1 Python 入門

必要なライブラリを意識しよう

- Pandas(データ分析ライブラリ)
- NumPy/SciPy(数値解析ライブラリ)
- Matplotlib(グラフ描画ライブラリ)
- statsmodels(統計ライブラ)

である。これら4つのライブラリを自由に使いこなせるようになろう。

そうすれば自ずと道は開かれる！！

ラストリゾート：Python 3.12.1 ドキュメント

https://docs.python.org/ja/3/index.html (https://docs.python.org/ja/3/index.html)

ライブラリーリファレンス https://docs.python.org/ja/3/library/index.html (https://docs.python.org/ja/3/library/index.html)

言語リファレンス https://docs.python.org/ja/3/reference/index.html (https://docs.python.org/ja/3/reference/index.html)

　まずPythonで戸惑うのはべき乗演算と配列の作成ではないだろうか？金融関連のユーザーはこの2つを多用するので、この2つの方法が簡単に手に入らないと、それだけであきらめてしまう。多くのユーザーはPythonのべき乗演算が半角アステリスク(*)を2つ並べると知ったとたんに驚嘆して断念してしまう。ほとんどのプログラミング言語ではハットマーク(^)1つで済んでいた演算が2文字になってしまうのだから当然だ。

　配列も問題だ！WEB上で「Python、配列」と検索すると「データ構造」、「リスト、タプル、辞書」に関するページが多数ヒットされる。しかし、なかなかスプレッドシート(表)風の配列に行き当たらない。そして、「配列」という言葉を発見するとそれは「Nampy配列」であったりする。それでもリスト、タプル、辞書が配列に似ているという直観から、仕方なくこれらを使い始める。しかし、金融関連のユーザーが多用する2次元配列で壁にぶつかってしまう。これに随分と手間取った挙句にここで断念してしまうユーザーもいるのではないだろうか？

　しかし、Pythonを使う理由はこのデータの扱い方にあるといっても過言ではない。Pythonの演算スピードを実感してしまうと、もう手放せなくなってしまう。それにデータを管理・加工する際の使い勝手は最高だ。

　また、セミナーの参加者のなかにはプログラミングとは全く無縁な人もいるに違いない。やりたいことが明確に意識され、自分の業務が理解できている人であれば、特にプログラミングの知識がなくても、また特別な教育を受けていなくても、プログラムを英語の文章だと思って読んでくださいといっただけで、1年後には立派なプログラマーになっている。

では心の準備ができたので前進！

　Pythonのプログラムが、誰にとっても読みやすく、理解しやすいのは、プログラムのブロック構造が『字下げ』により決まるところが大きい。そのために行構造を理解しておくと便利である。

1. 行(物理行)は改行によって終わる。
2. 複数行(物理行)にわたってプログラムを書くときには行の終わりにバックスラッシュ文字" (\) "を置く。そうするとそれは論理行になる。
3. 論理行の先頭は、実行文のグループ化の判断に用いられる。『字下げ』の位置が重要な理由はここにある。
4. コメントが1行の場合には先頭をシャープ(#)で始める。

　非常に重要な点を4つのポイントとして表現したが、これでわかってしまう人は天才だ。まず、ポイント2についてポイント4を使って入力[1]で説明してみよう。

In [1]:
```
# aという変数に文字列を入力した。その文字列は文章になっていて、一行に
# ″たくさん書くと読みづらいので半角で72文字にしましょう″、という暗黙の
# 取り決めについて説明している。入力[1]のセルの最後にaと書いたのは、こ
# れでaの内容が出力[1]に表示されるからである。つまり、入力[1]のセルの
# 内容は改行によって読みやすくなっている。これはスクリーンのサイズによ
# らずに誰にでも読みやすいように、コードを書きましょうという配慮だ。
a='１行に書いてよい文字数は半角で７２文字まで、これは全角で３６文字¥
になる。これを超えるとバックスラッシュで行を変える。もちろん、¥
読みやすくなるのであれば、７２文字にこだわる必要はない。'
a
```

Out[1]: '１行に書いてよい文字数は半角で７２文字まで、これは全角で３６文字になる。これを超えるとバックスラッシュで行を変える。もちろん、読みやすくなるのであれば、７２文字にこだわる必要はない。'

1.1 データ(言語リファレンス)

- データについて

データ、要因、要素、データ点(データポイント)はあいまいな使い方がされていて、混乱を招くことが多い。データはデータ点(datapoint, datum)、つまり1つ1つの個々のデータのあつまりのことであり、要因はデータ点の特徴によりグループ化したものだ。要素はあいまいにいろいろな意味に使われるが、本書ではデータ点と同義語とする。

- データモデル

本セミナーの内容を理解するために難しいプログラミングの知識はいらない。逆に無駄な知識は内容を理解する障害になってしまうかもしれない。しかし、最低限の取り決めは、理解を加速する。データモデルは、データを処理する過程でオブジェクト(obj)とその属性、そしてメソッドがどのように機能し、相互作用するかを規定している。最初はオブジェクトといわれるとピンとこないかもしれないが、なれるとこちらの方が自然だ。

- オブジェクト

Pythonではすべてのデータがオブジェクトとなる。日経平均株価の過去の価格データを考えてみよう。まずPythonでは、この価格データはオブジェクトである。つぎに、この価格データには日付が付いているので、その日付もオブジェクトである。価格データが始値、高値、安値、終値で構成されていれば、それらもオブジェクトである。オブジェクトは同一性、型と値をもつ。同一性はそれぞれのオブジェクトのid(アドレス)だと考えればよい。いろいろなオブジェクトの種類があるので、それを型と呼んで区別する。たとえば、リストに1つの始値を登録すれば、始値は数値であるので、浮動小数点型、価格の値は始値そのものとなる。同一性を意識することはない。始値を別の数値に変更しても、型を整数型に変更しても、同一性は変更されない。

- リスト、タプルと辞書

リストはデータを格納するオブジェクトでその内容を変えることができる。データの各要素には番号が割り振られていて、それは順序付き整数となる。これはインデックスの役目を果たす。a=[c,d,e]というようにデータを角括弧でくくって作る。要素c,d,eには、0,1,2という数字がインデックスとして割り振られる。つぎは辞書だ。これはリストと似ているが、インデックスを自分で指定できる。かつ、文字列でも機能する。波括弧とコロンを用いる。a={"x":b,"y":c}の場合、"x"と"y"がインデックスだ。３つ目はタプルだ。これはリストに似ているが鍵括弧ではなく丸括弧を用いて作る。内容の変更はできないので、演算結果の出力としてよく用いられる。

