
Pythonではじめる データ分析基礎講座

～ データ加工編 ～

本研修の目的と範囲

◆ 本研修の目的

- ・AI開発に必要なプログラミング言語「Python」スキルを習得した人材を増やすため。
- ・機械学習エンジニア・データ分析エンジニアを目指す人のスキルアップとして。

◆ 研修範囲

プログラミング言語「Python」を使用して

AI開発・データ分析を行う際の前処理で必要となる「データ加工方法」を中心に学びます。

またデータの要約・集計 及び データの可視化を行い、データの概要を把握する方法を学びます。

研修スケジュール

2

1. Python基礎知識
2. DataFrame(DF)の作成
3. DataFrame(DF)の確認

1日目

4. データの加工
5. データの要約・集計
6. データの可視化
7. 総合演習

2日目

※1日目の進捗により、2日目の開始章が前後する場合があります。

Python基礎知識

この章で学ぶこと ～Python基礎知識～

4

- Pythonについての基礎情報
- Pythonの基礎文法

Pythonってどんな言語なの？

5

Python とは



汎用プログラミング言語であり、以下のような特徴を持ちます。

- オープンソースプログラミング言語である
- 文法がシンプルであり、コードが少量で済む
- 文法によりインデント位置が決められており、可読性が高い
- Web開発、データ解析(AI)、ゲームといった幅広い分野で使用されている
- 多彩な**ライブラリ**サポートで高度な計算も容易

『 Youtube 』『 EverNote 』『 Instagram 』に利用されています。

Pythonってどんな言語なの？

6

オープンソースプログラミング言語 とは

自由に使用でき、自由に配布でき、商用利用も可能な言語のことです。

Pythonの他には『 Ruby 』『 Perl 』などがあります。

自由に使用できるため、さまざまな人がソフトウェアを作成しています。

汎用的な処理はライブラリとしてPyPIに公開することができ、

公開されているものは自由に利用することができます。

Pythonってどんな言語なの？

Python の起源

1991年 オランダ人のガイド・ヴァン・ロッサム氏によって開発されたプログラミング言語。
名前の由来は、イギリスのテレビ局BBCが製作・放送した大ヒットコメディ番組である
「空飛ぶモンティ・パイソン」からきていとされています。

6年以上前の1989年12月、私はクリスマス前後の週の暇つぶしのため「趣味」のプログラミングプロジェクトを探していた。オフィスは閉まっているが、自宅にはホームコンピュータがあるし、他にすることがなかった。私は最近考えていた新しいスクリプト言語のインタプリタを書くことにした。それは、ABCからの派生であり、Unix/Cハッカーの注意をひきつけるかもしれないと考えた。ちょっとしたいたずら心から(『空飛ぶモンティ・パイソン』の熱烈なファンだったというのも理由の1つ)、プロジェクトの仮称をPythonにした。
ー ガイド・ヴァンロッサム、「Programming Python」の序文



ガイド・ヴァン・ロッサム(出典「wikipedia」)

Pythonってどんな言語なの？

Python の文法は本当にシンプルなのか？

2つの値(a, b)の最大公約数を求めるプログラムを
Python, Java, Rubyの3つの言語で比較。

※最大公約数を求めるアルゴリズムはユークリッドの互除法を使用

Python

```
def gcd(a, b):  
    while b != 0:  
        a, b = b, a % b  
    return a
```

Java

```
private static long gcd(long a, long b) {  
    long candidate = a;  
    while (b % a != 0) {  
        candidate = a % b;  
        a = b;  
        b = candidate;  
    }  
    return candidate;  
}
```

Ruby

```
def gcd(a, b)  
    until b == 0  
        a, b = b, a % b  
    end  
    return a  
end
```

The Zen of Python

9

Pythonには“禅”と呼ばれる設計思想があります。

```
import this
```

Beautiful is better than ugly.

醜いより美しいほうがいい。

Explicit is better than implicit.

暗示するより明示するほうがいい。

Flat is better than nested.

ネストは浅いほうがいい。

Sparse is better than dense.

密集しているよりは隙間があるほうがいい。

Readability counts.

読みやすいことは善である。

…etc

豊富なライブラリ

ライブラリとは、
多彩な計算、データ加工を可能とする、モジュール(Pythonプログラム)群。
Pythonのライブラリには、標準ライブラリと外部ライブラリが存在し、
高度な処理を行う場合は外部ライブラリの活用が有効です。
データ加工、機械学習など、多彩なライブラリが存在します。

<例>	ライブラリ名	用途	標準/外部 ライブラリ
	Datetime	日付時間処理	標準
	Math	数学計算	標準
	Numpy	行列計算	外部
	Pandas	データ加工	外部
	Matplotlib	グラフ描画	外部
	scikit-learn	機械学習	外部
	Tensorflow	深層学習	外部

Anaconda

Anacondaはデータサイエンス向けに作成されたPythonパッケージで、Python本体と、データサイエンスでよく利用されるライブラリが同梱されています。

基本的なライブラリは抑えられているので、追加でライブラリをインストールすることなく利用することができます。
※もちろん追加でライブラリをインストールすることも可能です。



<https://www.anaconda.com/distribution/>

Jupyter Notebookとは

ブラウザ形式のテキストエディタ。(Anacnda同梱)
ノートブックと呼ばれる形式でプログラムを作成でき、
実行結果を確認しながら作業を進めるためのツールです。



<実行画面>

A screenshot of a Jupyter Notebook interface. The top bar shows the Jupyter logo, the text "python研修", and a "Logout" button. Below the bar is a menu (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations, running, and other functions. The main area contains three code cells. The first cell (In [1]) contains "import pandas as pd". The second cell (In [22]) contains "#CSVデータの読み込み" followed by two lines of code to read CSV files. The third cell (In [24]) contains "df_burger[0:5]". Below the third cell, the output (Out[24]) is displayed as a table with 10 columns: shop_id, gender, age, howto, eat, bunrui, order, kingaku, and stay. The table contains 5 rows of data.

```
In [1]: import pandas as pd

In [22]: #CSVデータの読み込み
df_burger = pd.read_csv('burger.csv',encoding='s-jis')
df_shop_label = pd.read_csv('shop_label.csv',encoding='s-jis')

In [24]: df_burger[0:5]

Out[24]:
```

	shop_id	gender	age	howto	eat	bunrui	order	kingaku	stay
0	0	女	18	徒歩	店内	BURGER	チーズバーガー	230.0	180
1	0	男	22	自動車	持ち帰り	BURGER	ハンバーガー	210.0	0
2	0	女	12	自動車	持ち帰り	BURGER	てりやきバーガー	270.0	0
3	0	女	22	徒歩	店内	BURGER	フィッシュバーガー	230.0	180
4	0	女	21	バイク	店内	BURGER	ホットドッグ	370.0	180

処理記載

結果表示

例題1－1)

13

Hello World を表示してください。

```
print('Hello World')
```

結果

Hello World

データタイプ(型)について

14

Pythonでは数値、文字列、ブール値に加えて、複数のデータを扱うコンテナ(リスト型・タプル型・辞書型)が用意されています。

ここでは数値、文字列、ブール値、リスト型・タプル型の記述方法について紹介します。

※辞書型については後程紹介します。

```
# 数値：囲み文字を指定しない場合
num = 123
# 文字列：シングルクォーテーション'またはダブルクォーテーション"で囲む
txt = '123'
# ブール値：TrueまたはFalse(1文字目は大文字)
flg = True
# リストの作成
list1 = [0, 1, 2, 3, 4]          ←"[]"の中に値を設定し定義
list2 = ['A', 1, 'B', 3, 'C']    ←数値、文字列の混合も定義可能
list3 = [['A', 1], ['B', 3, 'C']] ←リストを入れ子にすることも可能
# タプルの作成(リストとの違いはイミュータブル(変更不可能)な点。カレンダー等の作成に使用)
tuple = (0, 1, 2, 3)            ←"()"の中に値を設定し定義
```

例題1－2)

15

(1) 各変数定義の内容と型を表示してください。

```
# 数値
num = 123
print(num)
print('num:', type(num))
```

```
# 文字列
txt = '123'
print(txt)
print('txt:', type(txt))
```

```
結果
123
num: <class 'int'>
123
txt: <class 'str'>
```

