corr.py です。まず x、y 軸方向に別々の平均、標準偏差のデータを生成します。

```
size_of_sample = 50 #  サンプルサイズを設定
mu_x, sigma_x = 50, 2 # x 軸方向の平均、標準偏差
mu_y, sigma_y = 30, 5 # y 軸方向の平均、標準偏差
…
#  ランダムデータの生成
data_x = mu_x + sigma_x * np.random.randn(size_of_sample)
data_y = mu_y + sigma_y * np.random.randn(size_of_sample)
```

作図部分では、赤の丸マーカーに黒色の枠線を付け、さらに半透明にするため、alpha=0.4 を指定しました。

```
plt.scatter(data_x, data_y, color='r', edgecolor='k', marker='o', ¥
                s=24, alpha=0.4)
```

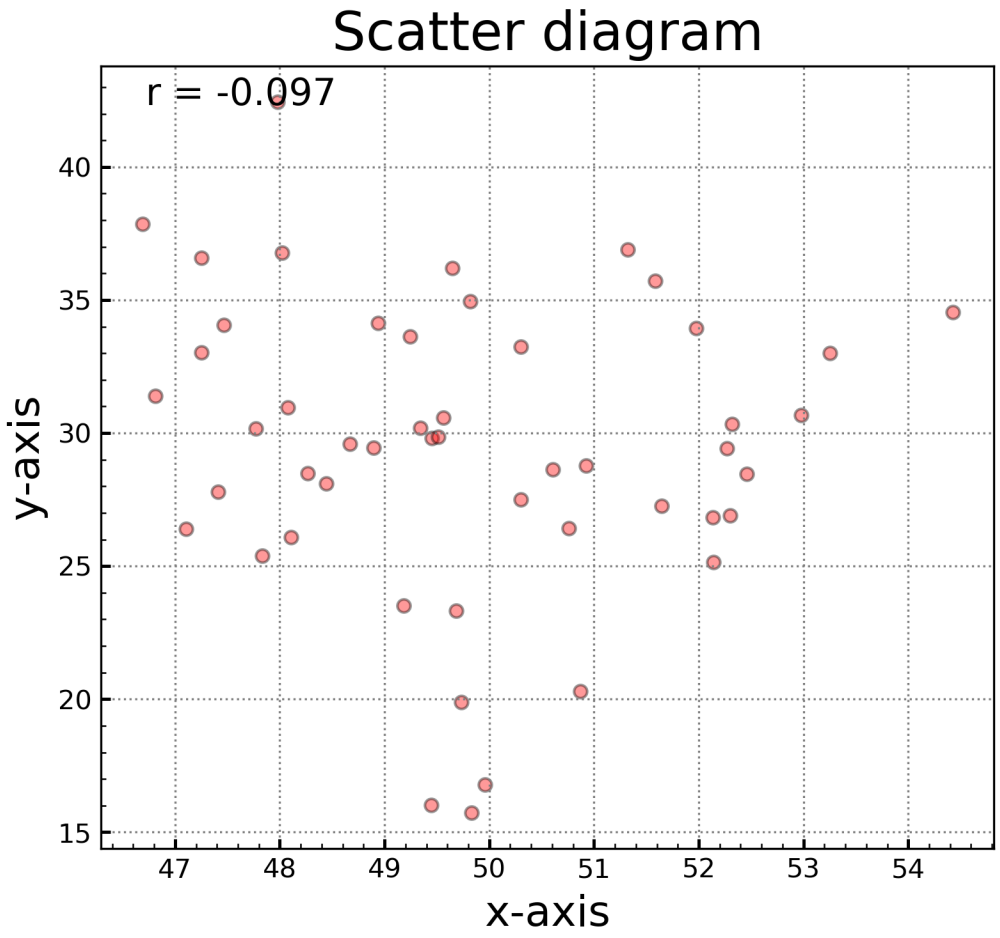


図４－５－９　散布図に相関係数を表示した

次の１行目は相関係数を計算する部分です。Numpy に np.corrcoef(data-1, data-2)というツールがあり、一般的に使われるピアソンの積率相関係数を計算することができます。戻り値は２次元の ndarray で返ってきており、それを corr に入力します。corr[0,0]と corr[1,1]は data-1 同士、data-2 同士の相関係数（r=1）なので、ここでは使いません。corr[1,0]と corr[0,1]は、data-1 と data-2 の相関係数なので、こちらを使います。corr[1,0]と corr[0,1]は全く同じ値なので、どちらを使っても構いません。2 行目は表示するテキストを設定して name に格納するためのもので、"データの表示順序と書式".format(データの中身)のように記述します。データの表示順序は、記述した順になり、{s:s}{f:.3f}は s という変数を文字型で、f という変数を浮動小数点型として小数点以下を３文字で表示します。データの中身は、s="r ＝ ", f=corr[1,0]なので、最初の文字列は r ＝ 、

次の浮動小数点数は corr[1,0]の中身です。小数点以下を 3 文字で表示するので、-0.097 のようになります。最終的に r = -0.097 のような文字列が設定されます。

```
corr = np.corrcoef(data_x, data_y) #  相関係数の計算
name = "{s:s}{f:.3f}".format(s="r = ", f=corr[1, 0]) #  表示するテキスト設定
```

古い書式でもエラーにはならないので、新しい書式に慣れない場合は、次の記法でも問題ないです。この場合、%.3f が浮動小数点型で小数点以下が 3 文字であることを意味します。%で区切った後ろ側が入力されるデータです。

```
name = "r=%.3f" % corr[1, 0] #  古い書式
```

なお、浮動小数点型の書式は{f:4.3f}のようにピリオドの前に数字を入れることもでき、全体の長さが最低 4 文字であることを表します。この例では相関係数の値が 6 文字で、4.3f の書式で指定した最低 4 文字を超えているため、実際の長さの 6 文字になります。他方、最低 8 文字を表す 8.3f のように 6 文字を超える値に設定すると、-0.097 の前に 2 文字分の空白が入ります。

設定された r = -0.097 の文字列をどこに表示するのかを計算するために、plt.axis()で軸の範囲を取得します。戻り値は x 軸下限（xmin）、x 軸上限（xmax）、y 軸下限（ymin）、y 軸上限（ymax）の順に返ってくるので、それぞれの値を格納します。この記述は、プログラムの最初で入力している xmin、xmax、ymin、ymax のいずれかに None が入っていた場合に範囲が正しくならないことを避けるためです。plt.axis()で得られた x 軸、y 軸の範囲を使い、x 軸の下限から 5%、y 軸の上限から 5%の位置を xloc、yloc にそれぞれ設定します。xloc、yloc と先ほど設定した name を plt.text に渡し、相関係数「r = -0.097」の文字列を左上に表示します。

```
xmin, xmax, ymin, ymax = plt.axis() #  軸の範囲の取得
xloc = xmin + 0.05 * (xmax - xmin) # x 軸上の座標
yloc = ymax - 0.05 * (ymax - ymin) # y 軸上の座標
plt.text(xloc, yloc, name, fontsize=14, color='k') #  相関係数を表示
```

ほぼ無相関のサンプルだけでは面白くないので、正の相関となるようなランダムサンプルを生成して、散布図と相関係数を表示してみます（図４－５－10）。作図に用いたプログラムは、scat_rand+corr2.py です。

正の相関となるようなサンプルを生成するために、工夫をしています。まず先ほど同様、x 軸方向に平均 50、標準偏差 2、y 軸方向に平均 30、標準偏差 5 の正規分布のサンプルを生成しますが、その値に赤色で示した 1 次の項を加えています。例えば x 軸方向には、-25 < x < 25 で a_x * x の変化をする
a_x * np.linspace(-size_of_sample/2, size_of_sample/2)
が加えられています。y 軸方向も同様です。ここに a_x = 0.8、a_y = 1.2 の係数が掛かっているので、x 軸方向に対して y 軸方向が 1.5 倍で変化するような直線の周辺に分布します。

```
size_of_sample = 50 #  サンプルサイズを設定
mu_x, a_x, sigma_x = 50, 0.8, 2 # x 軸方向の平均、標準偏差
mu_y, a_y, sigma_y = 30, 1.2, 5 # y 軸方向の平均、標準偏差
…
#  ランダムデータの生成
data_x = mu_x + a_x * np.linspace(-size_of_sample / 2, size_of_sample / 2) ¥
          + sigma_x * np.random.randn(size_of_sample)
data_y = mu_y + a_y * np.linspace(-size_of_sample / 2, size_of_sample / 2) ¥
          + sigma_y * np.random.randn(size_of_sample)
```

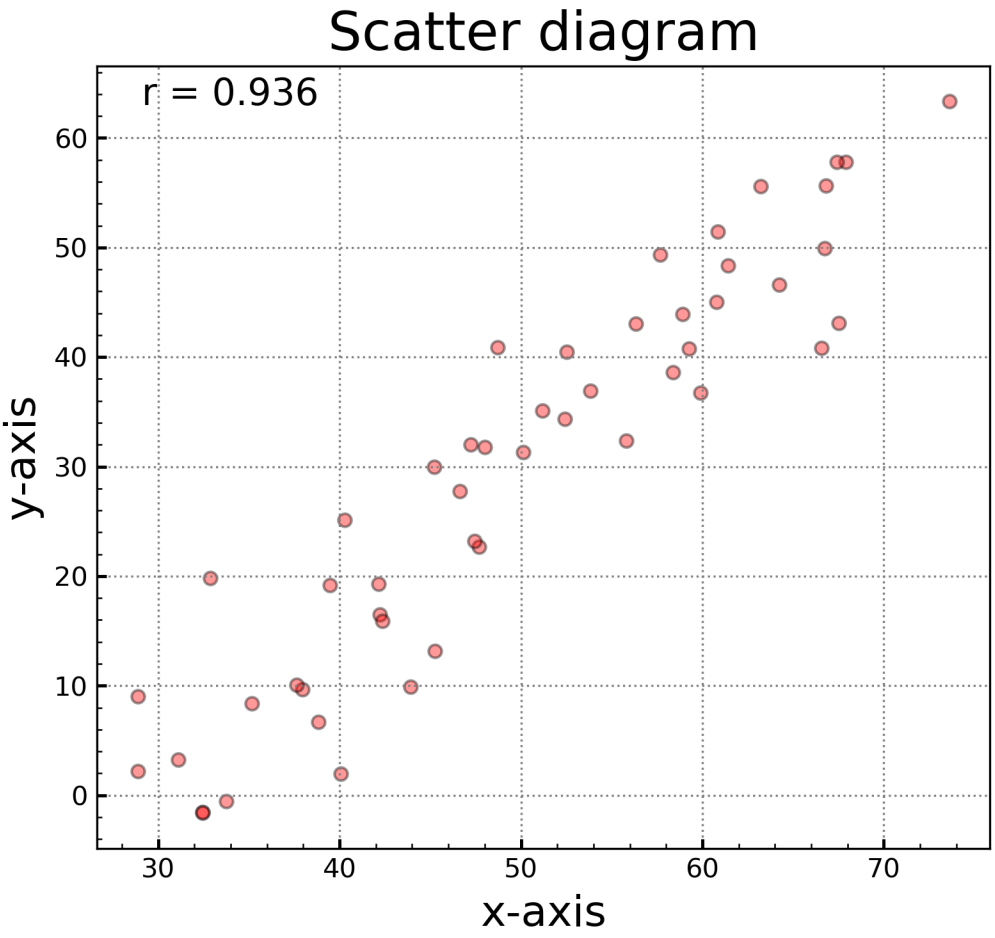


図４－５－10　正の相関を持つようなサンプルから散布図を作成した

4.5.8 回帰直線の追加

散布図に近似式を重ねた図を見たことがあると思います。前節の図４－５－10の散布図に線形回帰した回帰直線を追加してみます（図４－５－11）。scat_rand+corr3.py が作図を行うプログラムです。