- シーケンス

リストとタプルはシーケンスだ。インデックスが順序を持った有限な整数だからだ。ちなみに文字列型もシーケンス型である。辞書はインデックスが順序を持った整数ではないのでシーケンスではない。辞書はこの性質からマッピング型と呼ばれる。

- 呼び出し可能型

ユーザー定義関数、組み込み関数、組み込みメソッドは、関数呼び出しができる型である。

- モジュール

呼び出し可能型の集まり。

- ファイルオブジェクト

ファイルオブジェクトは開かれたファイルのこと。

データモデルの大まかな概念を表1.1にまとめた。

表1.1: データモデル

シーケンス型	有限な順序集合
	1.変更不能なシーケンス型：文字列、タプル型
	2.変更可能なシーケンス型：リスト型
数値型	整数型、長整数型、浮動小数点型、複素数型
マッピング型	辞書型
呼び出し可能型	関数の呼び出しが可能な型
	1. ユーザー定義関数
	2. ユーザー定義メソッド
	3. 組み込み関数
	4. 組み込みメソッド
モジュール	呼び出し可能型の集合
ファイルオブジェクト	開かれたファイル

ここでのデータモデルとは、Pythonが内部でデータを扱い処理するプロセスのことだ。データを構成するさまざまな要素を整理し、標準化し、データ同士の関連性や実世界で起きた事象の特性をも整理し標準化するものだ。

1.2 繰り返し処理言(語リファレンス)

繰り返し処理はコンピュータの得意とする処理である。人であれば繰り返し処理は簡単に間違えるが、コンピューターは忠実にこなす。しかし、for文の多用は禁物といわれる。PythonでのfOr文はin演算とペアで用いる。inのあとに来るのはコレクションだ。

for x in <コレクション>:
<繰り返し処理>

とすると、xにはコレクションのつぎの値が返される。

In [2]:
```
for x in ["for文は","イテラブルという","オブジェクトのコレクションに",¥
          "繰り返し処理を施す。"]:
    print(x, end="")#end=""により出力の改行はしない
```

for文はイテラブルというオブジェクトのコレクションに繰り返し処理を施す。

1.3 Python 標準ライブラリについて学ぼう

Python標準ライブラリには、プログラム言語の核となるデータ型と組み込み関数、変更可能シーケンス型のリストが含まれる。組み込み関数は表1.2に、組み込み型は表1.3に、シーケンスに共通な演算を表1.4にまとめた。詳しく知りたい人は以下を参照してほしい。

https://docs.python.org/ja/3/library/index.html#library-index
(https://docs.python.org/ja/3/library/index.html#library-index)

表1.2: 組み込み関数

1.abs(x) :	絶対値を返す。
2.enumerate(iterable):	カウントと値のタプルのリストを返す
3.float(x) :	浮動小数点に変換する。
4.int(x) :	整数に変換する。
5.max(iterable) :	最大値を返す。
6.min(iterable) :	最小値を返す。
7.print(object) :	出力する。
8.range(stop) :	1からstopまでの整数のリストを返す。
9.sorted(iterable) :	順番を並べ替える。
10.sum(iterable) :	合計を返す。
11.type(object):	オブジェクトの型を返す。
12.zip(∗iterable) :	タプルのリストを返す。
x:数または文字列、	object:オブジェクト、iterable:イテラブル

表1.3: 組み込み型

組み込み型	
数値型	動的型付言語：実行したときに変数の型が決定
シーケンス型	n個の要素をもつシーケンスから成りインデックスをもつ要素の集合
	変更不能シーケンス型:
	ー　文字列: 単引用符、2重引用符によってくくる。
	ー　タプル：複数の要素をカンマで区切り丸括弧( ) でくくる。
	変更可能シーケンス型:
	リスト：リストは幾つかの変数をグループとして扱うデータ構造
マッピング型	
	辞書：インデックス化されたオブジェクトの集合

表1.4:シーケンスに共通の演算

	s,tは同じ型のリスト,タプル,rangeのシーケンス
s+t	文字列の連結は「+」
s[i]	s[i] sのゼロから数えてi番目の要素を選択
スライス:	条件を満たすインデックスをもつすべての要素を抽出
	s[i:j]: sのiからj番目までの要素を選択
	s[i:j:k]: sのiからj番目までの要素をk毎にスライス
	s[:]: 全てのデータを抽出
	s[::-1]:全てのデータを逆順にして抽出
x in s	sのある要素がxと等しければTrue
len(s)	sの長さ

表1.4:シーケンスに共通の演算

min(s)	sの最小値
max(s)	sの最大値

In [3]:
```
# テキストシーケンス型に'vw'が含まれればTrue
'vw' in "vwxyz"
```

Out[3]: True

In [4]:
```
# 文字xがリストの要素に含まれればTrueを返す。
'x' in ["x","y","z"]
```

Out[4]: True

In [5]:
```
#end=""により出力が改行されない。
for i in [1,2,3]:
    print(i,end='')
```

```
123
```

In [6]:
```
# ２つのリストを同時に処理
for i,j in zip(["x","y","z"], [1,2,3]):
    print(i,j)
```

```
x 1
y 2
z 3
```

1.4 リストについて学ぶ

リスト型はPythonの中核をなし、有用なメソッドをもつ。リストは変更可能なシーケンス型だ。角括弧([ ]) のなかに要素、数式をカンマで区切って並べて作る。関数len()を用いて、要素数を得ることができる。シーケンスの長さがnの場合、そのインデックスは0,...,n-1となる。シーケンスsのインデックスiの要素はs[i]で得ることができる。スライス操作をサポートしているので、インデックスkのすべての要素をs[i:j]で選択することができる。この場合i≤k<jである。

シーケンスsの要素1,2,3,4,5を表示するには

In [7]:
```
s=[1,2,3,4,5]
s
```

Out[7]: [1, 2, 3, 4, 5]

とすればよい。最後の行に変数sを置くと、printを必要とせずに表示できる。これはJupyter notebook（Ipython notebook）の機能の1つだ。きれいで見やすい表示をしてくれる。

s[i]のiはインデックスで、指定した要素はタプルで返される。

In [8]:
```
s[0],s[1],s[2],s[3],s[4]
```

Out[8]: (1, 2, 3, 4, 5)