データ型名 (省略形)	内容	値の例	省略なし
strデータ型	文字列	“Python”	string
intデータ型	整数	123	integer
floatデータ型	浮動小数点数	123.123	float
boolデータ型	ブール値(真偽値)	TrueまたはFalse	bool
NoneTypeデータ型	値が存在しない	None	NoneType

例題1－2)

16

(2) リストの内容と型を表示してください。

```
# []内に値を設定し定義
list1 = [0, 1, 2, 3, 4]
print('list1:', list1, type(list1))

# 数値、文字列の混合も定義可能
list2 = ['A', 1, 'B', 3, 'C']
print('list2:', list2, type(list2))

# リストを入れ子にすることも可能
list3 = [['A', 1], ['B', 3, 'C']]
print('list3:', list3, type(list3))
```

結果

```
list1: [0, 1, 2, 3, 4] <class 'list'>
list2: ['A', 1, 'B', 3, 'C'] <class 'list'>
list3: [['A', 1], ['B', 3, 'C']] <class 'list'>
```

Pythonには辞書型(ディクショナリ型)といわれる配列の型も存在します。実際の辞書のように、ある値(key)に対してそれに対応する値(value)が存在します。

辞書型はkeyに対するvalueを持っているため、対応する値をすぐに呼び出すことができます。一方リストは、対応関係は保持できませんが、順番を保持できるというメリットがあります。

<辞書型の作成>

```
辞書 = {'key1':value1, 'key2':value2, 'key3':value3...}
```

<辞書型の利用>

```
# keyに対応するvalueを返す  
辞書['key']
```

例題1－3)

18

keyが'A'値が[1, 3, 5]、keyが'B'値が[2, 4, 6]となる辞書を作成し、key'A'を表示してください。

```
dictionary = {'A':[1, 3, 5], 'B':[2, 4, 6]}  
dictionary['A']
```

結果

[1, 3, 5]

演習1－1

19

- (1) 数値0～4を順に格納したリストと、文字'A'～'E'を順に格納したリストを作成してください。
- (2) keyに数値0～4、それぞれのvalueに文字'A'～'E'を格納した辞書を作成してください。

プログラムの記述について

インデント、コメントの記述方法について紹介します。

- インデントについて

Pythonでは、実行文をグループ(for文やif文)でまとめる為に、下記の様に、タブやスペースでインデントの付与が必要です(インデントがない場合エラー)。インデントは半角スペース4つ分が標準的です。

<for, ifを使用した繰り返しの処理例>

```
for i in data:
```

```
    □ if i == 'AAA':
```

```
        □ □ print('AAA')
```

```
    □ else:
```

```
        □ □ print('BBB')
```

インデントで左記範囲をグループ化する

(□は、タブ、またはスペース4つ)

- コメントについて

実行文の先頭に『#』を付与することでコメント化(処理されない)します。

```
# a + b ←こちらはコメント
```

```
a + b ←こちらは非コメント
```

制御フロー ～条件分岐処理(if、elif、else)～

21

if文:条件分岐処理を行います。

Pythonのif文は、条件部と処理部をインデントで明確に書き分ける必要があります。
また、elifを繰り返すことで複数の条件を判定させることができます。

<if文の構成>

```
if <条件1>:  
    条件1に一致した場合の処理  
  
elif <条件2>:  
    条件2に一致した場合の処理  
  
else:  
    条件に一致しなかった場合の処理
```

例題1－4)

22

xに任意の数値を設定し、偶数・奇数を判定してください。
($a \% b$ で、aをbで割った余りが求められます。)

```
x = 20
if (x % 2) == 0:
    print('偶数：', x)
else:
    print('奇数：', x)
```

結果

偶数： 20

制御フロー～反復処理(for文)～

for文:反復処理を行います。

Pythonのfor文は、任意のシーケンス型(リストまたは文字列)にわたって反復を行います。
反復の順番はシーケンス中に要素が現れる順番です。

<for文の構成>

```
for i in <任意のシーケンス型>:  
    反復される処理  
i <任意のシーケンス型>から順番に取り出される値
```

指定回数の反復を行う場合、range()を使用します。

```
for i in range(start, stop):  
    反復される処理
```

range()のオプション

start=None stop=必須 [ex: start=None stop=100の場合 range(100)]
startからstopまでの数値を順番に返す(startを省略した場合は0からstop)
startの値は含むが、stopの値は含まない

例題1－5)

24

(1) kingakulist内の数値をすべて表示してください。

```
kingakulist = [250, 280, 340, 200, 100, 500]

for i in kingakulist:
    print(i)
```

結果

```
250
280
340
200
100
500
```

例題1－5)

25

(2) 'hello'を1文字ずつ表示してください。

```
word = 'hello'

for i in word:
    print(i)
```

結果

```
h
e
l
l
o
```

例題1－5)

26

(3) for文を実行し結果を確認してください。

```
for i in range(10):  
    print(i)
```

結果

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

例題1－5)

27

(4) 20未満の数値に対して、偶数・奇数を判定してください。

```
for i in range(20):  
    if (i % 2) == 0:  
        print('偶数:', i)  
    else:  
        print('奇数:', i)
```

結果

```
偶数: 0  
奇数: 1  
偶数: 2  
奇数: 3  
偶数: 4  
奇数: 5  
偶数: 6  
:  
:
```

制御フロー～反復処理(while文)～

28

while文

条件式の値が真である間、実行を繰り返します。

条件式を満たす限り繰り返される為、無限に続いてしまう可能性があることに注意してください。

< while文の構成 >

```
while <条件式>:  
    繰り返し処理
```

break、continueによる反復処理制御

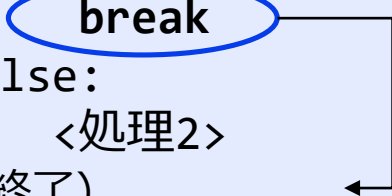
29

break, continue

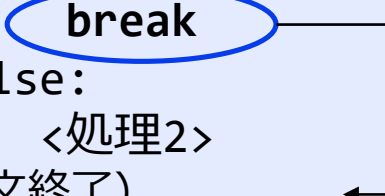
反復の途中で処理を中断する場合、break, continueを利用します。

<反復処理を抜ける場合>

```
for <反復条件>:  
    if <分岐条件> :  
        <処理1>  
        break  
    else:  
        <処理2>  
(for文終了)
```

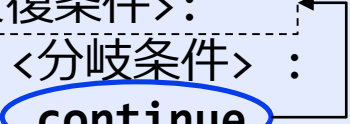


```
while <反復条件>:  
    if <分岐条件> :  
        <処理1>  
        break  
    else:  
        <処理2>  
(while文終了)
```

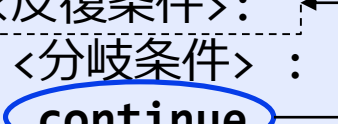


<反復処理に戻る場合>

```
for <反復条件>:  
    if <分岐条件> :  
        continue  
    <処理1>
```



```
while <反復条件>:  
    if <分岐条件> :  
        continue  
    <処理1>
```



例題1－6)

30

(1) 1～20未満の数値を順番に加算し、加算した数と累積和を表示してください。
ただし、合計が100を超えたら処理を止めること。

```
x = 0
for i in range(1, 20):
    x = x + i
    if x >= 100:
        print(i, x)
        break
    else:
        print(i, x)
```

結果

```
1 1
2 3
3 6
4 10
⋮
```

例題1－6)

31

(2) numlistから10より大きい値を出力してください。
ただし、if文内の処理にはcontinue以外記載しないこと。

```
numlist = [1, 5, 6, 2, 4, 9, 11, 3, 15, 20]

for number in numlist:
    if number < 10:
        continue
    print(number)
```

結果

```
11
15
20
```


(1) for文を使用し、1～20未満の数値で3の倍数のとき「3の倍数:数値」、それ以外は数値のみを出力してください。

結果例)