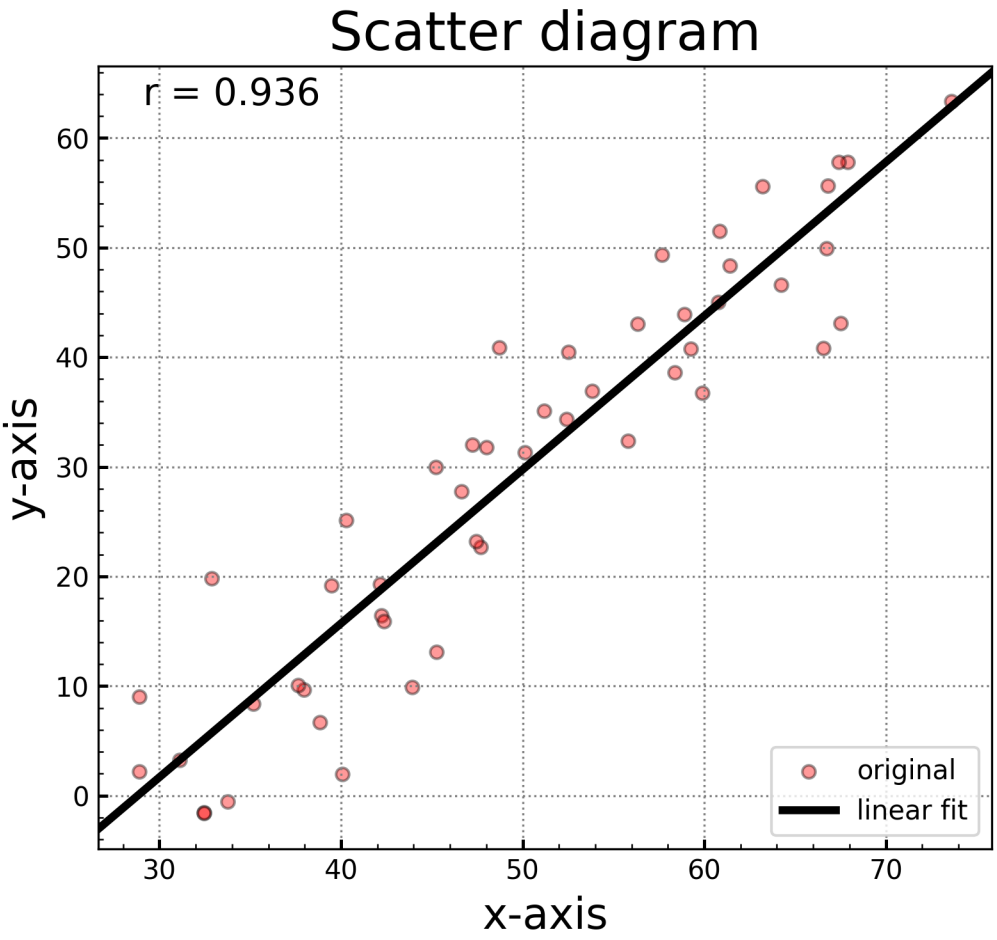


図４－５－11　回帰直線を加えた

標本 x と標本 y の相関係数（r）と回帰係数（a）の関係は、標本 x の標準偏差 σ_x と標本 y の標準偏差 σ_y を用いて $r = a(\sigma_x / \sigma_y)$ のように表されます。標準偏差は data_x.std()と data_y.std()で求めることができます。まず、標本 x と標本 y の標準偏差、及び、先ほどと同様の方法で計算しておいた相関係数を使い回帰係数を求めます。

```
a = corr[1,0] * data_y.std() / data_x.std() #  回帰係数（a）
```

また、切片（b）は b = Y − aX（X、Y は標本 x と標本 y の算術平均）です。

```
b = data_y.mean() - a * data_x.mean() #  切片（b）
```

次に、y=ax+b の回帰式を計算します。プログラム中では、回帰式の x 座標は x1 という ndarray、y 座標は y1 という ndarray に格納しており、それぞれ size_of_sample 分だけ生成します。まず np.linspace(**開始点**、**終了点**、**個数**)とすることで x 軸側データを生成します。その際に x 軸上の作図範囲の下限 xmin よりも小さい最大の整数を np.floor(xmin)、作図範囲の上限 xmax よりも大きい最小の整数 np.ceil(xmax)を、それぞれ開始点、終了点にしています。y 軸側データは y1 = a * x1 + b で生成します。このとき y 軸側データの個数は、x 軸側データの個数と同じになります。最後に plt.plot で回帰式をプロットします。

```
#  回帰式の計算
x1 = np.linspace(np.floor(xmin), np.ceil(xmax), size_of_sample)
y1 = a * x1 + b
plt.plot(x1, y1, color='k', lw=3, label='linear fit') #  回帰式のプロット
```

4.5.9 マーカーと線を重ねる順序

図４－５－11 をよく見ると、マーカーの上に線が重なって隠されてしまっています。この図ではマーカーが半透明で分かりにくいので、alpha=1.0 で不透明にした状態のものが図４－５－12 です。プログラムは scat_rand+corr4.py です。

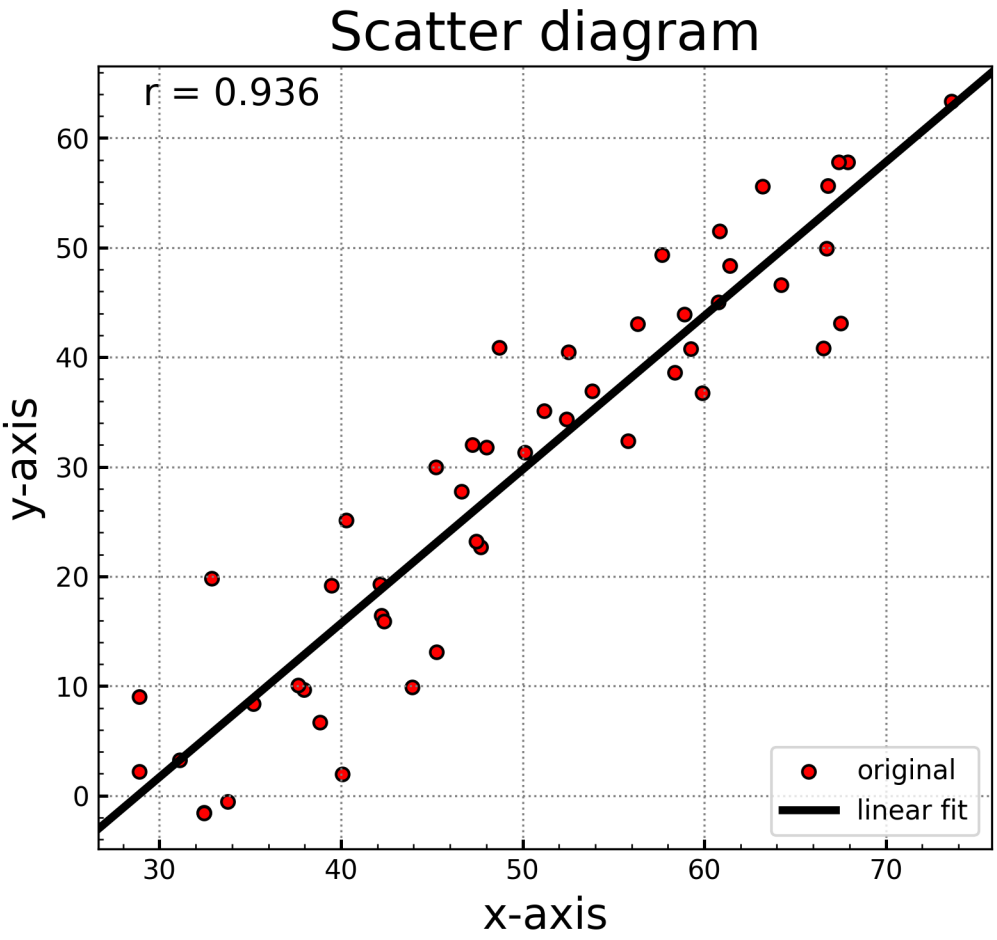


図４－５－12　マーカーが後から描いた線に隠れている

マーカーと線を重ねる順番を指定する方法があり、マーカーを上に重ねれば回避可能です。次のように、plt.plot や plot.scat には zorder=整数というオプションがあり、数字が大きい方が後から描かれます。このような指定を行うプログラムが scat_rand+corr5.py で、図４－５－13 のように。マーカーの順番を回帰直線よりも上にするため、散布図を作成するときに zorder=2 とします。

```
plt.scatter(data_x, data_y, color='r', edgecolor='k', marker='o', s=24, ¥
            alpha=0.4, label='original', zorder=2)
```

回帰直線をプロットする際には、zorder=1 とします。

```
plt.plot(x1, y1, color='k', lw=3, label='linear fit', zorder=1)
```