スライス操作でも要素を得ることができる。リストで返される。

In [9]:
```
s[0:5]
```

Out[9]: [1, 2, 3, 4, 5]

s[-5:]のようにスライス操作で後ろから指定することもできる。s[:]はすべてのデータを表示し、s[::-1]はデータを反転している。

In [10]:
```
print('後ろから指定 ',s[-5:])
print('全ての要素 ',s[:])
print('データの反転 ',s[::-1])
```

```
後ろから指定  [1, 2, 3, 4, 5]
全ての要素  [1, 2, 3, 4, 5]
データの反転  [5, 4, 3, 2, 1]
```

リストは連結も可能。

In [11]:
```
s=s+[10,11]
s
```

Out[11]: [1, 2, 3, 4, 5, 10, 11]

リストでは指定したインデックスの要素を入れ替えられる。

In [12]:
```
s[2]=1000
s
```

Out[12]: [1, 2, 1000, 4, 5, 10, 11]

リストはさらに幾つかのメソッドを持っている。xは要素、項目等であり、値、文字、文字列、リスト(タプル、辞書)でもよい。

1. append(x): xをリストの最後に加える。
2. count(x) : リストに含まれるxの数を数える。
3. insert(i,x) :i番目のポジションにxを挿入する。
4. remove(x) :xである最初の値を削除する。

In [13]:
```
s.append(2000)
s
```

Out[13]: [1, 2, 1000, 4, 5, 10, 11, 2000]

In [14]:
```
s.count(1)
```

Out[14]: 1

In [15]:
```
s.insert(2,3)
s
```

Out[15]: [1, 2, 3, 1000, 4, 5, 10, 11, 2000]

In [16]:
```
s.remove(2000)
print(s)
s.remove(1000)
s.remove(10)
s.remove(11)
s
```

```
[1, 2, 3, 1000, 4, 5, 10, 11]
```

Out[16]: [1, 2, 3, 4, 5]

for文とin演算子とprint関数を使って要素を表示することもできる。enumerateを使うとインデックスを作る手間が省ける。

In [17]:
```
#インデックスと要素をペアで表示
for i in s:
    print(i-1,end=" ")
    print(s[i-1],end=",")
print()
for i in s:             #best
    print(i-1,end=" ")
    print(i,end=",")
print()
for i in range(len(s)):
    print(i,s[i],end=",")
print()
for i,j in enumerate(s):#best
    print(i,j,end=",")
```

```
0 1,1 2,2 3,3 4,4 5,
0 1,1 2,2 3,3 4,4 5,
0 1,1 2,2 3,3 4,4 5,
0 1,1 2,2 3,3 4,4 5,
```

確かにfor文の多用はPythonの処理時間を遅らせるが、使い方によっては早まる。そのためにはfor文の中の処理の深い理解が必要だが、ここではそれはせずに、簡単に説明する。

for <変数> in <イテラブル>:
<ステートメント>

> <イテラブル>はコレクションで要素を一度に1つずつ返せる反復可能オブジェクトである。例えばリストとタプルとrangeオブジェクトだ。<ステートメント>は繰り返し処理で抽出された<イテラブル>の要素<変数>に順番に施される。

内包表記はリストを生成する方法の1つ。高速、幅広い用途で簡単で力強いPythonの記法として知られるPythonicの１つ。

In [18]:
```
b=[]
for x in range(5):
    b.append(x**2)
c=[x**2 for x in range(5)]#内包表記
b,c
```

Out[18]:
```
([0, 1, 4, 9, 16], [0, 1, 4, 9, 16])
```

In [19]:
```
xy=[]
for x in [1,2,3]:
    for y in [3,1,4]:
        xy.append((x, y))
xxyy=[(x, y) for x in [1,2,3] for y in [3,1,4]]#内包表記
xy,xxyy
```

Out[19]:
```
([(1, 3), (1, 1), (1, 4), (2, 3), (2, 1), (2, 4), (3, 3), (3, 1), (3, 4)],
 [(1, 3), (1, 1), (1, 4), (2, 3), (2, 1), (2, 4), (3, 3), (3, 1), (3, 4)])
```

より複雑な内包表記も可能、しかし複雑すぎると読みづらくなり逆効果となる場合もある。

In [20]:
```
c=[x**2 for x in range(5)if x<3]#内包表記
c
```

Out[20]:
```
[0, 1, 4]
```

1.5 ユーザー定義関数の基本

ユーザー定義関数は、プログラマが特定のタスクを実行するために自ら定義する関数です。これらの関数は、コードの再利用性を高め、プログラムの構造を整理し、可読性を向上させるために重要です。

1. **定義方法**: ユーザー定義関数は `def` キーワードを使って定義されます。関数の定義は以下の形式をとります：

   ```
   def function_name(parameters):
       # 関数の処理
       return result
   ```

   - `function_name` は関数の名前です。
   - `parameters` は関数に渡す引数です（なくてもよい）。
   - `return` ステートメントは関数の出力を返します。
2. **引数**: 引数は、関数に渡される値やデータです。引数は必須でなくてもよく、関数によっては複数の引数を取ることもあります。
3. **戻り値**: `return` ステートメントを使用して、関数の結果を呼び出し元に返すことができます。戻り値は任意であり、返さない関数も作成できます。

In [21]:
```
def add(a, b):
    return a + b

result = add(5, 3)
print(result)  # 出力: 8
```

```
8
```

1.6 NumPy配列

ndarraysの生成

In [22]:
```
import numpy as np
a=np.array([1.23456,1000,2])
a
```

Out[22]:
```
array([    1.23456, 1000.     ,     2.     ])
```

In [23]:
```
b=np.array([1.23,100000000,2])
b
```

Out[23]:
```
array([1.23e+00, 1.00e+08, 2.00e+00])
```

NumPy配列のスライス操作

In [24]:
```
b[0],b[1],b[2],b[:]
```

Out[24]:
```
(1.23, 100000000.0, 2.0, array([1.23e+00, 1.00e+08, 2.00e+00]))
```

1.7 NumPy zeros

指定された形状とデータ型に合わせて、すべての要素が0である配列を生成します。これは、特定のサイズの配列を初期化する際に非常に便利です。

```
numpy.zeros(shape, dtype=float, order='C')
```