1

2

3の倍数:3

4

⋮

(2) while文を使用し、(1)と同様の結果を出力してください。

制御フロー～反復処理に利用できる関数～

33

下記関数を利用することで、多彩なループ処理を行うことができます。

<items() :ディクショナリ型からキーと対応する値を取得>

```
dic.items()
```

dic :ディクショナリ型データ

<enumerate() :リスト内要素に番号を付与しながら取得>

```
enumerate(list)
```

list :リスト型データ

<zip() :2つ以上のリストを同時にループ>

```
zip(list1, list2)
```

例題1－7)

34

(1) ディクショナリ型からキーと対応する値を取得し表示してください。

```
dica = {'a':111, 'b':222, 'c':333, 'd':444, 'e':555}
for i, j in dica.items():
    print(i, j)
```

結果

```
a 111
b 222
c 333
d 444
e 555
```

例題1－7)

35

(2) リスト内要素に番号を付与し表示してください。

```
listb = ['a', 'b', 'c', 'd', 'e']  
for i, j in enumerate(listb):  
    print(i, j)
```

結果

```
0 a  
1 b  
2 c  
3 d  
4 e
```

例題1－7)

36

(3) 2つのリストを同時にループし表示してください。

```
lista = [0, 1, 2, 3, 4]
for i, j in zip(lista, listb):
    print(i, j)
```

結果

```
0 a
1 b
2 c
3 d
4 e
```

文字列操作

37

文字列は関数により、まとめて操作することが可能です。

<文字列操作>

str.操作関数()

操作関数	説明
Upper()	英字を大文字にする
Lower()	英字を小文字にする
Replace()	指定された文字を置き換える
Strip()	前後の空白を削除する
Isalpha()	アルファベットか判定する
Isdigit()	数値化判定する
Center()	文字を中央に揃える
Split()	指定文字で分割

例題1－8)

38

(1) 文字列 'Hello World' の 'o' を '*' に置換してください。

```
word = 'Hello World'
word = word.replace('o', '*')
print(word)
```

結果

Hell* W*rld

様々なリストの操作、リスト内包表記

リストには様々な操作の関数が用意されています。

<変数の度数集計>

リスト.操作関数()

操作関数

append(): 引数をリストに追加

extend(): 引数のリストで延長

insert(): 引数を指定位置に挿入

remove(): 引数をリストから1つ削除

reverse(): リストを逆順にする

index(): 引数の要素を検索

count(): 引数の出現回数をカウント

sort(): リストをソート

<リスト内包表記>

forやifを利用してリストを定義する方法があり、これをリスト内包表記と呼びます。リスト内包表記は簡潔に書けるほか、実行速度に優れます。

```
#リスト等の中の変数の中で条件を満たすもの
[変数(出力) for 変数 in リスト等 (if 条件)]
```

※set内包表記と、辞書内包表記も存在

例題1－9)

40

(1) 100未満の奇数のリストodd_numを作成してください。

```
odd_num = []  
for i in range(100):  
    if (i % 2) == 1:  
        odd_num.append(i)  
odd_num
```

結果

```
[1,  
 3,  
 5,  
 7,  
 9,  
 ⋮
```

例題1－9)

41

(2) 100未満の奇数のリストodd_numを内包表記で作成してください。

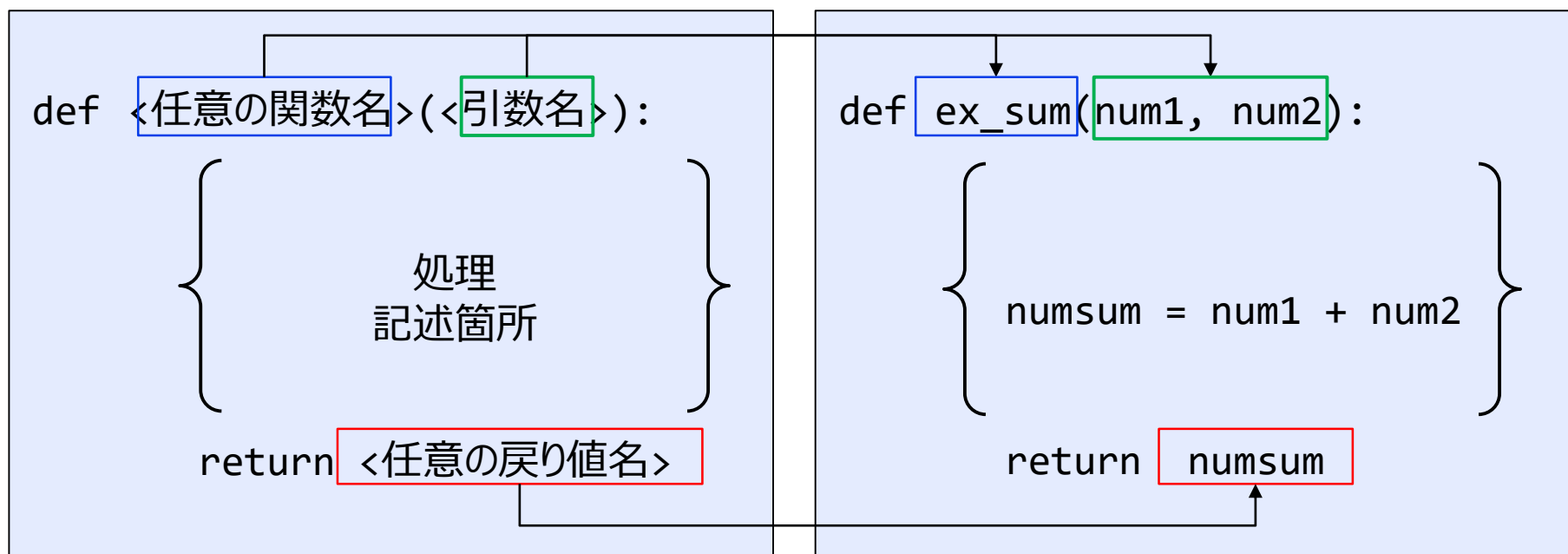
```
odd_num = [i for i in range(100) if (i % 2) == 1]  
odd_num
```

結果

```
[1,  
 3,  
 5,  
 7,  
 9,  
  ⋮
```

Pythonでは、同じ処理を複数回記載することを避けるために、defを使用し処理を関数として定義します。

<処理の関数化>



<関数の呼び出し>

<任意の関数名>(<引数名>)

『ex_sum』へ変数A,Bを設定し関数を呼び出し、Cを戻り値にします。

<呼び出し例>

A=3

B=5

C= ex_sum(A, B)

```
def ex_sum(num1, num2):
```

```
    numsum = num1 + num2
```

```
    return numsum
```

例題1－10)

44

(1) 引数に対して偶数・奇数を判定し、戻り値を返す関数を定義してください。

```
def hantei(num):  
    if (num % 2) == 0:  
        result = '偶数:' + str(num)  
    else:  
        result = '奇数:' + str(num)  
    return result
```

例題1－10)

45

(2) (1)で定義した関数を使用し、numlistの値を判定してください。

```
numlist = [1, 5, 3, 10, 12, 16]
for i in numlist:
    print(hantei(i))
```

結果

```
奇数:1
奇数:5
奇数:3
偶数:10
偶数:12
偶数:16
```

関数の定義 lambda

Pythonでは、ラムダ(lambda)式を使って無名関数を定義することができます。

<関数定義>

```
<任意の関数名> = lambda <引数名>: <関数定義>
```

<関数の呼び出し>

```
<任意の関数名>(<引数名>)
```

※但し、Pythonの規約(pep8)上、関数名の割当は推奨されず、割当てる場合はdefを使用

<例>

```
add = lambda a, b : a+b # lambda関数定義  
add(1, 2)  
# ⇒ 3
```

```
is_odd = lambda x : True if x%2 == 0 else False # If文も記載可能  
is_odd(4)
```

関数の定義 lambda ～引数に利用する～

ラムダ(lambda)式はdefに比べ簡潔に関数を定義できるため、関数を引数として定義するときに利用されます。

<sorted関数の例>

```
# その前にlambda宣言のおさらい
second = lambda x : x[1]
second('abc')
# ⇒ b   #x[1]で2番目の文字'b'が出力される

# xに入力された2文字目(数字)を使用してソートする
sorted(['a2','b1','c4','d3'],key = lambda x: x[1])
```