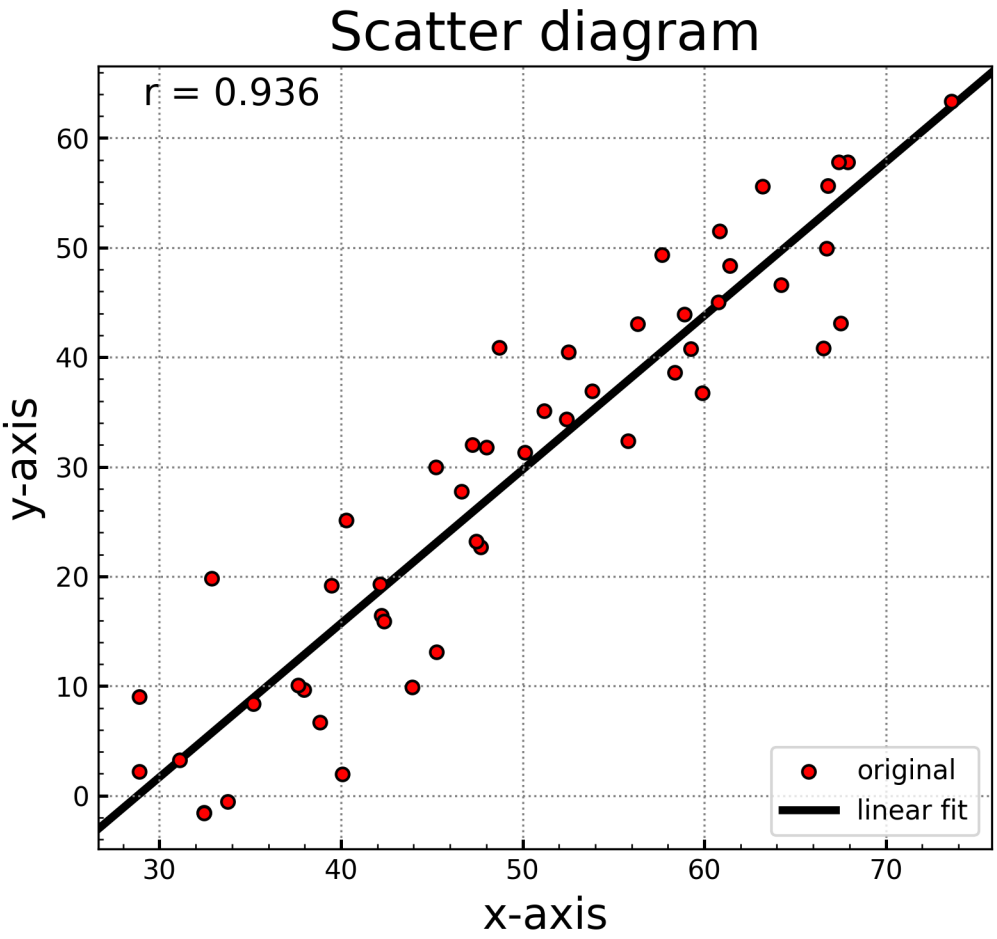


図４－５－13　マーカーが線の上に来るようにした

4.5.10 意味のある無相関、意味のない相関

ランダムデータを使った作図の最後に、極端なサンプルを使った散布図の作図を試してみます。作図に用いたプログラムは、scat_rand+corr6-1.py から scat_rand+corr6-4.py です。図４－５－14 には左上、右上、左下、右下の順に、6-1 から 6-4 のプログラムで作成された図を載せています。左側はほぼ無相関のサンプルで、右側は相関係数が大きいサンプルを表しています。相関係数の値だけを見ると、左側は無相関で右側は正の相関になっているので、あたかも右側のサンプルに意味があるように見えてしまいます。しかし、散布図から分布のパターンを読み取ってみると、左上は U 字型（実際は cos(x)の周辺に分布させたサンプル）、右上は x 軸の値が大きい側だけ y 軸の値が増加しているサンプル（exp(x)の周辺に分布させたサンプル）、左下は４箇所に分布の中心があるサンプル、右下は 1 点だけ突出しているサンプルです。このように散布図を描いてみると、散布図には相関係数では見えてこない様々な情報を含みうることが分かると思います。

まず左上の図を作成するために用いたサンプルの生成方法を見ていきます。scat_rand+corr6-1.py では、次のように y=cos(x)の周辺に分布するようなサンプルを生成しています。x 軸、y 軸側ともに、y=cos(x)からあまり離れた分布をしないように、誤差の標準偏差を 0.1 に設定します。a_y=1.0 なので、y1 = np.cos(2*np.pi*x1)で cos(x)に沿った y 軸の値にしています。図４－５－14 の左上は無相関ですが、相関係数だけで判断すれば y=cos(x)に従った分布という意味のある関係を見逃してしまいます。

```
mu_x, a_x, sigma_x = 50, 1.0, 0.1 #  標本 x の平均、係数、誤差の標準偏差
mu_y, a_y, sigma_y = 30, 1.0, 0.1 #  標本 x の平均、係数、誤差の標準偏差
…
x1 = a_x * np.linspace(-size_of_sample / 2, size_of_sample / 2)
y1 = a_y * np.cos(2*np.pi*x1) # y 軸に用いる関数
data_x = mu_x + x1 + sigma_x * np.random.randn(size_of_sample) # x 軸
data_y = mu_y + y1 + sigma_y * np.random.randn(size_of_sample) # y 軸
```

右上の図を作成するために用いたサンプルの生成方法を見ます。scat_rand+corr6-2.py では、次のように np.exp(15*x1/size_of_sample)とする

ことで、y=exp(ax)に沿ったサンプルを生成しています。相関係数は約 0.59 ありますが、このような分布に対して線形回帰した y=ax+b の回帰直線を近似式としてしまうのは間違いで、y=exp(ax)を近似式とする必要があります。

```
mu_x, a_x, sigma_x = 50, 1.0, 0.5 #  標本 x の平均、係数、誤差の標準偏差
mu_y, a_y, sigma_y = 30, 1.0, 0.5 #  標本 x の平均、係数、誤差の標準偏差
…
x1 = a_x * np.linspace(-size_of_sample / 2, size_of_sample / 2)
y1 = a_y * np.exp(15*x1/size_of_sample) # y 軸に用いる関数
data_x = mu_x + x1 + sigma_x * np.random.randn(size_of_sample) # x 軸
data_y = mu_y + y1 + sigma_y * np.random.randn(size_of_sample) # y 軸
```

左下の図を作成する時に用いたサンプルの生成方法です。scat_rand+corr6-3.py では、x 軸、y 軸上の平均値が異なり、標準偏差が 1 のランダムな標本を 4 つ生成しています。まず x 軸、y 軸上の平均値と標本 x、y の係数、誤差の標準偏差を設定します。

```
mu_xs = 5, -5, 5, -5 # x 軸上の平均値
mu_ys = 5, 5, -5, -5 # y 軸上の平均値
sigma_x = 1.0 #  標本 x の誤差の標準偏差
sigma_y = 1.0 #  標本 y の誤差の標準偏差
```

これらを使い、(5, 5)、(-5, 5)、(5, -5)、(-5, -5)を中心とし x 軸、y 軸側ともに標準偏差 1 の正規分布に従うようなサンプルを生成します。まず、データを格納するための ndarray 配列を data_x=np.array(list())のように作成します。そのように作成した data_x、data_y の配列に、生成されたランダムなサンプル data_xi、data_yi を np.append(元のデータ, 追加するデータ)のように追加していきます。相関係数から判断すると無相関ですが、散布図を見ると 4 箇所に分布しているので、4 つの異なる標本が存在しているという意味が見出せます。

```
data_x=np.array(list()) # ndarray の初期化（data_x）
data_y=np.array(list()) # ndarray の初期化（data_y）
for mu_x, mu_y in zip(mu_xs, mu_ys):
    data_xi = mu_x + sigma_x * np.random.randn(size_of_sample)
    data_yi = mu_y + sigma_y * np.random.randn(size_of_sample)
    data_x = np.append(data_x, data_xi) # ndarray に追加（data_x）
    data_y = np.append(data_y, data_yi) # ndarray に追加（data_y）
```

右下の図を作成する時に用いたサンプルの生成方法です。scat_rand+corr6-4.py では、平均値(x, y) = (50, 30)の周辺に x 軸方向、y 軸方向ともに標準偏差 1 の正規分布に従うランダムなサンプルを生成します。そのようなサンプルなので無相関です。最後に data_x、data_y に(x, y) = (100, 100)の異常値を追加します。そうすることで相関係数が約 0.98 になりますが、1 つのサンプルで強い正の相関になっているので、意味のある関係があるとは言えません。

```
mu_x, sigma_x = 50, 1.0 #  標本 x の平均、誤差の標準偏差
mu_y, sigma_y = 30, 1.0 #  標本 x の平均、誤差の標準偏差
…
#  ランダムデータの準備
data_xi = mu_x + sigma_x * np.random.randn(size_of_sample)
data_yi = mu_y + sigma_y * np.random.randn(size_of_sample)
#  異常値の追加
data_x = np.append(data_xi, 100.0) #  標本 x の最後に 100 を追加
data_y = np.append(data_yi, 100.0) #  標本 y の最後に 100 を追加
```

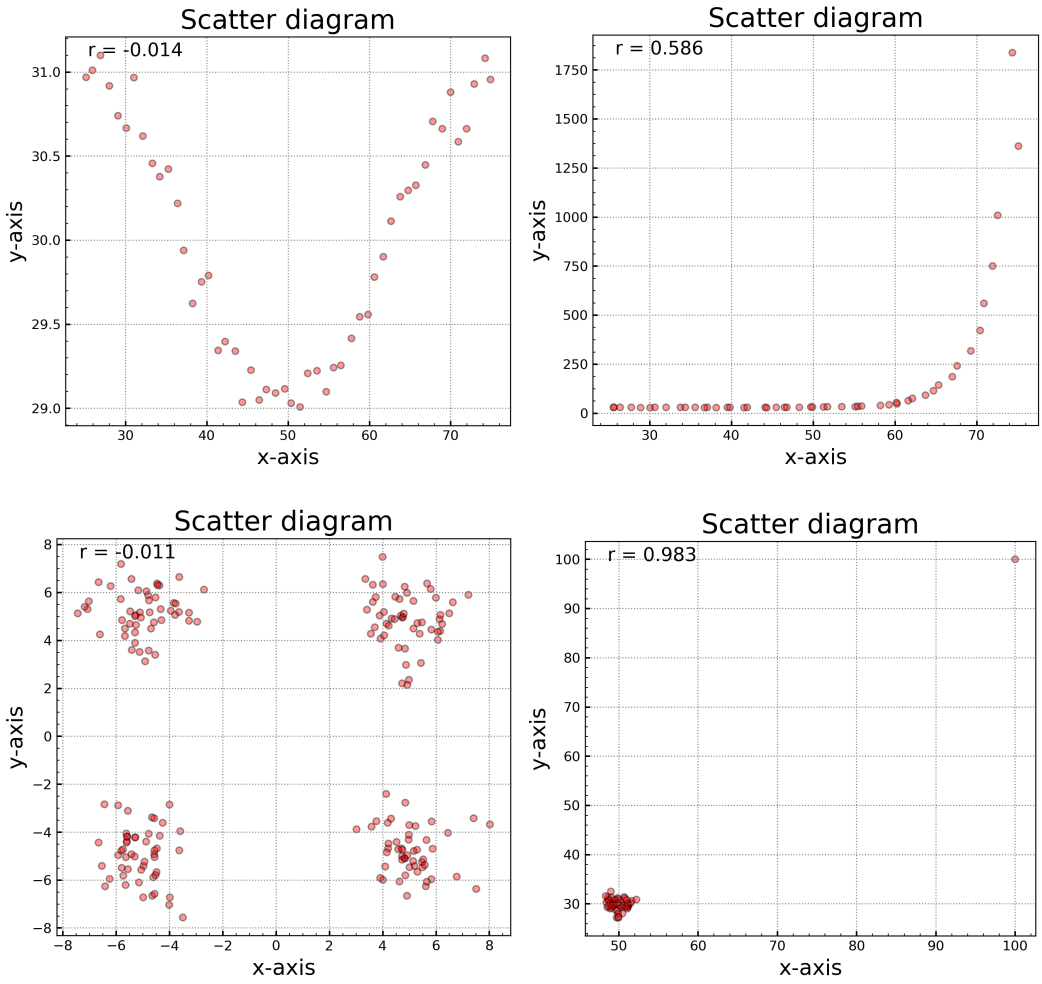


図４－５－14　様々な散布図。左側は相関係数がほぼ０のもので、右側は相関係数が大きいもの。

4.5.11 実際のデータを使ったプロット

それでは、実際のデータを使ったプロットを試してみます。北半球の気候変動のパターンとして北極振動（Arctic Oscillation：AO）と北大西洋振動（North Atlantic Oscillation：NAO）が知られており、両者の指数が似たような変動を示すとされています。実際にそのようになっているか、散布図を作って確かめてみたいと思います。index_scat.py は、NOAA/CPC のサイトから AO と NAO の index を取得して散布図を作図するためのプログラムです。データを web 上からダウンロードする際には、urllib.request.urlretrieve(URL, ファイル名)を使います。一度ダウンロードすれば、ao.txt、nao.txt に保存されて、保存したファイルを利用することができるので、retrieve = True とした時のみ urlretrieve が実行されるようにしています。ここでは AO index 取得部分を載せていますが、NAO index 取得も同様に行います。

```
import urllib.request #  最初に import
…
retrieve = True #  データの取得を行うかどうか
…
url="http://www.cpc.ncep.noaa.gov/products/precip/CWlink/daily_ao_index/monthly.ao.index.b50.current.ascii"
if retrieve:
    urllib.request.urlretrieve(url, "ao.txt") #  データをダウンロードして保存
```