- shape ：生成する配列の形状。整数または整数のタプルで指定します。例えば、 5 は長さ5の1次元配列を、 (5, 2) は5行2列の2次元配列を意味します。
- dtype ：配列のデータ型。デフォルトは float ですが、 int や complex など他のデータ型を指定することもできます。
- order ：配列のメモリ上での格納順序。 'C' はCスタイル（行優先）、 'F' はFortranスタイル（列優先）です。

In [25]:
```
import numpy as np
# 長さ5の1次元配列を生成
arr1d = np.zeros(5)
print(arr1d)
```

```
[0. 0. 0. 0. 0.]
```

In [26]:
```
# 3行4列の2次元配列を生成
arr2d = np.zeros((3, 4))
print(arr2d)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

1.8 NumPy linspace

指定された区間を一定の間隔で分割する数値の配列を生成するために使用されます。

```
numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis
=0)
```

- start ：配列の開始値。
- stop ：配列の終了値。
- num ：生成する等間隔のサンプル数。デフォルトは50です。
- endpoint : True の場合、終了値を配列に含めます。 False の場合は含めません。
- retstep : True の場合、サンプル間のステップサイズも出力します。
- dtype ：配列のデータ型。指定されない場合は、入力データに基づいて推測されます。
- axis ：戻り値の配列における間隔を設定する軸。

In [27]:
```
import numpy as np

# 0から10までを等間隔で5つの数値で分割
arr = np.linspace(0, 10, num=5)
print(arr)
```

```
[ 0.   2.5  5.   7.5 10. ]
```

1.9 NumPyによる疑似乱数生成

理想的な乱数列は熱雑音とか原子核分裂などの物理乱数である。本質的にランダムに生成された乱数列には周期や再現性がないなどのメリットがある。コンピューターにより生成する疑似乱数列では、長い周期と高次元での緻密性から長い間メルセンヌツイスターが用いられてきた。しかし、最近では統計的評価、計算速度、メモリー占有率、予測不可能性、再現性に優れたPCG系列の乱数が注目されている。

NumPy generatorの乱数ルーチンは、シーケンスを作成するBitGeneratorと、それらのシーケンスを使用して異なる統計分布からサンプルを取得するGeneratorの組み合わせで擬似乱数を生成している。そのパフォーマンスは目を見張るものがある。本書ではPCG系を用いる。

出所：https://www.pcg-random.org/index.html (https://www.pcg-random.org/index.html)

In [28]:
```
from numpy.random import default_rng, Generator, PCG64, MT19937
rng = default_rng()
```

1.10 pandas

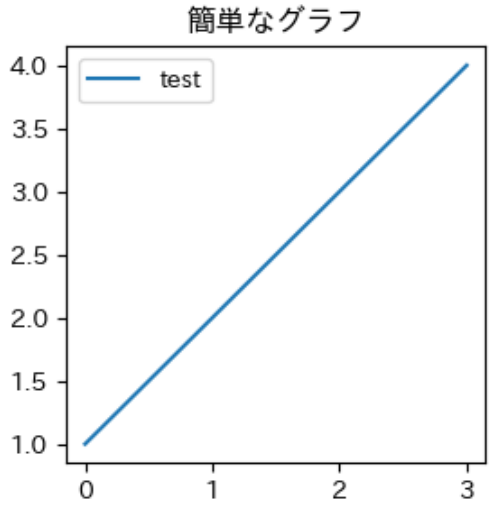
これは、関連するデータ、ラベル付けされたデータの作業を簡単で直感的に行えるようにするパッケージだ。かつ、高速かつ柔軟性のあるデータ構造を提供する。pandasの存在がPythonを使う1つの理由でもある。詳しくはあとで扱う。

pandasのインポートとフォーマットの指定方法

In [29]:
```
import pandas as pd
```

In [30]:
```
# 日本語フォントの初期設定
import matplotlib.pyplot as plt
import japanize_matplotlib