※sort と sortedの違い

sort ... list型のメソッド、戻り値無し 使い方 :list_a.sort() #list_aが変更される

sorted ... 組み込み関数、戻り値あり 使い方: list_b = sorted(list_a) #list_aは変更されない

例題1－11)

48

リストに含まれる文字列の2文字目で並び替えてください。

```
value1 = ['a3', 'b1', 'c2']  
  
print(sorted(value1, key = lambda x: x[1]))
```

結果

```
['b1', 'c2', 'a3']
```

演習1－3

49

- (1)数値の入力 i に対し、文字列 No_ i を返す関数を定義してください。
ex:3を入力すると No_3 と返る
- (2) (1)で作成した関数を、0～30未満の各数字に対して適用してください。

DataFrame(DF)の作成

- Pandasについて
- DataFrame(DF)についての基礎情報
- DataFrame(DF)の作成方法

pandasとは

Pythonでデータ加工を行う際に使用する外部ライブラリです。

データの読み込みから、加工、集計、グラフ化まで簡単に行うことができます。

pandasは主に以下の2種類のデータ構造を使います。

データに加えて行ラベル(DataFrameは列ラベルも)を持つ構造になります。

- Series : 1次元配列
- DataFrame : 2次元配列

【 Series 】

index	-
0	A
1	B
2	C

【 DataFrame 】

index	col1	col2	col3
0	A	D	G
1	B	E	H
2	C	F	I

DataFrame(DF)の構造

53

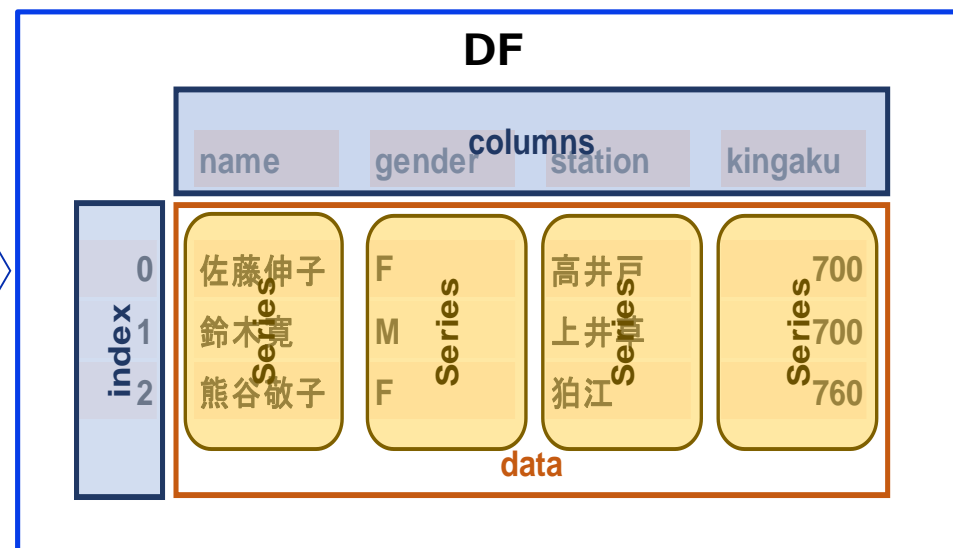
DataFrame(DF)の構造について紹介します。

DFは、以下の要素で構成されています。

- index : DF左端列、index(行番号)情報を保有
- columns : DF上端列、column(列名・列番号)情報を保有
- data : データの値、各列ごとにSeriesで保持されている

<構造イメージ>

	name	gender	station	kingaku
0	佐藤伸子	F	高井戸	700
1	鈴木寛	M	上井草	700
2	熊谷敬子	F	狛江	760



DataFrame(DF)の作成

DataFrame(DF)を定義します。

また、定義前にライブラリのインポートも記述します。

<ライブラリのインポート>

```
import pandas as pd
```

※ "as" は以降の記述で "pandas" を "pd" と記述する際に使用

<DataFrameの定義>

```
pd.DataFrame(data)
```

pd. DataFrameのオプション

data=必須

リスト

Series

DataFrameを指定

index=None

indexとして使用したいリスト(*)

columns=None

columnとして使用したいリスト(*)

(*)デフォルトは0から自動付与

例題2－1)

55

DataFrameを定義してください。

```
import pandas as pd

# データを定義
data = [['佐藤伸子', 'F', '高井戸', 700],
        ['鈴木寛', 'M', '上井草', 700],
        ['熊谷敬子', 'F', '狛江', 760]]

# column名を定義
columns = ['name', 'gender', 'station', 'kingaku']

# index名を定義
index = [0, 1, 2]

# データフレームを定義
df = pd.DataFrame(data, columns=columns, index=index)
```


DataFrame(DF)の作成

56

定義したDataFrameの内容や、項目の型を確認します。

<DFの確認>

DF

※変数名、リスト名、DF名のみを記述すると、実行結果に内容を表示(jupyterのみ)

例題2－2)

57

DataFrameの内容を確認してください。

```
df
```

結果

	name	gender	station	kingaku
0	佐藤伸子	F	高井戸	700
1	鈴木寛	M	上井草	700
2	熊谷敬子	F	狛江	760

DataFrame(DF)の作成

58

辞書を利用して、DataFrameを定義することもできます。

<辞書によるDataFrameの定義>

```
DF = pd.DataFrame({'列a':[値a1, 値a2···], '列b':[値b1, b2···]})
```

例題2－3)

59

辞書型のデータからDataFrameを定義してください。

```
dictionary = {'A':[1, 3, 5], 'B':[2, 4, 6]}  
odd_even = pd.DataFrame(dictionary)  
odd_even
```

結果

	A	B
0	1	2
1	3	4
2	5	6

(1) 以下の情報に列名('name', 'age', 'gender', 'favorite_food')をつけたDataFrameを定義してください。

```
['田中実', '52歳', '男', 'おにぎり'],  
['鈴木茂', '41歳', '男', 'カレーライス'],  
['佐藤和子', '29歳', '女', 'パスタ']
```

(2) 下記の辞書からDataFrameを定義してください。

```
dictionary = {1:['a1','a2'], 2:['b1','b2'] , 3:['c1','c2']}
```

DataFrame(DF)の確認

- CSVファイルからのDFへのデータ読み込み方法
- 読み込んだDFの確認方法
- DFにたいするデータ参照、抽出方法(インデックス値、条件)
- データ項目の型、要約統計量の確認
- 欠損値の扱い
- CSVファイルへの出力
- データ間の演算

csvファイルを読み込み、DataFrameとして定義します。

<csv形式データ読み込み>

```
pd.read_csv(filepath)
```

pd.read_csvのオプション

filepath=必須

入力ファイルのファイルパス

または、ファイル名を指定

header='infer'

入力ファイルの指定された行目をヘッダ行とする

デフォルトは0行目

ヘッダが不要な場合はNoneを指定

encoding=None

入力ファイルの文字コードを指定

デフォルトは'UTF-8'

windowsのファイルは'cp932'が主に使用される

その他のデータ形式でも読み込みが可能

・read_tsv() …タブ区切りデータの読み込み

・read_excel() …エクセルブックの読み込み

pathlibを使用し、ファイルパスを組み立てます。

ファイルパスの区切りがmacOSとWindowsでは異なり、どちらでも正しく動作させるために必要です。

<ライブラリのインポート>

```
from pathlib import Path
```

<notebookパスの取得>

```
nbpath = Path('.').resolve()
```

<パスの組み立て>

Notebookパスから相対パスで指定する

```
target_dir = nbpath.joinpath('./data/hogehoge.csv')
```

カレントディレクトリパスは
Path.cwd()で取得できる。

※jupyter notebookを起動した場所に依存する

例題3－1)