次に取得したデータを読み込み、Pandas の DataFrame に格納します。保存された ao.txt を開いてみると、列方向は年、月、データの順でデータの区切りはスペース、行方向は時系列順にデータが格納されています。ヘッダはありません。Pandas に read_fwf という、列データがスペースで区切られたデータを読み込むツールがあり、得られた結果を DataFram として返却してくれるので、それを利用します。ちなみに Pandas には、csv データを読み込むツールもあり read_csv です（6.4.2 節で解説）。

read_fwf のオプションとして、ヘッダがないことを意味する header=None、時間軸のデータを認識させるオプション parse_date を与えています。このオプションには、時間軸として扱う列番号を与えることができ、parse_dates=[[0, 1]]

のように列番号をリストで与えると、1 列目と 2 列目を合わせて時間軸データに変換してくれます。もし月が先で年が後のデータであれば、parse_dates=[[1, 0]]です。DataFrame では、時刻などデータを参照する際の基準となる index を作ることができ、ここでは 1 列目を index にすることを意味する index_col=[0]を与えています。読み込まれたデータの 1 列目ではなく、parse_dates で変換された 1 列目の時刻データが index となります。読み込まれたデータの列に名前を付けるオプションが、names です。例えば ao.txt は 3 列の年、月、AO index データなので、それぞれに"year"、"mon"、"aoi"の名前を付けました。nao.txt の場合も同様に、"year"、"mon"、"naoi"の名前を付けています。このように列に名前を付けておくと、DataFrame が持っている.loc メソッドで切り出すことが可能になります。

```
dataset1 = pd.read_fwf("ao.txt", header=None, parse_dates=[[0, 1]], ¥
    index_col=[0], names=["year", "mon", "aoi"]) # AO index 読み込み
dataset2 = pd.read_fwf("nao.txt", header=None, parse_dates=[[0, 1]], ¥
    index_col=[0], names=["year", "mon", "naoi"]) # NAO index 読み込み
```

このように DataFrame に格納されたデータを、.loc メソッドで切り出して 1 次元の ndarray に変換します。dataset1.loc[**開始時刻**:**終了時刻**, **列の名前**]のように時間軸の範囲と列の名前で切り出しを行なっています。データは 1950 年からありますが、散布図は最近のデータから作成していて、開始年（syear）が 2010 年、終了年（eyear）が 2023 年です。開始時刻は str(syear)+"-01-01"なので、2010-01-01、終了時刻は str(eyear)+"-12-31"なので、2023-12-31 です。AO や NAO には冬季に卓越するような季節性があるので、本来ならば季節変化を考慮した解析をする必要がありますが、作図を簡単にするため、ここでは全部の月を使います。列方向には、先ほど設定した"aoi"や"naoi"の名前で切り出します。

```
syear = 2010
eyear = 2023
…
#  入力データの作成
data_x = dataset1.loc[str(syear)+"-01-01":str(eyear)+"-12-31","aoi"] # AO
data_y = dataset2.loc[str(syear)+"-01-01":str(eyear)+"-12-31","naoi"] # NAO
```

作成した入力データ data_x、data_y を plt.scatter に渡して、これまでと同様に散布図を作成します（図４－５－15）。マーカーは円形で色を赤、枠線の色を黒に設定しました。不透明度は設定していないので、デフォルトの alpha=1.0 になっています。

```
xlabel='AO'
ylabel='NAO'
title = "Scatter diagram"
…
plt.title(title, fontsize=20) #  タイトルを付ける
#  散布図
plt.scatter(data_x, data_y, color='r', marker='o', s=24, edgecolor='k')
plt.xlabel(xlabel, fontsize=16) # x 軸のラベル
plt.ylabel(ylabel, fontsize=16) # y 軸のラベル
```

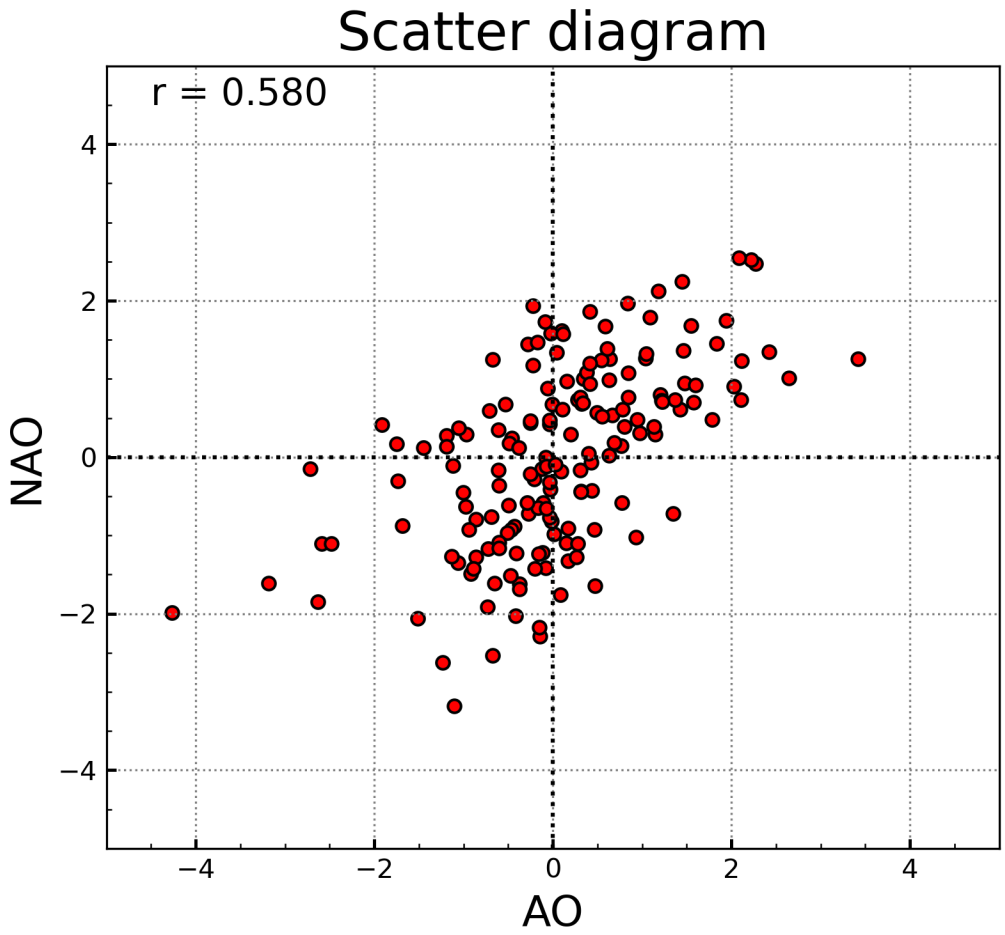


図４－５－15　x 軸に AO、y 軸に NAO の index をとった散布図

正の相関になっているので、回帰直線を引き回帰式も表示してみます（図４－５－16)。作図に用いたプログラムは、index_scat2.py です。これまでと同様に回帰係数と切片の計算を行います。

```
a = corr[1, 0] * data_y.std() / data_x.std() #  回帰係数（a）
b = data_y.mean() - a * data_x.mean() #  切片（b）
```

この値を使い y = ax + b（a、b には計算した値を入れる）のように表示するため、次のように文字列 name を設定しました。相関係数の表示の部分よりも複雑ですが、"データの表示順序と書式".format(データの中身)のようになっていて、s1、f1、s2、f2 の名前をつけ、s1、s2 は文字列、f1、f2 は.3f の書式で表し

た浮動小数点型にしています。さらに符号を表示するため、f2 は+.3f としています。s1="y = "、s2="x + "のように文字列を入力し、f1=a、f2=b のように回帰係数と切片の値を入力することで、y = 0.613x + -0.018 の文字列が name に入ります。

```
name = "{s1:s}{f1:.3f}{s2:s}{f2:+.3f}".format(s1="y = ", f1=a, s2="x", f2=b)
```

相関係数は、x 軸の下限から 5 %、y 軸の上限から 5 %の位置に表示していました。回帰式を表示する位置を相関係数の少し下にするため、x 軸の下限から 5 %、y 軸の上限から 10%に設定します。

```
xloc = xmin + 0.05 * (xmax - xmin) # x 軸上の座標
yloc = ymax - 0.10 * (ymax - ymin) # y 軸上の座標
#  回帰式の計算
x1 = np.linspace(np.floor(xmin), np.ceil(xmax), len(data_x))
y1 = a * x1 + b
plt.plot(x1, y1, color='k', lw=3, label='linear fit', zorder=2) #  回帰式のプロット
plt.text(xloc, yloc, name, fontsize=14, color='k') #  回帰式を表示
```

回帰式を作成する際の plt.plot に与えた zorder は zorder=2 です。散布図のマーカーを上にするため、plt.scat では zorder=3 にしています。2 と 3 を使ったのには理由があり、x=0、y=0 の線が一番下に来るようにしたかったためです。x=0、y=0 には点線が描かれているのであまり目立ちませんでしたが、図 4 – 5 –15 ではマーカーの上に点線が描かれています。ここで、axvline や axhline にも zorder を与えることで、図 4 – 5 –16 では x=0、y=0 の線の上にマーカーが描かれるようになりました。

```
plt.scatter(data_x, data_y, color='r', marker='o', s=24, edgecolor='k', ¥
                label='original', zorder=3) #  散布図
…
ax.axvline(x=0, color='k', ls=':', zorder=1) # x=0 の線を付ける
ax.axhline(y=0, color='k', ls=':', zorder=1) # y=0 の線を付ける
```