# DataFrameの作成
df=pd.DataFrame([1,2,3,4],columns=['test'])
# plotを用いた簡単なグラフ
df.plot(figsize=(3,3))# データのプロットとグラフの大きさの指定
# 日本語表記
plt.title('簡単なグラフ')
#plt.savefig('簡単なグラフ.jpeg',dpi=600)
plt.show()
```



1.11 Statsmodels

統計モデル、時系列モデルの分析に用いるクラスと関数を提供する。本書では線形回帰、単位根検定、コレログラムなどの機能をつかう。

In [31]:
```
import statsmodels.api as sm    #接頭語をsmとしてstatsmodels.apiをインポート
x=np.arange(20)
x0=sm.add_constant(x)
y=np.array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20])
res=sm.OLS(y.T,x0).fit()
x0[:4],res.params[0],res.params[1]
```

Out[31]:
```
(array([[1., 0.],
        [1., 1.],
        [1., 2.],
        [1., 3.]]),
 0.9999999999999947,
 1.0000000000000007)
```

In [32]:
```
res.summary()
```

Out[32]:

OLS Regression Results

Dep. Variable:	y	**R-squared:**	1.000
Model:	OLS	**Adj. R-squared:**	1.000
Method:	Least Squares	**F-statistic:**	3.149e+31
Date:	Tue, 16 Jan 2024	**Prob (F-statistic):**	1.21e-273
Time:	15:11:54	**Log-Likelihood:**	632.95
No. Observations:	20	**AIC:**	-1262.
Df Residuals:	18	**BIC:**	-1260.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	1.0000	1.98e-15	5.05e+14	0.000	1.000	1.000
x1	1.0000	1.78e-16	5.61e+15	0.000	1.000	1.000

Omnibus:	3.724	**Durbin-Watson:**	0.061
Prob(Omnibus):	0.155	**Jarque-Bera (JB):**	1.419
Skew:	-0.163	**Prob(JB):**	0.492
Kurtosis:	1.737	**Cond. No.**	21.5

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

1.12 基本的なグラフ表示の設定

matplotlibは本書で扱うグラフを用いた分析で多用される。補足的にseabornが用いられる。

1 プロット、散布図、頻度図(ヒストグラム)

- plot(*args, *kwargs)

Python 1

plot	**(∗args, ∗∗kwargs)**
∗args:	シーケンス型の引数
∗∗kwargs:	マッピング型の引数)
plot(x,y):	yとxを線またはマーカーでプロットする。
plot(y):	xを1,...nのインデックスとしてyを線またはマーカーでプロットする。
alpha:	色の濃さ 0から1
color or c:	{'b'(青),'g'(緑),'r'(赤),'c'(シアン),'m'(マゼンダ),'y'(黄),'k'(黒)}
figure:	図の大きさの指定
label：	要素の名前
linestyle or ls	{'-', '--', '-.', ':', '', ...}
linewidth or lw：	線の幅

plot	(∗args, ∗∗kwargs)
marker:	{"."(点),"*"(星),"o"(丸),"+"(+),"x" (x),"s"(四角形)}

- scatter(x, y, **kwargs)

Python 2

scatter	(x, y)
scatter(x,y):	x とyの散布図を作成する。
x,y	配列など

plot参照

- hist(x, bins=None, density=False, cumulative=False, **kwargs)

Python 3

hist	(x, bins=None, density=False, cumulative=False, ∗∗kwargs)
hist(x):	xの頻度図を作成する。
x:	配列またはシーケンス
bins:	容器の数;整数、シーケンス、ディフォルト＝10
density:	確率密度関数
cumulative:	累積分布関数

2 グラフ軸回り、他の設定

Python 4	軸回り、他の設定
figure(figsize):	新しいfigureを設定;figsizeはfigureの大きさをインチで指定(幅、高さ)
add_subplot:	サブプロット配置の一部として、図にAxesを追加する。
ylabel:	y軸にラベルを設定
xlabel:	x軸にラベルを設定
title:	グラフのタイトルを設定
legend(loc,bbox_to_anchor):	凡例の位置の設定； loc:凡例の位置; bbox_to_anchor:凡例の箱の位置

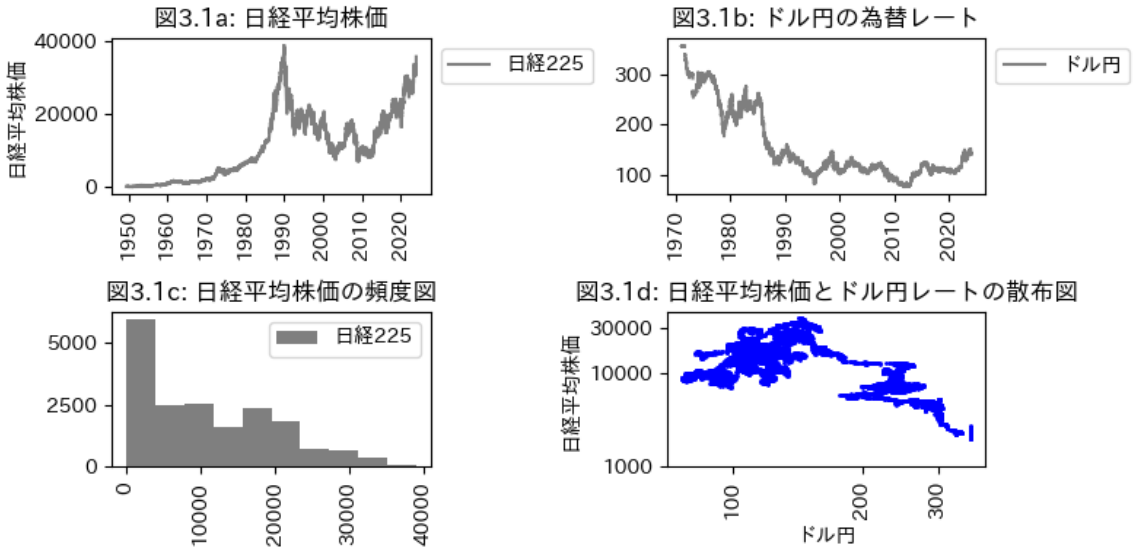
In [33]:
```
import pandas_datareader.data as web
start="1949/5/16"