65

csvファイル(kakeibo.csv)を読み込み、DataFrame(df_kakeibo)を定義してください。

```
# csvの読み込み(現在のノートブックとcsvが同じディレクトリにある場合)
df_kakeibo = pd.read_csv('kakeibo.csv')
df_kakeibo
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費
0	2018/6	1001	46	未婚	7	47000	47134	NaN	3300	12400
1	2018/6	1002	31	こども1人	7	52000	48235	17642	3900	25300
2	2018/6	1003	40	こども2人	3	63000	54801	25356	8000	24600
3	2018/6	1004	22	既婚	7	62000	40996	10479	1900	27700
4	2018/6	1005	46	未婚	3	58000	35858	15811	6400	26100

⋮

先頭末尾の内容確認

66

DataFrame(DF)の先頭、末尾行の内容を確認します。

nに表示したい行数を設定します。空欄の場合5行(デフォルト値)を表示します。

<データの先頭から任意行を確認>

```
DF.head(n)
```

<データの末尾から任意行を確認>

```
DF.tail(n)
```

例題3－2)

67

(1) df_kakeiboの先頭から5行目までを表示してください。

```
# データフレームの上から5行(デフォルト)を表示  
df_kakeibo.head()
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費
0	18-Jun	1001	46	未婚	7	47000	47134	NaN	3300	12400
1	18-Jun	1002	31	こども1人	7	52000	48235	17642	3900	25300
2	18-Jun	1003	40	こども2人	3	63000	54801	25356	8000	24600
3	18-Jun	1004	22	既婚	7	62000	40996	10479	1900	27700
4	18-Jun	1005	46	未婚	3	58000	35858	15811	6400	26100

例題3－2)

68

(2) df_kakeiboの末尾から3行目までを表示してください。

```
# データフレームの末尾から3行を表示  
df_kakeibo.tail(3)
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費
196	18-Jul	1098	24	未婚	5	NaN	25231	13319	2700	32400
197	18-Jul	1099	40	未婚	2	42000	25007	10875	2100	24300
198	18-Jul	1100	32	こども2人	1	51000	42876	21957	12500	32400

index,column を指定した内容確認

69

DataFrame(DF)内のindex(行番号)やcolumn(列名)を指定し内容を確認します。

<インデックスを指定したDFの確認>

```
DF[start:end]
```

<columnを単一指定したDFの確認>

```
DF['column名']
```

※column名を1つ指定した場合、Seriesで表示

<イメージ>

index	col1	col2	col3	col4
0	[0,0]	[0,1]	[0,2]	[0,3]
1	[1,0]	[1,1]	[1,2]	[1,3]
2	[2,0]	[2,1]	[2,2]	[2,3]
3	[3,0]	[3,1]	[3,2]	[3,3]
4	[4,0]	[4,1]	[4,2]	[4,3]
5	[5,0]	[5,1]	[5,2]	[5,3]
...
n-1	[n-1,0]	[n-1,1]	[n-1,2]	[n-1,3]
n	[n,0]	[n,1]	[n,2]	[n,3]

df [2:5]

index	col1	col2	col3	col4
0	[0,0]	[0,1]	[0,2]	[0,3]
1	[1,0]	[1,1]	[1,2]	[1,3]
2	[2,0]	[2,1]	[2,2]	[2,3]
3	[3,0]	[3,1]	[3,2]	[3,3]
4	[4,0]	[4,1]	[4,2]	[4,3]
5	[5,0]	[5,1]	[5,2]	[5,3]
...
n-1	[n-1,0]	[n-1,1]	[n-1,2]	[n-1,3]
n	[n,0]	[n,1]	[n,2]	[n,3]

df ['col2']

例題3－3)

70

(1) df_kakeiboの2行目から4行目までを表示してください。

```
# インデックスを指定したDFの確認  
df_kakeibo[2:5]
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費
2	18-Jun	1003	40	こども2人	3	63000	54801	25356	8000	24600
3	18-Jun	1004	22	既婚	7	62000	40996	10479	1900	27700
4	18-Jun	1005	46	未婚	3	58000	35858	15811	6400	26100

例題3－3)

71

(2) df_kakeiboの'年齢'を表示してください。

```
# columnを単一指定したDFの確認  
df_kakeibo['年齢']
```

結果

```
0    46  
1    31  
2    40  
3    22  
4    46  
5    50  
6    47  
7    48
```

⋮

index,column を指定した内容確認

72

DataFrame(DF)内のindex(行番号)やcolumn(列名)を指定し内容を確認します。

<column名を複数指定したDFの確認>

```
DF[['column名1', 'column名2']]
```

※column名はlist型で渡す点に注意

<条件を満たすcolumn値を行に持つDFの確認>

```
DF[DF['column名1'] == 条件値]
```

<イメージ>

index	col1	col2	col3	col4
0	[0,0]	a	[0,2]	[0,3]
1	[1,0]	b	[1,2]	[1,3]
2	[2,0]	a	[2,2]	[2,3]
3	[3,0]	b	[3,2]	[3,3]
4	[4,0]	a	[4,2]	[4,3]
5	[5,0]	b	[5,2]	[5,3]
...
n-1	[n-1,0]	b	[n-1,2]	[n-1,3]
n	[n,0]	a	[n,2]	[n,3]

df ['col2','col3']

index	col1	col2	col3	col4
1	[1,0]	a	[1,2]	[1,3]
2	[2,0]	b	[2,2]	[2,3]
5	[5,0]	a	[5,2]	[5,3]
n-1	[n-1,0]	b	[n-1,2]	[n-1,3]
n	[n,0]	a	[n,2]	[n,3]

df [df ['col2']=='a']

条件に該当した行のみ表示される

参考:条件とブールインデックス

条件とブールインデックスを使ってデータの抽出ができます。

A:ブールインデックス指定

```
df_animal[[True,False,True,False,True,False]]
```

	動物	分類
0	キリン	哺乳類
2	ゾウ	哺乳類
4	ゴリラ	哺乳類

Trueの部分が抽出される



B:条件指定

```
df_animal['分類'] == '哺乳類'
```

```
0    True
1   False
2    True
3   False
4    True
5   False
Name: 分類, dtype: bool
```

ブールインデックスが出力される

A+B:条件指定(ブールインデックス)

```
df_animal[df_animal['分類'] == '哺乳類']
```

	動物	分類
0	キリン	哺乳類
2	ゾウ	哺乳類
4	ゴリラ	哺乳類

df_animal

	動物	分類
0	キリン	哺乳類
1	オウム	鳥類
2	ゾウ	哺乳類
3	カラス	鳥類
4	ゴリラ	哺乳類
5	ワシ	鳥類

例題3－4)

74

(1) df_kakeiboの'年齢'と'家族構成'を表示してください。

```
# column名を複数指定したDFの確認  
df_kakeibo[['年齢', '家族構成']]
```

結果

	年齢	家族構成
0	46	未婚
1	31	こども1人
2	40	こども2人
3	22	既婚
4	46	未婚
	⋮	

例題3－4)

75

(2) df_kakeiboの'年齢'が33のデータを先頭から5行表示してください。

```
# column名に設定された値が条件に合致するDFの確認  
df_kakeibo[df_kakeibo['年齢'] == 33].head()
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費
13	18-Jun	1014	33	既婚	1	45000	36748	12628	1300	9400
15	18-Jun	1016	33	こども3人	8	48000	53522	24961	13300	21400
30	18-Jun	1031	33	未婚	5	52000	34544	16627	1600	19700
32	18-Jun	1033	33	未婚	8	48000	35174	16802	700	22800
93	18-Jun	1094	33	既婚	7	60000	43652	18207	5900	21100

- (1) test_scores.csvを読み込み、DataFrame(df_test)を定義してください。
- (2) (1)で作成したdf_testから、'学年'が3のデータを先頭から10行表示してください。

index,column を指定した内容確認

77

DataFrame(DF)内のindex(行番号)やcolumn名(列名)を指定し内容を確認します。

<"単一の値"の確認>