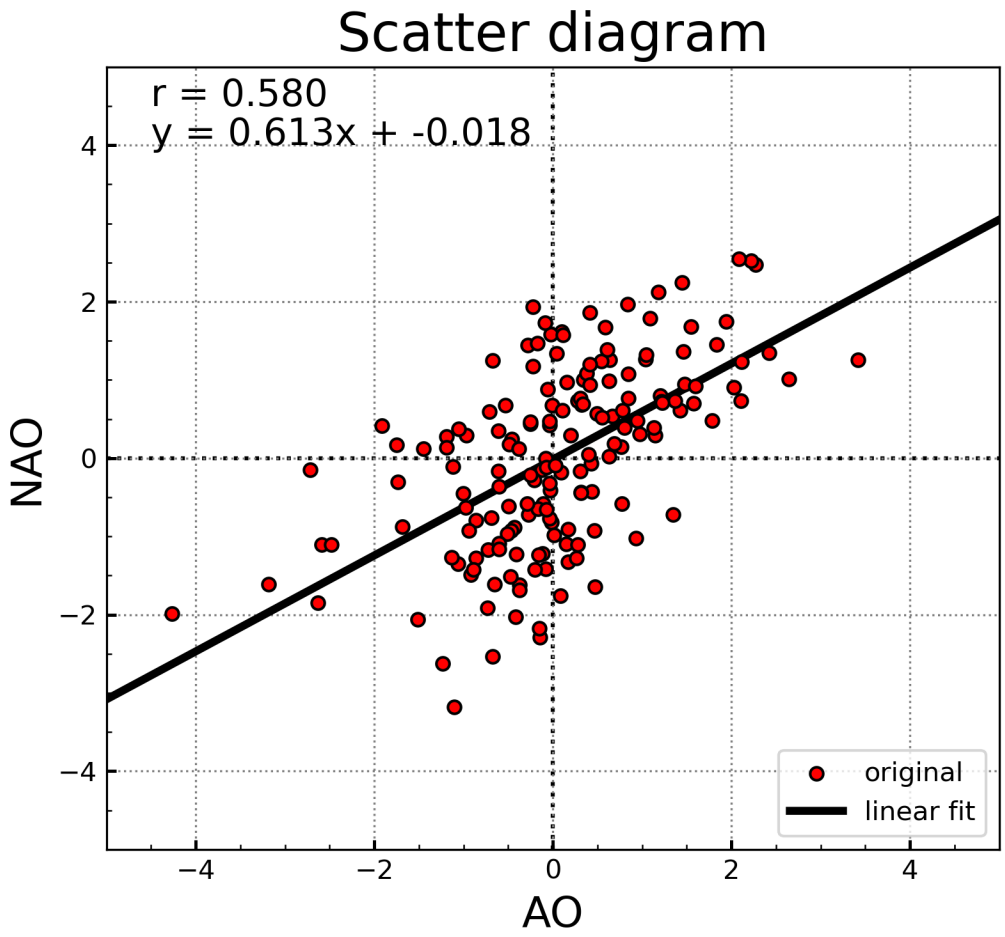


図４－５－16　回帰直線と回帰式を追加した

4.5.12 相関係数の範囲

計算した相関係数に信頼区間を付けたいこともあるかと思います。ここでは、そのような方法を紹介しておきます。index_scat3.py は、相関係数に 95%の信頼区間を表示するプログラムです。作図したものが図４－５－17 です。標本の大きさを n、相関係数を r、信頼率を A とします。95%信頼区間なら信頼率 A=0.95、信頼限界は(1-α)100%で、α=1-A=0.05 です。正規分布で上側確率が$\alpha/2$ に対するパーセント点を $Z_{\alpha/2}$ とし、正規分布表を参照すると、$Z_{\alpha/2}$=0.96 です。このとき、対応する 95%信頼区間の上限より上に 2.5%、下限より下に 2.5%のサンプルが分布しています。信頼区間の計算に用いるのがフィッシャーの Z 変換と逆変換です。標本相関係数に対するフィッシャーの Z 変換値を$\xi_r = f(r)$とすれば、Z 変換値は次式のようになります。

$$\xi_r = f(r) = \frac{1}{2} ln\left(\frac{1+r}{1-r}\right)$$

Z 変換した値に対して次のように上限、下限を求めます。

$$Z_u = \xi_r + \frac{Z_{\alpha/2}}{\sqrt{n-3}}$$

$$Z_l = \xi_r - \frac{Z_{\alpha/2}}{\sqrt{n-3}}$$

フィッシャーの Z 変換の逆変換を用いて上側信頼限界（$r_u = f^{-1}(Z_U)$）、下側信頼限界（$r_l = f^{-1}(Z_L)$）を求めます。

$$r = f^{-1}(Z) = \frac{\exp(2Z) - 1}{\exp(2Z) + 1}$$

Z=Z_u として導出した r_u が上側信頼限界、Z=Z_l として導出した r_l が下側信頼限界です。このように信頼区間を計算する部分を r_range という関数にしました。関数の引数として、標本の大きさ（n）、相関係数（r）、$\alpha/2$ に対するパーセント点（Zxx）を取り、関数の戻り値で上側信頼限界（ru）と下側信頼限界（rl）を返します。フィッシャーの Z 変換では、自然対数を返す np.log を使います。Z の上限、下限の計算では、平方根を返す np.sqrt、Z 変換の逆変換では e^x を返す np.exp を使います。

```
# n:標本の大きさ、r: 相関係数、Zxx:α/2 に対するパーセント点
def r_range(n, r, Zxx):
    Z = 0.5 * np.log((1+r)/(1-r)) #  フィッシャーの Z 変換
    ZU = Z + Zxx / np.sqrt(n - 3) #  上限
    ZL = Z - Zxx / np.sqrt(n - 3) #  下限
    #  フィッシャーの Z 変換の逆変換で信頼限界を求める
    ru = (np.exp(2*ZU) - 1) / (np.exp(2*ZU) + 1) #  上側
    rl = (np.exp(2*ZL) - 1) / (np.exp(2*ZL) + 1) #  下側
    return(ru, rl)
```

メインプログラム中では次のように呼び出しており、data_x の長さがサンプル数（相関係数の場合は data_x、data_y とも長さは同じなので、どちらを使っても良い)、corr[1, 0]が相関係数(スカラー値として渡すので、このような表記)、Z05=1.96 を渡しています。

```
ru, rl = r_range(len(data_x), corr[1, 0], 1.96) #  信頼区間の計算
```

計算された ru、rl と相関係数 corr[1，0]を使い、次のように表示する文字列を設定しています。先ほどの回帰直線の式を表示する場合よりも複雑になりましたが、文字列、数値、文字列、数値、、、の順に表示させたい順に並べているだけです。

```
name = "{s1:s}{f1:.3f}{s2:s}{f2:.3f}{s3:s}{f3:.3f}{s4:s}".format(s1="r = ", ¥
f1=corr[1,0], s2=" (", f2=rl, s3="-", f3=ru, s4=")")
```

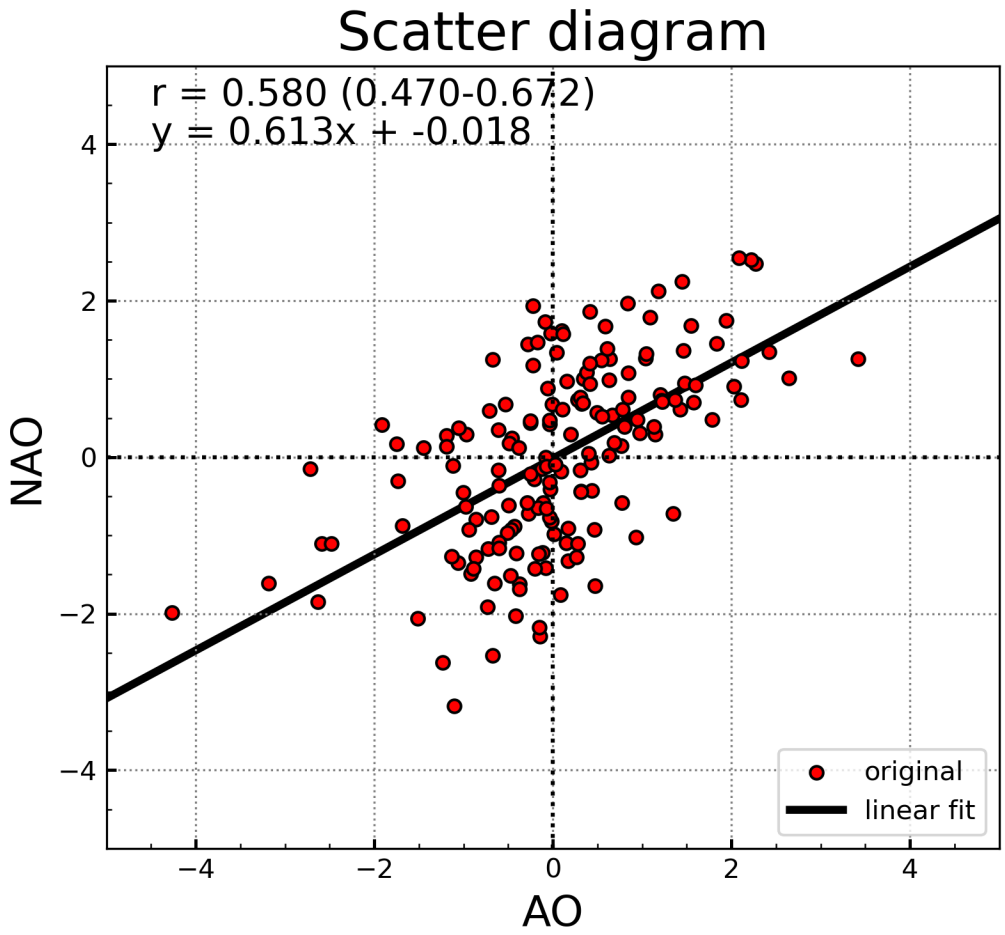


図４－５－17　相関係数に信頼区間を加えた

4.5.13 散布図のオプション

最後に散布図で使用可能なオプションの一覧を表４－５－１にまとめておきます。

表４－５－１　matplotlib の pyplot.scatter で使用可能なオプション一覧

オプション	説明
x, y（必須）	x軸, y軸に使用する配列
s	サイズ、デフォルト値：rcParams['lines.markersize']**2
c or color	マーカーの色、または、連続した色の値（cmapと共に使う）、デフォルト値：None
marker	マーカーの形、デフォルト値：‘o’（円形）
cmap	カラーマップ、c が float 型の場合のみ利用可、デフォルト値：False
norm	正規化を行う場合、c が float 型の場合のみ利用可、デフォルト値：None
vmin	正規化時の最小値、デフォルト値：None（データの最小値）
vmax	正規化時の最大値、デフォルト値：None（データの最大値）
linewidth or lw	線の太さ、デフォルト値：None
edgecolor	枠線の色、’none’で枠線無し、デフォルト値：'face'（cと同じ色に設定）
alpha	不透明度、デフォルト値：1.0
label	凡例を付ける場合、デフォルト値：None
zorder	プロットを重ねる順序、整数で指定

使用方法：plt.scatter(x, y)、plt.scatter(x, y, オプション)

4.6 複雑な図枠の配置

4.6.1 散布図をヒストグラムと一緒に描く

これまでは散布図とヒストグラムは単独で作成してきましたが、散布図の x 軸側と y 軸側に、それぞれのデータのヒストグラムを散布図よりも小さいサイズで描くことがあります。このように複雑なサブプロット配置を行うには、GridSpec を使います。図４－６－１は、GridSpec を使って３つに分割する場合を模式的に表したもので、散布図を配置する左上の枠を大きくするために縦軸を６：1、横軸を４：1 に分割しています。先ほどの散布図に GridSpec を使ってヒストグラムを加えたものが図４－６－２です。左上に先ほどの散布図が、残りの右上と左下にヒストグラムが配置されています。作図に用いたプログラムが index_scat+hist.py です。

GridSpec を使うためには、最初に gridspec の import を行います。

```
from matplotlib import gridspec
```

これまでは、fig. add_subplot でサブプロットを生成していましたが、代わりに gridspec.GridSpec を用います。引数は、1 番目が縦軸方向の配置数、2 番目が横軸方向の配置数です。配置するサブプロットが、それぞれの軸方向の長さに対してどのような比率で分割されるのかを指定するオプションが、縦軸方向の分割比率を表す height_ratios、横軸方向の分割比率を表す width_ratios です。それぞれ、height_ratios=**分割比率のタプル**、width_ratios=**分割比率のタプル**、の形式で与えます。タプルの要素数は、縦軸方向の配置数、横軸方向の配置数に合わせる必要があります。

gridspec.GridSpec から生成したインスタンス gs は、縦軸と横軸方向の図枠を gs[縦軸方向の番号，横軸方向の番号]のように持っています。図枠の順を図４－６－１のように左上、右上、左下のようにした場合、それぞれの図枠が、gs[0, 0]、gs[0, 1]、gs[1, 0]のように縦軸、横軸の番号をスライスで与えたものと対応します。