n225 = web.DataReader("NIKKEI225", 'fred',start).NIKKEI225
fx=web.DataReader("DEXJPUS", 'fred',start).DEXJPUS
port=np.log(pd.concat([n225,fx],axis=1)).dropna()
port[:1]
```

Out[33]:

DATE	NIKKEI225	DEXJPUS
1971-01-04	7.601572	5.879779

In [34]:
```
fig = plt.figure(figsize=(8,4))
ax1 = fig.add_subplot(2,2,1)
ax1.plot(n225,alpha=0.5,color='black',label="日経225")
plt.ylabel('日経平均株価')
plt.title("図3.1a: 日経平均株価")
plt.xticks(rotation=90)
plt.legend(loc='upper left', bbox_to_anchor=(1, 1))
#-----------------------------------------------------
ax2 = fig.add_subplot(2,2,2)
ax2.plot(fx,alpha=0.5,color='black',label="ドル円")
plt.xticks(rotation=90)
plt.legend(loc='upper left', bbox_to_anchor=(1, 1))
plt.title("図3.1b: ドル円の為替レート")
#-----------------------------------------------------
ax3 = fig.add_subplot(2,2,3)
ax3.hist(n225,alpha=0.5,color='black',label="日経225")
plt.xticks(rotation=90)
plt.legend(loc='upper right')
plt.title("図3.1c: 日経平均株価の頻度図")
#-----------------------------------------------------
ax4 = fig.add_subplot(2,2,4)
ax4.scatter(port.iloc[:,1],port.iloc[:,0],color='blue',s=1)
plt.xlabel('ドル円')
plt.ylabel('日経平均株価')
plt.xticks([np.log(100),np.log(200),np.log(300)],[100,200,300])
plt.yticks([np.log(1000),np.log(10000),np.log(30000)],[1000,10000,30000])
plt.xticks(rotation=90)
plt.title("図3.1d: 日経平均株価とドル円レートの散布図")
plt.tight_layout()
plt.show()
```



3 2軸のグラフ

y軸の右と左の軸を利用して、目盛りを2つもつグラフを描くことができる。

Python 5

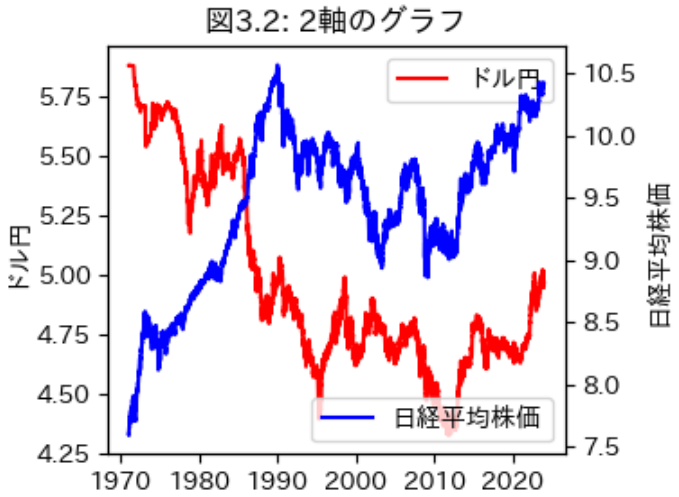
Axes.twinx()

x軸を共有しながら2つのｙ軸をもつグラフを作成できる。

set_ylabel: y軸のラベルの設定

set_xlabel：x軸のラベルの設定

In [35]:
```
fig, ax1 = plt.subplots(figsize=(4,3))
ax2 = ax1.twinx()
ax1.plot(port.iloc[:,1],color='r',label="ドル円")
ax1.legend()
ax2.plot(port.iloc[:,0],color='b',label="日経平均株価")
ax1.set_ylabel('ドル円')
ax2.set_ylabel('日経平均株価')
ax2.legend()
plt.title('図3.2: 2軸のグラフ')
plt.tight_layout()
#plt.savefig("図3.2.jpeg",dpi=600)
plt.show()
```



1.13 plt.show()

Python のデータ可視化ライブラリである Matplotlib の一部で、作成したグラフやチャートを表示するために使用されます。

Matplotlib でグラフを作成した後、 `plt.show()` を呼び出すと、現在アクティブな図や軸に対するすべての描画内容が表示されます。この関数は、図をスクリーンに描画し、ユーザーがそれを見ることができるようにします。

2 Pandas入門

Pythonでは目的に応じたデータ分析を可能にするために、幾つかのデータ構造が構築された。

1. Pythonに組み込まれた3つのデータ構造：
   a. リスト、
   b. タプル、そして
   c. 辞書

   を用いれば、主なデータ処理のすべてが可能となる。
2. NumPy（Numerical Pythonの略)の効率的な多次元配列は、配列指向の算術演算機能を提供する。Cでプログラムされ たAPIにデータを渡し、その演算結果をNumPy配列として受け取るという離れ業はPythonを使う理由の一つだ。
3. pandasは、統計解析や表形式のデータ解析に用いられる。Seriesは基本的にNumPy配列の代替となり、DataFrameはデータを表として表現し、順序や名前を持つ列を、そのデータ型が異なっても処理することが可能であることから、多用される。

In [36]:
```
import pandas_datareader.data as web
import pandas as pd
from pandas.tseries.frequencies import to_offset
import numpy as np
import datetime
```

In [37]:
```
data=[10000,20000,30000]
Series=pd.Series(data,)
Series
```

Out[37]:
```
0    10000
1    20000
2    30000
dtype: int64
```

この場合、自動的にインデックスを0,1,2,3と付けてくれた。これはリストでもNumPy配列でも可能だ。つぎに

月	10000
火	20000
水	30000

ではどうだろうか？

In [38]:
```
data=[10000,20000,30000]
index=['月','火','水']
series=pd.Series(data,index=index,name='価格')
series.index.name='曜日'
series
```

Out[38]:
```
曜日
月    10000
火    20000
水    30000
Name: 価格, dtype: int64
```

曜日(文字列)をインデックスとして指定している。これは辞書と同じ機能だ。しかし、Seriesでは列の数をこれ以上に増やすことはできない。Seriesについてまとめておこう。

Python 2.1

Series	**(data=None, index=None, name=None)**
data:	スカラー、リスト、タプル、辞書など
index:	1次元リストなど、ディフォルトはRangeIndex(0,1,2,...,n)
name:	Seiresの名前

列の数を増やしたいのであれば、DataFrameの出番だ。

月	1	10000
火	2	20000
水	3	30000

In [39]:
```
data=[[1,10000],[2,20000],[3,30000]]
index=['月','火','水']
frame=pd.DataFrame(data,index=index,columns=['個数','価格'])
frame.index.name='曜日'
frame
```

Out[39]:

曜日	個数	価格
月	1	10000
火	2	20000
水	3	30000

DataFrameを使えば、簡単に列(要因)の数を増やすことができる。ここでは、dataにリストを用いたが、辞書でもSeriesでも可能だ。出力[5]に注目してほしい。Seriesの出力[3]と比べてみよう。Seriesの場合は、そっけないが、DataFrameの場合は、色分けもされ、線もひかれていて見やすい。これがDataFrameの1つの特徴で表の作成にも使える。DataFrameのリファレンスをPython 2.2にまとめた。

Python 2.2

DataFrame	(data=None, index=None, columns=None)
data:	Series, DataFrame,ndarray,リスト,タプル、辞書など
index:	Series, リストなど、ディフォルトはRangeIndex(0,1,2,...,n)
columns:	列の名前

ここで、配列の表記の方法はリレーショナルデータベースでの基本なので復習しておこう。

	列1	列2
行1	要素(1,1)	要素(1,2)
行2	要素(2,1)}	要素(2,2)

行番号と列番号を指定すると要素を抽出できる。Seriesであれば行番号を指定すると要素を確定できる。行番号の指定はインデックスを用いて行われる。

2.1.1 head()とtail()

データ行の最初の部分と最後の部分に興味のあるときは、head()とtail()が便利である。head()は最初の5つのデータを表示し、tail()は最後の5つのデータを表示する。また、head(n),tail(n)とするとn個のデータを表示してくれる。

In [40]:
```
print('＊　Seriesのhead2つ分')
print(Series.head(2))
print('＊　DataFrameのtail2つ分')
print(frame.tail(2))
```