```
DF.at[index, 'column名']
```

```
DF.iat[index, column番号]
```

<"範囲内の値"の確認>

```
DF.loc[index, 'column名']
```

```
DF.iloc[index, column番号]
```

<イメージ>

index	col1	col2	col3	col4
0	[0,0]	a	[0,2]	[0,3]
1	[1,0]	b	[1,2]	[1,3]
2	[2,0]	a	[2,2]	[2,3]
3	[3,0]	b	[3,2]	[3,3]
4	[4,0]	a	[4,2]	[4,3]
5	[5,0]	b	[5,2]	[5,3]
...

df.at[2, 'col4']

or

df.iat[2, 3]

index	col1	col2	col3	col4
0	[0,0]	a	[0,2]	[0,3]
1	[1,0]	b	[1,2]	[1,3]
2	[2,0]	a	[2,2]	[2,3]
3	[3,0]	b	[3,2]	[3,3]
4	[4,0]	a	[4,2]	[4,3]
5	[5,0]	b	[5,2]	[5,3]
...

df.loc[[2, 3, 4], ['col3', 'col4']]

or

df.iloc[2:5,2:4]

例題3－5)

78

(1) 行番号と列名/列番号を指定し、先頭行の'年齢'を表示してください。

```
#行番号と列名  
df_kakeibo.at[0, '年齢']
```

結果
46

```
#行番号と列番号  
df_kakeibo.iat[0, 2]
```

結果
46

例題3－5)

79

(2) 行番号と列名/列番号を指定し、先頭から2行の'年齢'を表示してください。

#行番号と列名

```
df_kakeibo.loc[0:1, '年齢']
```

結果

```
0    46
```

```
1    31
```

```
Name: 年齢, dtype: int64
```

#行番号と列番号

```
df_kakeibo.iloc[0:2, 2:3]
```

結果

	年齢
0	46
1	31

例題3－5)

80

(2) 行番号と列名/列番号を指定し、先頭から2行の'年齢'を表示してください。

```
#行番号と列番号(list指定)  
df_kakeibo.iloc[[0,1,],[2]]
```

結果

年齢	
0	46
1	31

条件を指定した内容確認

81

'()'や'&'などの演算子を組み合わせることで、複数条件に該当するデータの内容を確認します。
'and'や'or'では動かないため、'&'や'|'を使う必要があります。

<"and" 条件の指定>

```
DF[(DF['column名'] == 'AAA') & (DF['column名'] == 'BBB')]
```

<"or" 条件の指定>

```
DF[(DF['column名'] == 'AAA') | (DF['column名'] == 'BBB')]
```

<"not" 条件の指定>

```
DF[~(DF['column名'] == 'AAA')]
```

'()'や'&'などの演算子を組み合わせることで、複数条件に該当するデータの内容を確認します。
'and'や'or'では動かないため、'&'や'|'を使う必要があります。

<"in" 条件の指定>

```
DF[(DF['column名'].isin(['抽出したい要素1', '抽出したい要素2']))]
```

例題3－6)

83

(1) '地方ID'が1かつ'家賃'が50000以上のデータを表示してください。

```
df_kakeibo[(df_kakeibo['地方ID'] == 1) &
            (df_kakeibo['家賃'] >= 50000)]
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費
21	18-Jun	1022	36	未婚	1	62000	28675	10354	1900	16700
31	18-Jun	1032	31	こども2人	1	53000	39971	18946	6300	22500
35	18-Jun	1036	48	未婚	1	56000	36659	14367	200	22800
54	18-Jun	1055	39	こども1人	1	60000	46706	19533	5500	32800
63	18-Jun	1064	40	未婚	1	54000	33284	13544	3400	12600

⋮

例題3－6)

84

(2) '年齢'が25以下または'年齢'が45以上のデータを表示してください。

```
df_kakeibo[(df_kakeibo['年齢'] <= 25) |  
            (df_kakeibo['年齢'] >= 45)]
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費
0	18-Jun	1001	46	未婚	7	47000	47134	NaN	3300	12400
3	18-Jun	1004	22	既婚	7	62000	40996	10479	1900	27700
4	18-Jun	1005	46	未婚	3	58000	35858	15811	6400	26100
5	18-Jun	1006	50	既婚	6	49000	47784	12785	2200	10100
6	18-Jun	1007	47	こども2人	2	49000	54272	20066	11700	15500

⋮

```
df_kakeibo[df_kakeibo['家族構成'].isin(['子ども1人', '既婚'])]
```

年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費	
1	18-Jun	1002	31	こども1人	7	52000.0	48235.0	17642.0	3900.0	25300
3	18-Jun	1004	22	既婚	7	62000.0	40996.0	10479.0	1900.0	27700
5	18-Jun	1006	50	既婚	6	49000.0	47784.0	12785.0	2200.0	10100
7	18-Jun	1008	48	既婚	3	49000.0	40166.0	16850.0	4000.0	10800
12	18-Jun	1013	30	こども1人	6	54000.0	46061.0	15342.0	6100.0	30700
13	18-Jun	1014	33	既婚	1	45000.0	36748.0	12628.0	1300.0	9400
				⋮						

(応用)条件を指定した内容確認

86

where関数で元のデータと同じ長さの条件に該当するデータを確認します。また、otherに値を指定し、条件に該当しなかったデータをotherで埋めることができます。

<"where" を使用した確認>

```
【DF or Series】.where(cond)
```

whereのオプション

cond=必須

条件を設定

(データと同じ長さのSeries)

other=None

上記条件にて該当しない場合のパディング値

デフォルトはNaN

例題3－7)【応用】

87

whereを使用し、'年齢'に対して40未満はそのまま、40以上はNaNに置き換えて表示してください。

```
df_kakeibo['年齢'].where(df_kakeibo['年齢'] < 40)
```

結果

0	NaN
1	31.0
2	NaN
3	22.0
4	NaN
5	NaN
6	NaN
7	NaN

(応用)条件を指定した内容確認

88

queryを利用することで、複数条件を指定した確認をよりシンプルに記載することができます。
また、指定した行のみを抽出することができます。

<"query" を使用した確認>

```
DF.query(expr)
```

queryのオプション

expr(str)=必須

適用したい条件を文字列型で記載

<例>

・A列が"aa" かつ B列が10以上

```
DF.query('A == "aa" and B >= 10')
```

・行番号が0～5 または B列が10以上

```
DF.query('index in [0, 1, 2, 3, 4, 5] or B >= 10')
```

例題3－8)【応用】

89

(1) queryを使用し、'地方ID'が1かつ'家賃'が50000以上のデータを表示してください。

```
df_kakeibo.query('地方ID == 1 and 家賃 >= 50000')
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費
21	18-Jun	1022	36	未婚	1	62000	28675	10354	1900	16700
31	18-Jun	1032	31	こども2人	1	53000	39971	18946	6300	22500
35	18-Jun	1036	48	未婚	1	56000	36659	14367	200	22800
54	18-Jun	1055	39	こども1人	1	60000	46706	19533	5500	32800
63	18-Jun	1064	40	未婚	1	54000	33284	13544	3400	12600

⋮

例題3－8)【応用】

90

(2) queryを使用し、'年齢'が25以下または45以上のデータを表示してください。

```
df_kakeibo.query('年齢 <= 25 or 年齢 >= 45')
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費
0	18-Jun	1001	46	未婚	7	47000	47134	NaN	3300	12400
3	18-Jun	1004	22	既婚	7	62000	40996	10479	1900	27700
4	18-Jun	1005	46	未婚	3	58000	35858	15811	6400	26100
5	18-Jun	1006	50	既婚	6	49000	47784	12785	2200	10100
6	18-Jun	1007	47	こども2人	2	49000	54272	20066	11700	15500

⋮

- (1) 演習3-1(1)で作成したdf_testの行番号と列名を指定し、'国語'の点数を先頭から3行表示してください。
- (2) '学年'が3かつ'数学'の点数が80以上のデータを表示してください。
- (3) queryを使用し、'国語'または'英語'の点数が80以上のデータを表示してください。

項目の型、要約統計を確認-1

92

項目の型、要約統計を確認します。

<DF項目の型を確認>

```
DF.dtypes
```

- ・文字列項目はobject型、数値項目をint型、日付項目はdatetime型で表示

<DFの列数・行数を確認>

```
DF.shape
```

項目の型、要約統計を確認-2

93

項目の型、要約統計を確認します。

<DFの要約統計を表示>

```
DF.describe()
```

- ・DF内各項目の件数、平均、標準偏差、最大最小、四分位を表示

describeのオプション

include=None

表示する項目を指定、デフォルトは数値項目のみ

'all'のとき、DF内のすべての項目の要約統計を表示

'object'のとき、DF内の文字列項目の要約統計を表示

※統計量countとuniqueは欠損値を含まないことに注意

例題3－9)

94

(1) 全ての項目について型を表示してください。