次にサブプロットを生成して、サブプロットのインスタンスを要素に取ったリスト ax として参照できるようにしておきます。生成された ax の要素 ax[0]、ax[1]、ax[2]が、それぞれ左上、右上、左下の図枠に対応します。そのため、作図の際には例えば ax[0]に対して操作を行います（ax[0]はインスタンスで、

ax[0].scatter のようなインスタントメソッドが使えます)。

```
gs = gridspec.GridSpec(2, 2, height_ratios=(4, 1), width_ratios=(6, 1))
ax = [plt.subplot(gs[0, 0]), plt.subplot(gs[0, 1]), plt.subplot(gs[1, 0])]
```

まず左上の枠（ax[0]）に ax[0].scatter で散布図を作成します。これまでと同様に x=0 の線、y=0 の線やグリッド線を ax[0].axvline、ax[0].axhline、ax[0].grid で付けていきます。また ax[0].plot で回帰式を描き、ax[0].text で相関係数や回帰式の表示を、ax[0].legend で凡例を付けます。

```
ax[0].scatter(data_x, data_y, color='r', marker='o', s=24, ¥
                edgecolor='k', label='original', zorder=2) #  散布図を描く
ax[0].axvline(x=0, color='k', ls=':') # x=0 の線を付ける
ax[0].axhline(y=0, color='k', ls=':') # y=0 の線を付ける
ax[0].grid(color='gray', ls=':') #  グリッド線を描く
ax[0].text(xloc, yloc, name, fontsize=14, color='k') #  相関係数を表示
ax[0].plot(x1, y1, color='k', lw=3, label='linear fit', zorder=1) #  回帰式を描く
ax[0].text(xloc, yloc, name, fontsize=14, color='k') #  回帰式を表示
ax[0].legend(loc='lower right') #  凡例を付ける
```

次に右上の枠（ax[1]）に左右を反転させ縦軸のメモリを右側に配置したヒストグラムを作成します。ヒストグラム作成は ax[1].hist で行いますが、これまでとは違いオプションで orientation="horizontal"を指定しています。これによって、ヒストグラムの棒を縦ではなく横に描くようになります。さらに ax[1].yaxis.tick_right で縦軸の目盛を右側に配置します。x 軸にデータの値、y 軸に頻度を取っているので、xaxis のように思えますが、表示する図のどの軸に相当するのかで決めているので、yaxis です。軸の範囲を設定する場合も同様で、set_ylim で x 軸データの範囲、set_xlim で y 軸の頻度範囲を指定します。

```
ax[1].hist(data_y, density=True, bins=edges, orientation="horizontal", ¥
        color='b', alpha=0.4, edgecolor='k') #ヒストグラムを描く
ax[1].yaxis.tick_right() #  縦軸の目盛りを右側に配置
ax[1].set_ylim([ymin, ymax]) # x 軸の範囲（軸が変わったので y 軸を変更）
ax[1].set_xlim([0, 0.6]) # y 軸の範囲（軸が変わったので x 軸を変更）
```

最後に左下の枠（ax[2]）にヒストグラムを作成します。こちらのヒストグラムはデフォルトの縦方向なので、x 軸、y 軸の範囲指定はそのままです。

```
ax[2].hist(data_x, density=True, bins=edges, ¥
        color='b', alpha=0.4, edgecolor='k') #  左下のヒストグラムを描く
ax[2].set_xlim([xmin, xmax]) # x 軸の範囲
ax[2].set_ylim([0, 0.6]) # y 軸の範囲
```

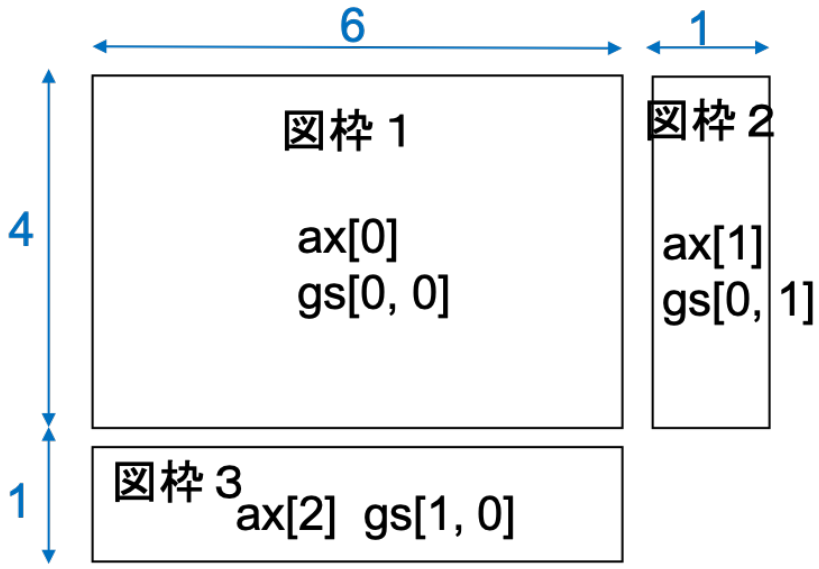


図 4 － 6 － 1　GridSpec により 3 つに分割する場合

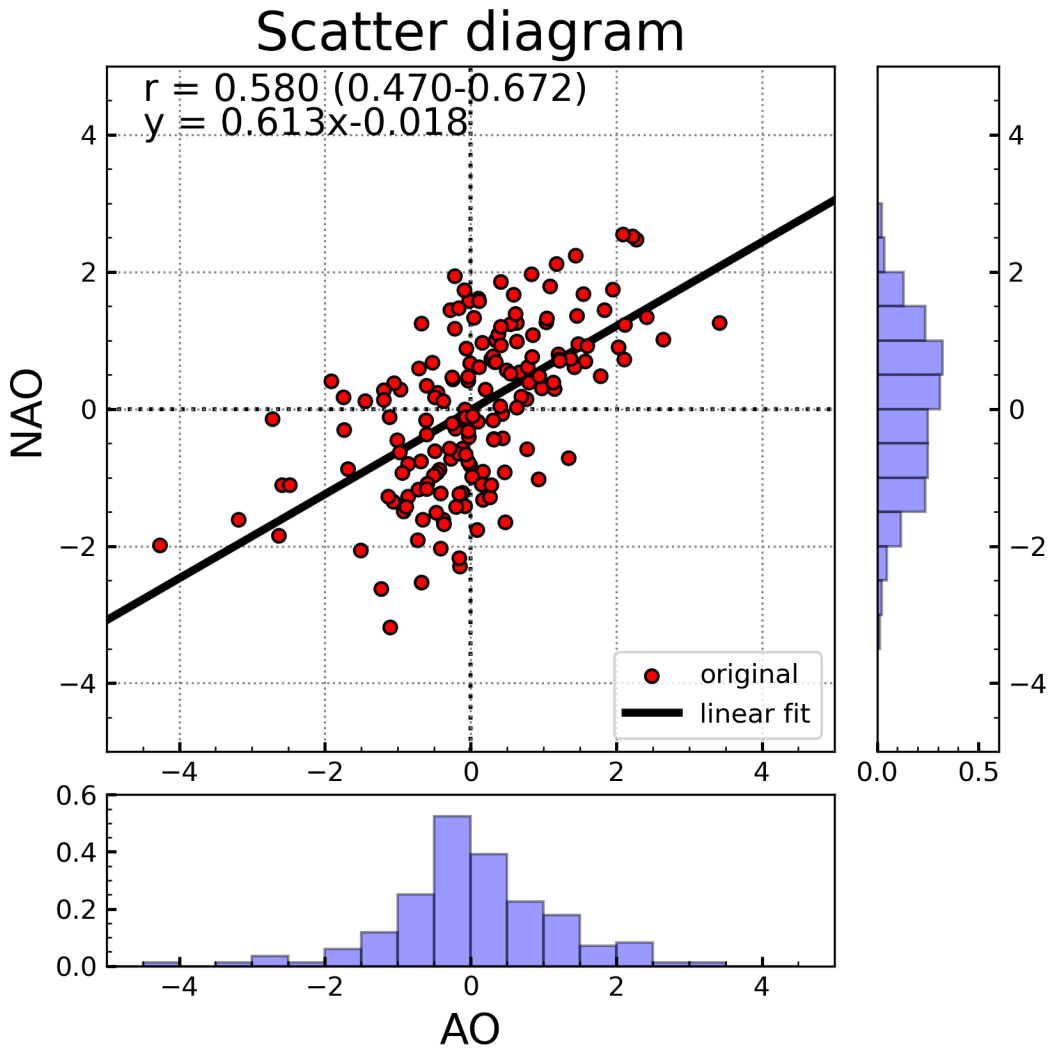


図４－６－２　x 軸データに AO index、y 軸データに NAO の index を取り作成した散布図の右側と下側にそれぞれのデータのヒストグラムを付けた

4.6.2 GridSpec を用いた様々な図枠の配置方法

GridSpec を使ってどのような配置まで可能なのか、様々な配置方法を試してみたいと思います。先ほどは height_ratios と width_ratios に与える分割比率のタプルは、height_ratios=(4, 1)のように 2 要素しか含めていませんでした。さらに要素を増やして複雑な配置にすることも可能です。gridspec_sample.py では、縦方向に 3 分割、横方向に 3 分割しています（図４－６－３）。縦横の分割数を表す 1 番目と 2 番目の引数を 3 にした上で、height_ratios=(2, 2, 1), width_ratios=(3, 3, 1)のようにしているので、縦軸を 2：2：1、横軸を 3：3：1 に分割することになります。

このように分割したので、最大 9 つまで図枠を追加できますが、gs[0, 0]、gs[0, 1]、gs[0, 2]、gs[1, 0]、gs[1, 1]、gs[2, 0]、gs[2, 2]の 7 つの図枠を追加しました。最初が縦軸方向の番号、2 番目が横軸方向の番号です。

```
gs = gridspec.GridSpec(3, 3, height_ratios=(2, 2, 1), width_ratios=(3, 3, 1))
ax = [plt.subplot(gs[0, 0]), plt.subplot(gs[0, 1]), plt.subplot(gs[0, 2]), ¥
    plt.subplot(gs[1, 0]), plt.subplot(gs[1, 1]), plt.subplot(gs[2, 0]), ¥
    plt.subplot(gs[2, 2]) ]
```