```
＊　Seriesのhead2つ分
0    10000
1    20000
dtype: int64
＊　DataFrameのtail2つ分
     個数      価格
曜日
火     2  20000
水     3  30000
```

2.2 時系列データの生成

　時系列データはインデックスを時間経過とするデータの集まりのことである。金融市場の終値だけからなるデータは終値を列としたSeriesで構築し、4本値のような場合には始値、高値、安値、終値を列としたDataFrameで構築する。インデックスにはDatetimeIndexが用いられる。それはdatetimeモジュールにより実装される。datetime型は日付けと時刻からなる。　入力[7]では、datesをdatetime型としている。終値を乱数として生成し、Seriesのデータとして採用し、インデックスをdatesとしている。列名は'終値'である。インデックス名はDateである。printを用いて内容を表示している。　つぎに、4本値をやはり乱数として発生させ、データ構造をDataFrameとした。インデックスはdatesである。列名は'始値','高値','安値','終値'とした。インデックス名はDATEである。結果は出力[7]に表示される。

In [41]:
```
from datetime import datetime
dates=[datetime(2023,1,2),datetime(2023,1,5),datetime(2023,1,7)]
ts=pd.Series(np.random.standard_normal(3),index=dates,name='終値')
ts.index.name='Date'
print(ts)
```

```
Date
2023-01-02    0.685715
2023-01-05   -0.261772
2023-01-07    0.373807
Name: 終値, dtype: float64
```

In [42]:
```
tsd=pd.DataFrame(np.random.standard_normal((3,4)),index=dates,columns=['始値','高値','安値','
tsd.index.name='DATE'
tsd
```

Out[42]:

DATE	始値	高値	安値	終値
2023-01-02	-2.214418	2.179293	-0.403152	0.557114
2023-01-05	0.845614	-0.445161	2.225274	-0.416796
2023-01-07	0.456155	-0.804494	0.298718	0.499966

2.2.1 列の追加

　DataFrameの列の追加の機能を利用して、データを結合することができる。本書ではこの方法を多用する。Nikkei225のコピーをportとして作成し、FREDから取得したドル円の為替レートをDEXJPUSとして追加する。

In [43]:
```
fx = web.DataReader('DEXJPUS',"fred",start)
port=n225.copy()
port['DEXJPUS']=fx
port.tail(3)
```

Out[43]:
```
DATE
2024-01-12 00:00:00                                          35577.11
2024-01-15 00:00:00                                          35901.79
DEXJPUS                              DEXJPUS
DATE
1971-0...
Name: NIKKEI225, dtype: object
```

2.3 データの選択

2.3.1 インデックスのラベルを用いたデータの選択(フィルタリング)

オブジェクト名(obj)の後に[ ]を付け、インデックスのラベルを用いデータまたは行を指定する。

obj['a']

2.3.2 ブール型配列、数式を用いたデータの選択(フィルタリング)

インデックスのラベルの代わりに、ブール型配列や数式を用いてブール型配列を生成してデータ、または行を選択する。

ojb[obj>1]

2.3.3 locを用いた方法

.locの入力としては、

- ラベル('a')
- リスト(['a','b',...,'z'])
- スライス('a' : 'z')
- ブール配列型数式(obj[obj>1])

を用いて必要なデータを取得できる。　特定の行と列を取得したいときは、行をまず指定して、列を指定する。loc[行の指定、列の指定].また、行のみ指定するにはloc[行の指定]、列のみ指定したいときはloc[:,列の指定]とする。

2.3.4 ilocを用いた方法

.ilocでは取得するデータは

- 整数
- 整数のリスト
- 整数のスライス

で指定する。行と列を指定したいときは、iloc[行の指定、列の指定]で行う。行のみ指定するにはiloc[行の指定]、列のみ指定したいときはiloc[:,列の指定]とする。

入力[12]の例では日付の文字列を用いて、データを選択している。必ずしもdatetime型である必要はない。

In [44]:
```
Nikkei225 = web.DataReader("NIKKEI225", 'fred',start)
port=pd.concat([Nikkei225,fx],axis=1)#.dropna()
```

In [45]:
```
print("＊ 2.3.1.の例")
print(ts['2023/1/2'])
print("＊ 2.3.2の例")
print(port["2022-01-04":"2022-01-07"])
print("＊ 2.3.2の数式の例")
print(port[port['NIKKEI225']>38500].index)
print("＊ 2.3.3のlocの例")
port_after1990=port.loc["1990-01-04":]
print(port_after1990.loc["1990-01"])
print("＊ 2.3.4のilocの例")
print(port.iloc[-2:])
port.index[0]
```

```
＊ 2.3.1.の例
0.6857147269968653
＊ 2.3.2の例
            NIKKEI225  DEXJPUS
DATE
2022-01-04   29301.79   116.12
2022-01-05   29332.16   115.91
2022-01-06   28487.87   115.78
2022-01-07   28478.56   115.61
＊ 2.3.2の数式の例
DatetimeIndex(['1989-12-18', '1989-12-20', '1989-12-26', '1989-12-27',
               '1989-12-28', '1989-12-29', '1990-01-04'],
              dtype='datetime64[ns]', name='DATE', freq=None)
＊ 2.3.3のlocの例
            NIKKEI225  DEXJPUS
DATE
1990-01-04   38712.88   143.37
1990-01-05   38274.76   143.82
1990-01-08   38294.96   144.10
1990-01-09   37951.46   145.20
1990-01-10   37696.50   145.25
1990-01-11   38170.13   145.40
1990-01-12   37516.77   145.40
1990-01-15        NaN      NaN
1990-01-16   36850.36   145.48
1990-01-17   36821.14   145.49
1990-01-18   36729.49   146.12
1990-01-19   36836.54   145.65
1990-01-22   37257.00   146.43
1990-01-23   37378.02   146.09
1990-01-24   36778.98   145.32
1990-01-25   36969.11   144.42
1990-01-26   36874.07   143.35
1990-01-29   37173.70   143.13
1990-01-30   37215.67   144.10
1990-01-31   37188.95   144.55
＊ 2.3.4のilocの例
            NIKKEI225  DEXJPUS
DATE
2024-01-12   35577.11      NaN
2024-01-15   35901.79      NaN
```

Out[45]:
```
Timestamp('1949-05-16 00:00:00', freq='B')
```

行と列を同時に指定する例

In [46]:
```
port.iloc[-2:,0:]
```

Out[46]:

DATE	NIKKEI225	DEXJPUS
2024-01-12	35577.11	NaN
2024-01-15	35901.79	NaN

同じデータを取得するにもいろいろな方法がある。

2.3.5 asof

指定した日時からNaNsを除いた最後のデータの取得,グラフを表示に用いたりする。

In [47]:
```
Nikkei225.asof("2022-12-28")
```

Out[47]:
```
NIKKEI225    26340.5
Name: 2022-12-28 00:00:00, dtype: float64
```

2.4 時系列データの結合と削除

時系列データはインデックスとなる日付と要素で構成されている。要素が1種類であれば単変量、複数あれば多変量と呼ばれる。たとえば、終値だけだと単変量で、4本値は多変量である。

2.4.1データの結合(concat)

日経平均株価の推移をドル円の為替レートとの関連で調べたい場合には、この2つのデータを結合すると便利である。そのようなときには

Python 4.5

pandas.concat	**(objs,axis=0)**
objs:	SeriesまたはDataFrameのシーケンス
axis:	結合する軸方向

を用いる。

In [48]:
```
port=pd.concat([Nikkei225,fx],axis=1)#.dropna()
print(port.head())
port=port.dropna()
print(port[:1])
print(port[-1:])
```