```
# DF項目の型を表示  
df_kakeibo.dtypes
```

結果

```
年月      object  
会員ID    int64  
年齢      int64  
家族構成  object  
地方ID    int64  
家賃      int64  
食費      int64  
公共料金  int64  
医療費    int64  
雑費      int64  
dtype: object
```

例題3－9)

95

(2) DataFrameの列数・行数を表示してください。

```
# DFの列数・行数を表示  
df_kakeibo.shape
```

結果

(199, 10)

例題3－9)

96

(3) 全ての項目について基礎統計を表示してください。

```
# DFの基礎統計を表示  
df_kakeibo.describe(include = 'all')
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費
count	199	199.000000	199.000000	199	199.000000	197.000000	198.000000	197.000000	198.000000	199.000000
unique	2	NaN	NaN	5	NaN	NaN	NaN	NaN	NaN	NaN
top	18-Jul	NaN	NaN	未婚	NaN	NaN	NaN	NaN	NaN	NaN
freq	100	NaN	NaN	92	NaN	NaN	NaN	NaN	NaN	NaN
mean	NaN	1050.251256	36.331658	NaN	4.658291	51441.624365	40272.772727	16870.730964	4753.535354	23025.628141
std	NaN	28.796345	7.571915	NaN	2.310236	5250.159706	8935.540243	4676.859149	4162.213710	9481.182665
min	NaN	1001.000000	22.000000	NaN	1.000000	41000.000000	21380.000000	9007.000000	0.000000	5600.000000
25%	NaN	1025.500000	30.500000	NaN	2.000000	48000.000000	33131.000000	13438.000000	1425.000000	15700.000000
50%	NaN	1050.000000	37.000000	NaN	5.000000	51000.000000	40257.000000	15913.000000	3350.000000	22400.000000
75%	NaN	1075.000000	42.000000	NaN	7.000000	55000.000000	47110.750000	19782.000000	7475.000000	30250.000000
max	NaN	1100.000000	50.000000	NaN	8.000000	67000.000000	60316.000000	29113.000000	16800.000000	55700.000000

csvファイルを出力する

97

DataFrameをcsvファイルとして出力します。

<DFをcsvファイルへ出力>

```
DF.to_csv(path)
```

to_csvのオプション

path=None

出力ファイルのファイルパス、またはファイル名を指定
省略した場合は文字列を返す

encoding=None

出力ファイルの文字コードを指定
デフォルトは'UTF-8'

index=True

出力ファイルインデックス有無を指定
デフォルトはTrue

例題3－10)

98

全データと、'年齢'が30より大きいデータをそれぞれcsvファイルへ出力してください。

```
# 全データをCSVファイルへ出力
df_kakeibo.to_csv('df_kakeibo_all.csv')

# 条件に該当するデータをcsvファイルへ出力
df_kakeibo[df_kakeibo['年齢'] > 30].to_csv('df_kakeibo_age.csv')
```

結果
出力されたcsvファイルを確認

欠損値について、有無の確認と置換を行います。
Pythonの欠損値は、NaNで表示されます。

<欠損値を含んでいるか確認(Trueが欠損)>

```
DF.isnull()
```

<欠損値を置換>

```
DF.fillna()
```

fillnaのオプション

value=必須

欠損値から置換する指定の値を設定

inplace=False

Trueのとき、元のデータを変更する

Falseのとき、新規のオブジェクトを返す

例題3－11)

100

(1) df_kakeiboが欠損値を含んでいるか確認してください。

```
# 欠損値を含んでいるか確認  
df_kakeibo.isnull().sum()
```

結果

```
年月          0  
会員ID        0  
年齢          0  
家族構成      0  
地方ID        0  
家賃          2  
食費          1  
公共料金      2  
医療費        1  
雑費          0  
dtype: int64      ⋮
```

```
# (おまけ)欠損値を含む行のみ表示  
df_kakeibo[df_kakeibo.isnull().any(axis=1)]
```

※anyは条件に合致するものが少なくとも1つあればTrueを返す

例題3－11)

101

(2) 欠損値を0で置換し、末尾を表示してください。

```
# 欠損値を一律0で置換し末尾を表示  
df_kakeibo.fillna(0).tail()
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費
194	18-Jul	1096	27	こども3人	4	0	44573	24321	14700	18600
195	18-Jul	1097	43	未婚	8	46000	46510	0	2600	14100
196	18-Jul	1098	24	未婚	5	0	25231	13319	2700	32400
197	18-Jul	1099	40	未婚	2	42000	25007	10875	2100	24300
198	18-Jul	1100	32	こども2人	1	51000	42876	21957	12500	32400

例題3－11)

102

(3) 欠損値を0で置換し、元のデータを更新してください。

```
# 欠損値を一律0で置換
df_kakeibo.fillna(0, inplace=True)
# 末尾を表示
df_kakeibo.tail()
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費
194	18-Jul	1096	27	こども3人	4	0	44573	24321	14700	18600
195	18-Jul	1097	43	未婚	8	46000	46510	0	2600	14100
196	18-Jul	1098	24	未婚	5	0	25231	13319	2700	32400
197	18-Jul	1099	40	未婚	2	42000	25007	10875	2100	24300
198	18-Jul	1100	32	こども2人	1	51000	42876	21957	12500	32400

四則演算(演算子、関数)

DataFrame(DF)の列間の演算を行います。

<演算子による計算(列と数値)>

```
【DF or Series】 * 数値
```

<演算子による計算(列と列)>

```
【DF or Series】 + 【DF or Series】
```

※減算、除算、も同様に演算可能

文字列項目も同様に演算可能だが、加算は文字列同士の結合、乗算は数値分の繰り返しとなる。減算、除算は不可。

<関数による加算>

```
DF.sum()
```

sumのオプション

axis=0

行単位(1)または列単位(0)の加算を選択可能

デフォルトは0

四則演算(演算子、関数)

104

NaN(欠損値)と演算子による計算をすると、結果はすべてNaNになります。
一方で関数による計算では、欠損値を除く場合があります、使い分けが必要になります。

<DFの作成>

```
DF = pd.DataFrame({'A':[1,2], 'B':[1,np.NaN]})
```

<演算子による足し算>

```
DF['C'] = DF['A'] + DF['B']
```

<関数による足し算>

```
DF['D'] = DF[['A','B']].sum(axis=1)
```

<結果>

	A	B	C	D
0	1	1	2	2
1	2	NaN	NaN	2

例題3－12)

105

(1) '雑費'について、消費税を8%から10%に変更した値を表示してください。

```
# 演算子による計算(列と数値)  
df_kakeibo['雑費'] * 1.10 / 1.08
```

結果

```
0    12629.629630  
1    25768.518519  
2    25055.555556  
3    28212.962963  
4    26583.333333  
5    10287.037037  
6    15787.037037  
7    11000.000000
```

⋮

例題3－12)

106

(2) '家賃'に'公共料金'を加算した値を表示してください。

```
# 演算子による計算(列と列)
df_kakeibo['家賃'] + df_kakeibo['公共料金']
```

結果

```
0    58231
1    69642
2    88356
3    72479
4    73811
5    61785
6    69066
7    65850
```

⋮

例題3－12)

107

(3) '家賃'と'食費'を行単位に加算した値を表示してください。

```
# 関数による加算(行単位)  
df_kakeibo[['家賃', '食費']].sum(axis=1)
```

結果

```
0    94134  
1   100235  
2   117801  
3   102996  
4    93858  
5    96784  
6   103272  
7    89166  
  
⋮
```

例題3－12)

108

(4) '家賃'と'食費'を列単位に加算した値を表示してください。

```
# 関数による加算(列単位)
df_kakeibo[['家賃', '食費']].sum(axis=0)
```