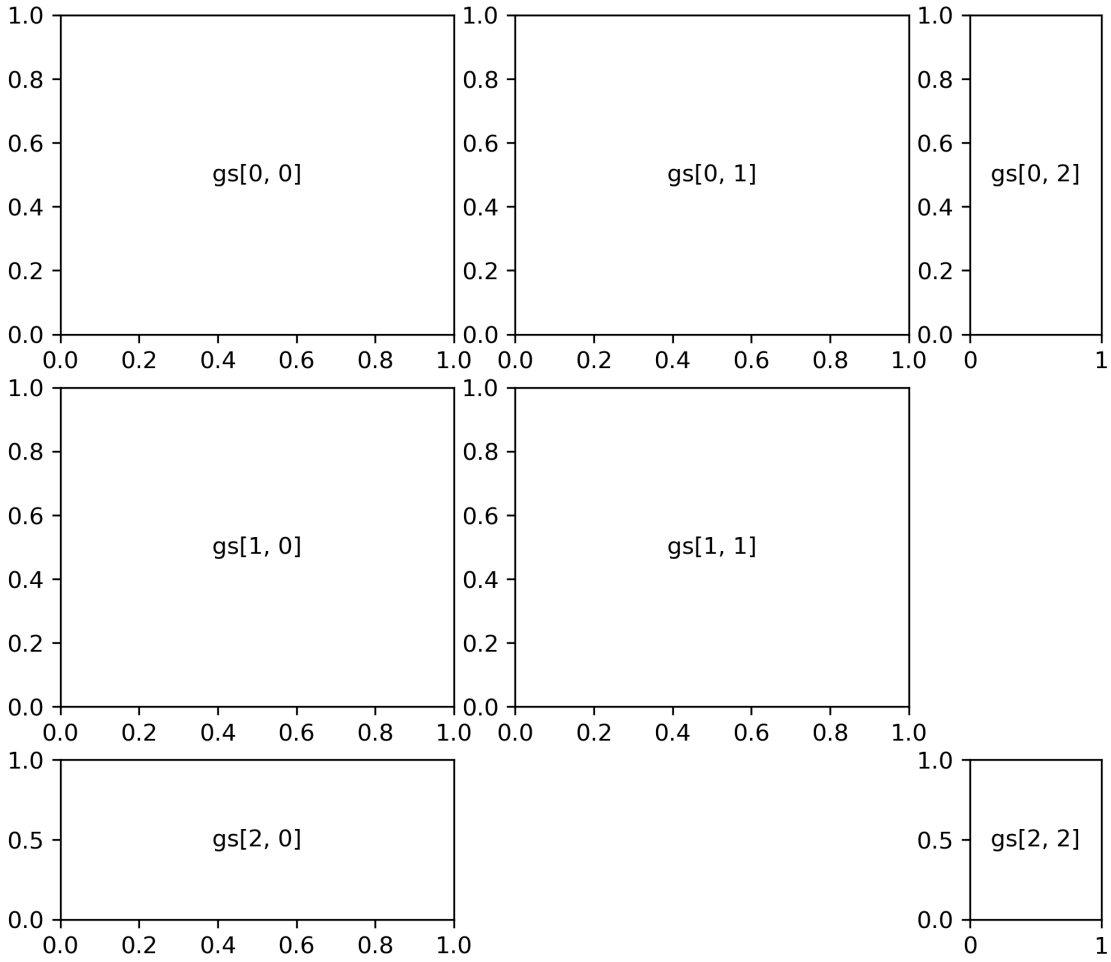


図４－６－３　縦横に３分割した場合

７ つの図枠の中心にテキストを配置します。ここでもサブプロット作成時のインスタンスをリスト ax の要素に入れているので、ax[0]～ax[6]の text メソッドを使います。サブプロットを作成した時点では、x 軸、y 軸とも 0～1 の範囲なので、1 番目と 2 番目の引数は 0.5 です。また水平方向の位置 ha='center'、鉛直方向の位置 va='center'を指定して (0.5, 0.5) が中心に来るようにします。

```
ax[0].text(0.5, 0.5, "gs[0, 0]", ha='center', va='center')
ax[1].text(0.5, 0.5, "gs[0, 1]", ha='center', va='center')
ax[2].text(0.5, 0.5, "gs[0, 2]", ha='center', va='center')
ax[3].text(0.5, 0.5, "gs[1, 0]", ha='center', va='center')
ax[4].text(0.5, 0.5, "gs[1, 1]", ha='center', va='center')
ax[5].text(0.5, 0.5, "gs[2, 0]", ha='center', va='center')
ax[6].text(0.5, 0.5, "gs[2, 2]", ha='center', va='center')
```

GridSpec で分割した 9 領域を複数合わせて 1 つの図枠とすることもできます（図４－６－４）。gridspec_sample2.py で作成しました。このように複雑な図枠配置を行う場合には、最初に完成後のイメージ図を作った方が良いでしょう。

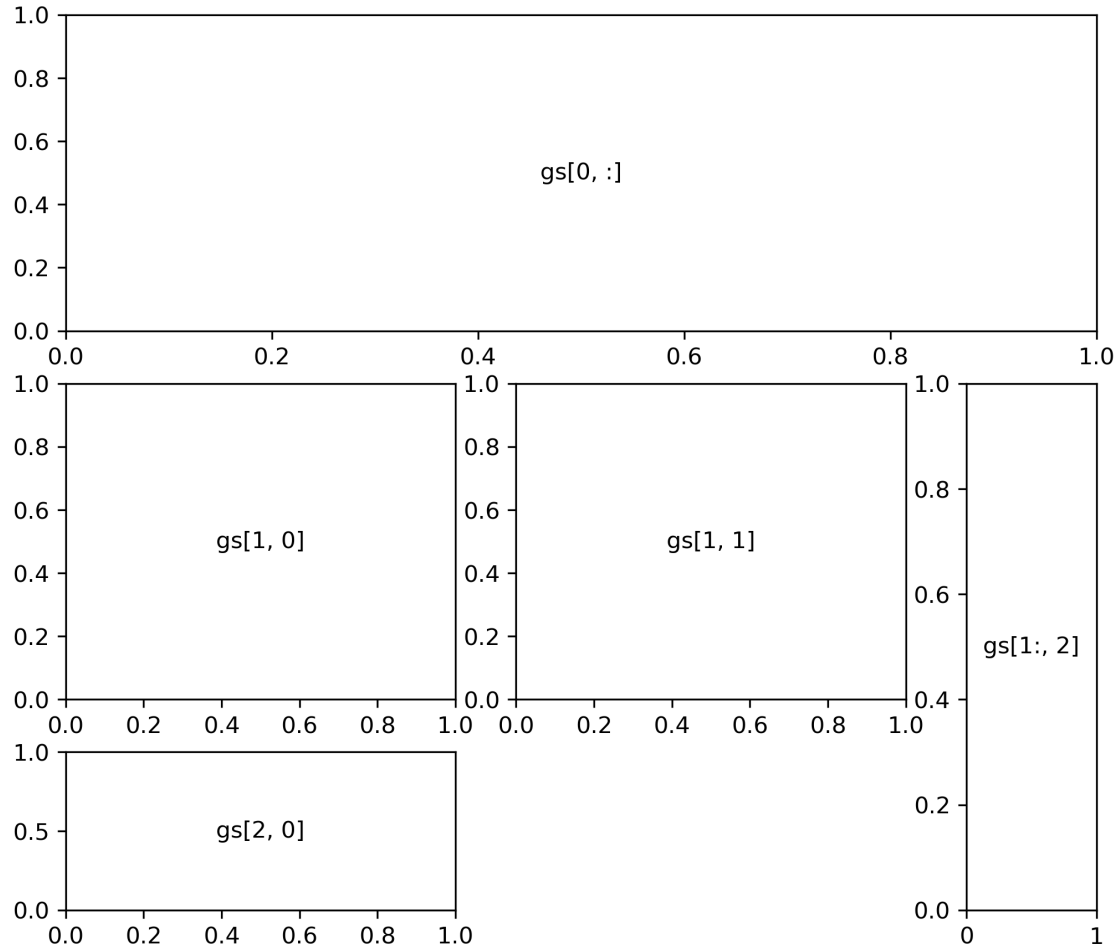


図４－６－４　複数の領域にまたがる図枠を作成

次のように gs[0, :]、gs[1, 0]、gs[1, 1]、gs[2, 0]、gs[1:, 2]の 5 つの図枠を追加しました。ここで gs[0, :]のコロンは、横軸方向全てを使うことを表すので、縦軸 0 番目（一番上）の領域の横軸方向にある 3 つの領域にまたがった図枠ができています。gs[1:, 2]の 1：は縦軸方向の 2 番目から最後の領域まで使うことを意味します。今は縦軸に 3 番目までしかないので、横軸 3 番目の領域の縦軸側 2〜3 番目にまたがった図枠ができます。

```
ax = [plt.subplot(gs[0, :]), ¥
    plt.subplot(gs[1, 0]), plt.subplot(gs[1, 1]), plt.subplot(gs[2, 0]), ¥
    plt.subplot(gs[1:, 2]) ] #  サブプロット作成
```

4.6.3 図枠の中に図枠を配置

サブプロットとは別に任意の場所に図枠を配置したい場合もあるかと思います。plt.axes を使うことで、独立した図枠を生成できます（図４－６－５）。gridspec_sample3.py で作成しました。

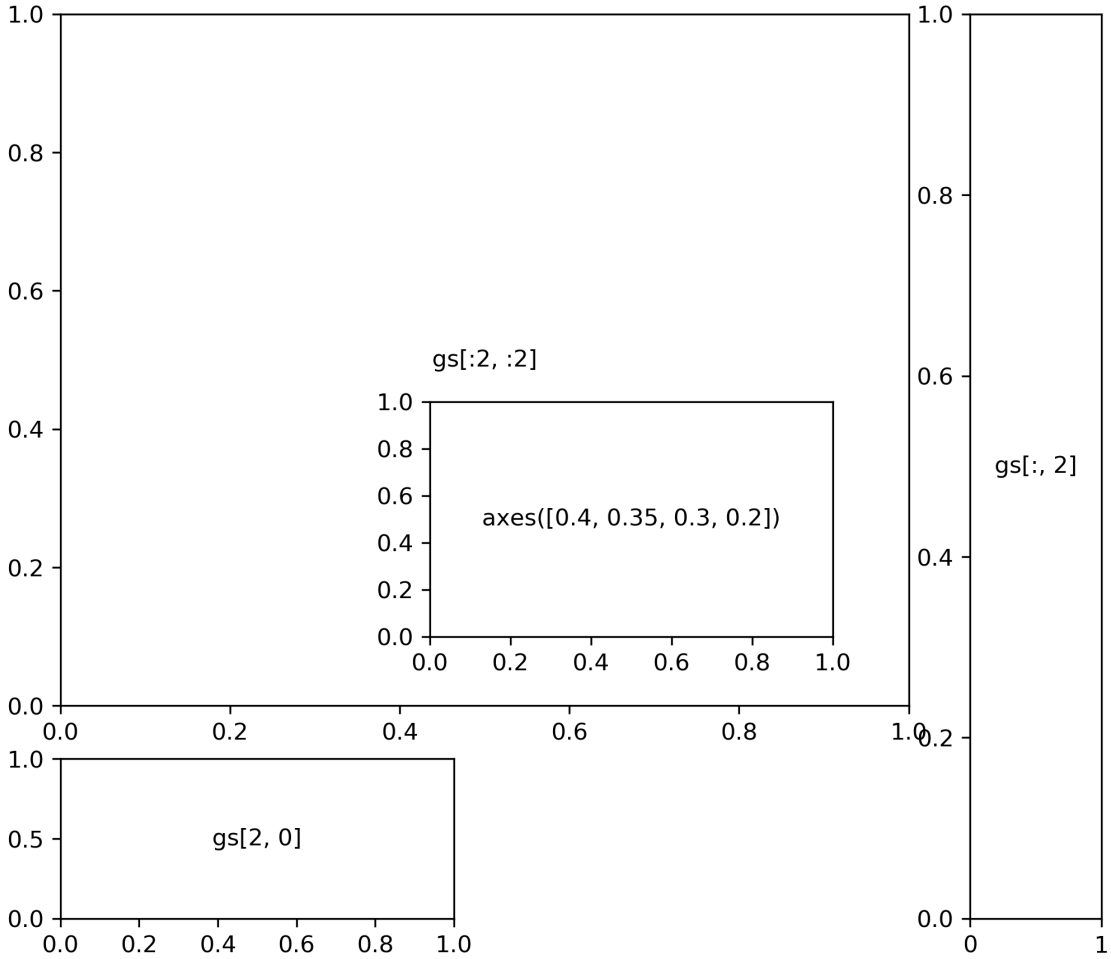


図４－６－５　図枠の中に図枠を配置

まず、plt.subplot で gs[:2, :2]、gs[2, 0]、gs[:, 2]の３つの図枠を生成します。gs[:2, :2]は縦軸方向に 0～1 番目、横軸方向に 0～1 番目を使うことを意味します。：2 なので 2 番目までと思ってしまいがちですが、python のスライスでは**開始点～終了点の 1 つ前**、までを切り出します。gs[:, 2]の方は、横軸の 3 番目の領域にある縦軸全てを使った図枠の生成です。

```
ax = [plt.subplot(gs[:2, :2]), ¥
    plt.subplot(gs[2, 0]), ¥
    plt.subplot(gs[:, 2]) ] #  サブプロット作成
```