```
            NIKKEI225  DEXJPUS
DATE
1949-05-16     176.21      NaN
1949-05-17     174.80      NaN
1949-05-18     172.53      NaN
1949-05-19     171.34      NaN
1949-05-20     169.20      NaN
            NIKKEI225  DEXJPUS
DATE
1971-01-04    2001.34   357.73
            NIKKEI225  DEXJPUS
DATE
2024-01-05   33377.42   144.52
```

双方に共通したIndexがない行の列にはNaN(Not a Number:欠測値)が表示される。dropna()は欠測値をもつ行を削除してくれる。このように双方のデータに共通の取引日があるデータのみを選ぶことができる。また、すべてのデータを受け入れて、片方の取引日に欠測値がある場合には前日の価格を用いて、代替することも可能である。　データの結合は、外れ値の処理、重複データの処理、重複日付の処理、欠測値の処理などと密接な関係にあり、詳細はPython for Data Analysisなどの専門書を参照してほしい。

2.4.2 行・列の削除

dropを用いた行と列の削除を試してみよう。axis=1で列の削除、axis=0で行の削除となる。

In [49]:
```
port=port.drop('DEXJPUS',axis=1)
print(port.head(2))

port=port.drop('1971-01-04',axis=0)
print(port.head(2))
```

```
            NIKKEI225
DATE
1971-01-04    2001.34
1971-01-05    1989.44
            NIKKEI225
DATE
1971-01-05    1989.44
1971-01-06    1981.74
```

2.5 データの加工、分析

データの分析は静的な分析と動的な分析に分けることができる。 静的な分析は記述統計のようにデータの期間を固定して、統計量を求める分析手法である。 一方、動的な分析とは同じように統計量を求めるのであるが、 全体の期間の中にそれよりも短い期間を設け、その期間をある一定量づつずらしていく分析手法である。

2.5.1 記述統計と要約統計量　pandasのSeriesとDataFrameも要約統計量についての関数をもっている。

count	計数	quantitle	4分位	sum	合計	mean	平均
median	中央値	var	分散	std	標準偏差	skew	歪度
kurt	尖度	cumsum	累積合計	diff	差分	pct_change	変化率

などである。ここでは日経平均株価の変化率のデータ数、平均、標準偏差、歪度と尖度を求めてみよう。

In [50]:
```
tsd=Nikkei225.pct_change().dropna()
print("データ数 ",tsd.count()[0])
print("平均",tsd.mean()[0])
print("標準偏差",tsd.std()[0])
print("歪度",tsd.skew()[0])
print("尖度(0)",tsd.kurt()[0])
```

```
データ数　19480
平均 0.00034509772787385753
標準偏差 0.012000987509144575
歪度 -0.12803182168387905
尖度(0) 9.722199143118564
```

2.6 繰り返し処理

DataFrameのforループにおいて一行ずつデータを取り出す方法には、iterrowsメソッドとitertuplesメソッドがある。

Python 4.8

DataFrame.iterrows	()
	iterrowsはそれぞれの行のSeriesを返す。
DataFrame.itertuples	(index=True, name='Pandas')
index:	真のときにはインデックスを返す。
name:	文字列、ディフォルトは"Pandas"
返り値	イテレータ

疑似時系列データを作ってどちらが効率よく処理を実行するか試してみよう。DatetimeIndexと乱数を生成して、時系列データを合成する。それを読み取り経過時間を測定する。また、numpy配列とリストのイテレータとラベルで要素を取得する場合について試してみよう。

In [51]:
```
index=pd.date_range(start="2023/1/1",periods=1000000,freq="min")
from numpy.random import default_rng
rng=default_rng()
rnd=rng.standard_normal((1000000,4))
frame=pd.DataFrame(rnd,index=index)
start_now=datetime.now()
print("itterows")
for a,b in frame.iterrows():
    date=a
    o=b[0]
    h=b[1]
    l=b[2]
    c=b[3]
print(datetime.now()-start_now,'-----------------------')
```

```
itterows
0:00:24.374696 -----------------------
```

In [52]:
```
start_now=datetime.now()
print("itteturples")
for b in frame.itertuples():
    o=b[1]
    h=b[2]
    l=b[3]
    c=b[4]
print(datetime.now()-start_now,'-----------------------')
```

```
itteturples
0:00:01.459068 -----------------------
```

In [53]:
```
start_now=datetime.now()
print("numpy array")
for b in rnd:
    o=b[0]
    h=b[1]
    l=b[2]
    c=b[3]
print(datetime.now()-start_now,'-----------------------')
```

```
numpy array
0:00:00.357263 -----------------------
```

In [54]:
```
rndlist=rnd.tolist()
start_now=datetime.now()
print("list")
for b in rndlist:
    o=b[0]
    h=b[1]
    l=b[2]
    c=b[3]
print(datetime.now()-start_now,'-----------------------')
```

```
list
0:00:00.168820 -----------------------
```

In [55]:
```
start_now=datetime.now()
print("arange + numpy array")
for i in np.arange(len(rnd)):
    o=rnd[i][0]
    h=rnd[i][1]
    l=rnd[i][2]
    c=rnd[i][3]
print(datetime.now()-start_now,'-----------------------')
```

```
arange + numpy array
0:00:01.035330 -----------------------
```

In [56]:
```
start_now=datetime.now()
print("range+list")
for b in range(len(rndlist)):
    o=rndlist[i][0]
    h=rndlist[i][1]
    l=rndlist[i][2]
    c=rndlist[i][3]
print(datetime.now()-start_now,'-----------------------')
```

```
range+list
0:00:00.350540 -----------------------
```

In [ ]:
```

```