生成した図枠のうち、左上の図枠の中に plt.axes で別の図枠を作成します。引数は 1 番目から順に、x 軸（横軸）方向の位置、y 軸（縦軸）方向の位置、x 軸方向の長さ（幅）、y 軸方向の長さ（高さ）です。

```
ax2 = plt.axes([0.4, 0.35, 0.3, 0.2]) #  図枠の生成
```

4.6.4 実際のデータを使った例

それでは、先ほどの AO と NAO の index を使って図枠の中に図枠を配置してみましょう。作図に用いたプログラムが index_plt.py です。このプログラムでは、2000〜2023 年までの折れ線グラフを大枠に作成し、2019〜2022 年を拡大したものを小枠に作成します（図４－６－６）。長期間のデータでは赤の AO index と青の NAO index の２つの折れ線グラフの違いが分かりにくいですが、拡大してみると、2019 年末から 2020 年初めには AO だけ大きな正になっていたことなど、細かい違いが読み取れるようになると思います。

ここでは 1 つの図枠の中に別の図枠を生成したいだけなので、大枠の方は、固定長のサブプロットを生成する fig.add_subplot で作成しています。

```
ax = fig.add_subplot(1,1,1) #  大枠の作成
```

大枠の中に、2000〜2023 年までの AO、NAO index の折れ線グラフを作成します。dataset1、dataset2 には x 軸データに用いる時間軸の情報と y 軸データの両方が含まれています。

```
ax.plot(dataset1, color='r', label='AO') # AO index
ax.plot(dataset2, color='b', label='NAO') # NAO index
```

なお現時点ではエラーにはならないのですが、将来的には Pandas でプロットを行う場合に register_matplotlib_converters()を行なっておく必要があるという Warning が出るので、回避のためにおまじないをしておきます。

```
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

大枠のデータ範囲は、set_xlim、set_ylim で設定しました。set_xlim では、日付を文字列で与えており、開始時刻は str(syear)+"-01-01"、終了時刻は str(eyear)+"-12-31"のように、dataset から時間の範囲で切り出す時と同じ形式が使えます。

```
syear = 2000 #  開始年（大枠）
eyear = 2023 #  終了年（大枠）
ymin = -8 # y 軸の下限の設定（大枠）
ymax = 5 # y 軸の上限の設定（大枠）
…
ax.set_xlim([str(syear)+"-01-01", str(eyear)+"-12-31"]) # x 軸の範囲
ax.set_ylim([ymin, ymax]) # y 軸の範囲
```

小枠の方は、plt.axis で独立した図枠を作成し ax2 として参照できるようにしています。与えているリストの要素は、x 軸上の座標、y 軸上の座標、横幅、縦幅です。

```
ax2 = plt.axes([0.20, 0.27, 0.3, 0.2]) #  小枠の作成
```

生成した図枠の中に AO、NAO index の折れ線グラフを作成します。

```
ax2.plot(dataset1, color='r', label='AO') # AO index
ax2.plot(dataset2, color='b', label='NAO') # NAO index
```

小枠の範囲を設定する際に、開始年（syear_s）を 2019、終了年（eyear_s）を 2022、y 軸の下限（ymin_s）を-2.5、y 軸の上限（ymax_s）を 2.5 とすることで、大枠に描いた折れ線グラフよりも拡大されたものとなります。

```
syear_s = 2019 #  開始年（小枠）
eyear_s = 2022 #  終了年（小枠）
ymin_s = -2.5 # y 軸の下限の設定（小枠）
ymax_s = 2.5 # y 軸の上限の設定（小枠）
…
ax2.set_xlim([str(syear_s)+"-01-01", str(eyear_s)+"-12-31"]) # x 軸の範囲
ax2.set_ylim([ymin_s, ymax_s]) # y 軸の範囲
```

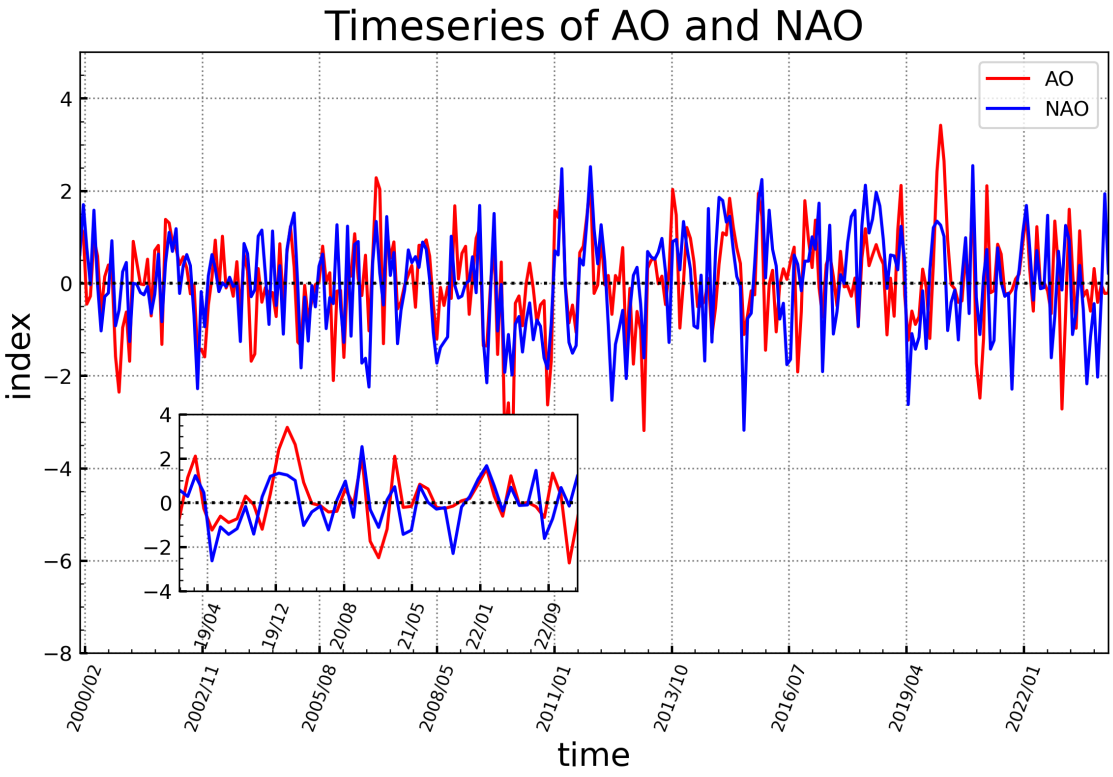


図４－６－６　AO index と NAO index の時系列図

x 軸の体裁を整える際に、いくつかのテクニックを使いました。まず、大枠の x 軸の時刻ラベルの向きを斜めにしています。それを行うために、
ax.set_xticklabels(ax.get_xticklabels(), rotation=70)
としています。set_xticklabels は x 軸ラベルのパラメターを設定するメソッドですが、すでに設定されているオプションを変更するので、ax.get_xticklabels() で現在の設定を取得した上で、rotation=70 のように変更したいパラメターを記述します。

なお ax.get_xticklabels の設定を変更すると FixedLocator と一緒に使用するようにとの Warning が出るため、先に x 軸の主目盛りを FixedLocator を使って設定しておきます。
ax.xaxis.set_major_locator(ticker.FixedLocator(ax.get_xticks().tolist()))

ax.xaxis.set_major_formatter は、x 軸の大目盛りラベルの書式を設定するためのものです。matplotlib には時刻を扱うための matplotlib.dates があり、それを mdates として参照しています。mdates.DateFormatter は時刻表記を変更

するためのもので、'%Y/%m'を引数として与えることで、「4 桁の年/月」の表記に変えられます。ax.xaxis.set_minor_formatter は、x 軸の小目盛りラベルの書式を設定するためのもので、ticker.NullFormatter()を引数に与えることで、小目盛りラベルを消すことができます。

```
import matplotlib.ticker as ticker
import matplotlib.dates as mdates
…
ax.xaxis.set_major_locator(ticker.AutoLocator()) # x 軸大目盛り
ax.xaxis.set_minor_locator(ticker.AutoMinorLocator()) # x 軸小目盛り
ax.xaxis.set_major_locator(ticker.FixedLocator(ax.get_xticks().tolist()))
ax.set_xticklabels(ax.get_xticklabels(), rotation=70, size="small")
ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y/%m')) #  時刻表記
ax.xaxis.set_minor_formatter(ticker.NullFormatter()) #  小目盛りラベルなし
```

小枠についても同様のテクニックを使いました。ただし、日付については、'%Y/%m'（4 桁の年）ではなく'%y/%m'（2 桁の年）にしています。

```
ax2.xaxis.set_major_locator(ticker.AutoLocator()) # x 軸大目盛り
ax2.xaxis.set_minor_locator(ticker.AutoMinorLocator()) # x 軸小目盛り
ax2.xaxis.set_major_locator(ticker.FixedLocator(ax2.get_xticks().tolist()))
ax2.set_xticklabels(ax2.get_xticklabels(), rotation=70, size="small")
ax2.xaxis.set_major_formatter(mdates.DateFormatter('%y/%m')) #  時刻表記
ax2.xaxis.set_minor_formatter(ticker.NullFormatter()) #  小目盛ラベルなし
```

時刻表記に用いられる書式指定子を一覧にまとめておきます（表４－６－１）。これらの書式指定子と文字列、空白を組み合わせて時刻表記のフォーマットを指定します。

表４－６－１　時刻表記に用いられる書式指定子一覧

書式指定子	説明	例
%Y	西暦を４桁	2019
%y	西暦の下２桁	19
%m	0 埋めした２桁の月	04
%d	0埋めした２桁の日	01
%H	0埋めした２桁の時（24h）	18
%I	0埋めした２桁の時（12h）	06
%M	0埋めした２桁の分	00
%S	0埋めした２桁の秒	00
%f	0埋めした６桁のマイクロ秒	000000
%B	ロケールの月名	April
%b	ロケールの月名の短縮形	Apr
%A	ロケールの曜日名	Monday
%a	ロケールの曜日名の短縮形	Mon
%j	年中の日数（1/1が001）	091