結果

```
家賃    10239000
食費     8007072
dtype: int64
```

- (1) 演習3-1(1)で作成したdf_testの数値項目について、要約統計を表示してください。
- (2) 欠損値を含む行を表示してください。
- (3) '国語'、'数学'、'英語'の3教科の合計点数(行単位)を表示してください。

データの加工

- データ加工の必要性
- データ列の追加
- データ行・列の削除
- データの重複排除
- データのソート
- 欠損値の削除
- データの結合
- データ項目の型変換
- 日付データの扱い
- 関数適用

データ加工はなぜ必要？

112

データ加工とは、分析を行う際の前処理のことです。

本格的な分析を始めるためには加工をする必要があります。

たとえば、売上のレシートデータ(POS)を分析するとき、POSデータは蓄積に適した形のデータであるため、分析の目的に合わせた形への加工が必要となります。

日時	会員ID	品名	単価	個数	店舗名
2018/7/1	1001	ハンバーガー	200円	2	京都店
2018/7/1	1005	ポテト	250円	1	東京店
2018/7/1	1045	ハンバーガー	200円	3	千葉店



日時	品名	単価	個数	合計
2018/7/1	ハンバーガー	200円	5	1,000円
2018/7/1	ポテト	250円	1	250円

新しい列を追加する

113

既存データに新しい列を追加します。リストとSeriesの長さはDFの行数と同じでないといけません。値を代入する場合、すべての行が同じ値となります。

<新しい列を追加>

```
DF['新規のcolumn名'] = (Series or リスト or 値)
```

例題4－1)

114

(1) '家賃'、'食費'と'公共料金'を加算した新しい列'生活費'を作成してください。

```
df_kakeibo['生活費'] = df_kakeibo[['家賃', '食費',  
                                   '公共料金']].sum(axis=1)  
df_kakeibo
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費	生活費
0	18-Jun	1001	46	未婚	7	47000	47134	0	3300	12400	94134
1	18-Jun	1002	31	こども1人	7	52000	48235	17642	3900	25300	117877
2	18-Jun	1003	40	こども2人	3	63000	54801	25356	8000	24600	143157
3	18-Jun	1004	22	既婚	7	62000	40996	10479	1900	27700	113475
4	18-Jun	1005	46	未婚	3	58000	35858	15811	6400	26100	109669

⋮

例題4－1)

115

(2) '生活費'が100,000以上はそのまま、100,000未満は欠損に書き換えた列'生活費2'を作成してください。

```
df_kakeibo['生活費2'] = df_kakeibo['生活費'].where(  
    df_kakeibo['生活費'] >= 100000)  
df_kakeibo
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費	生活費	生活費2
0	2018/6	1001	46	未婚	7	47000	47134	0	3300	12400	94134	NaN
1	2018/6	1002	31	こども1人	7	52000	48235	17642	3900	25300	117877	117877
2	2018/6	1003	40	こども2人	3	63000	54801	25356	8000	24600	143157	143157
3	2018/6	1004	22	既婚	7	62000	40996	10479	1900	27700	113475	113475
4	2018/6	1005	46	未婚	3	58000	35858	15811	6400	26100	109669	109669

⋮

DataFrameの指定された行、列を削除します。

<行、列の削除>

```
DF.drop(index, columns)
```

dropのオプション

index=None

削除するindexを指定

複数の場合リストで指定

columns=None

削除するcolumnsを指定

複数の場合リストで指定

inplace=False

Trueのとき、元のデータを変更する

デフォルトはFalse

例題4－2)

117

1行目と'年齢'、'家族構成'を削除してください。

```
df_kakeibo.drop(index=0, columns=['年齢', '家族構成'])
```

結果

	年月	会員ID	地方ID	家賃	食費	公共料金	医療費	雑費	生活費	生活費2
1	18-Jun	1002	7	52000	48235	17642	3900	25300	117877	117877
2	18-Jun	1003	3	63000	54801	25356	8000	24600	143157	143157
3	18-Jun	1004	7	62000	40996	10479	1900	27700	113475	113475
4	18-Jun	1005	3	58000	35858	15811	6400	26100	109669	109669
5	18-Jun	1006	6	49000	47784	12785	2200	10100	109569	109569

⋮

(応用)行,列の欠損値削除

118

DataFrameの欠損値のある行、列を削除します。

<欠損値の削除>

```
DF.dropna()
```

dropnaのオプション

axis=0

行単位(0)または列単位(1)の削除を選択可能

デフォルトは0

how='any'

'any'のとき、1つでもNaNがあれば削除

'all'のとき、すべてがNaNの行、列を削除

デフォルトは'any'

inplace=False

Trueのとき、元のデータを変更

デフォルトはFalse

例題4－3)【応用】

119

(1) 欠損値を含む行を削除したdf_dropna1を作成してください。

```
# 欠損値を含む行の削除
df_dropna1 = df_kakeibo.dropna()
df_dropna1
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費	生活費	生活費2
1	18-Jun	1002	31	こども1人	7	52000	48235	17642	3900	25300	117877	117877
2	18-Jun	1003	40	こども2人	3	63000	54801	25356	8000	24600	143157	143157
3	18-Jun	1004	22	既婚	7	62000	40996	10479	1900	27700	113475	113475
4	18-Jun	1005	46	未婚	3	58000	35858	15811	6400	26100	109669	109669
5	18-Jun	1006	50	既婚	6	49000	47784	12785	2200	10100	109569	109569

⋮

例題4－3)【応用】

120

(2) 欠損値を含む列'生活費2'を削除したdf_dropna2を作成してください。

```
# 欠損値を含む列の削除
df_dropna2 = df_kakeibo.dropna(axis=1)
df_dropna2
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費	生活費
0	2018/6	1001	46	未婚	7	47000	47134	0	3300	12400	94134
1	2018/6	1002	31	こども1人	7	52000	48235	17642	3900	25300	117877
2	2018/6	1003	40	こども2人	3	63000	54801	25356	8000	24600	143157
3	2018/6	1004	22	既婚	7	62000	40996	10479	1900	27700	113475
4	2018/6	1005	46	未婚	3	58000	35858	15811	6400	26100	109669

⋮

- (1) 演習3-1(1)で作成したdf_testにおいて、5教科の点数を合計した列'5教科'を作成してください。
- (2) 欠損値を含む行を削除したdf_test1を作成してください。

DataFrameの重複を削除します。

<重複の削除>

```
DF.drop_duplicates()
```

drop_duplicatesのオプション

subset=None

指定された列のみで重複を判断

Noneの場合、すべての列で判断

inplace=False

Trueのとき、元のデータを変更

デフォルトはFalse

keep='first'

'first'のとき、重複のうち最初の行を残す

'last'のとき、重複のうち最後の行を残す

False(引用符不要)のとき、重複した行をすべて削除

デフォルトは'first'

例題4－4)

123

'家族構成'から重複を削除して、家族構成表を作成してください。

```
df_kazoku = df_kakeibo['家族構成'].drop_duplicates()  
df_kazoku
```

結果

```
0    未婚  
1    こども1人  
2    こども2人  
3    既婚  
15   こども3人  
Name: 家族構成, dtype: object
```

```
# (おまけ)uniqueメソッドを使った重複排除リスト取得  
df_kakeibo['家族構成'].unique()
```

DataFrameの列の順番を入れ替えます。

<列の入れ替え>

```
DF[[入れ替え後のcolumns]]
```

columnsには'列名1','列名2','列名3'...のように、並び替え後の列の順番に記述します。

例):

<DF>

	A	B	C
1	a1	b1	c1
2	a2	b2	c2

<DF[['A','C','B']]>

	A	C	B
1	a1	c1	b1
2	a2	c2	b2

例題4－5)

125

'年齢'と'地方ID'を入れ替えて表示してください。

```
df_kakeibo[['年月', '会員ID', '地方ID', '家族構成', '年齢',  
            '家賃', '食費', '公共料金', '医療費', '雑費', '生活費']]
```

結果

	年月	会員ID	地方ID	家族構成	年齢	家賃	食費	公共料金	医療費	雑費	生活費
0	18-Jun	1001	7	未婚	46	47000	47134	0	3300	12400	94134
1	18-Jun	1002	7	こども1人	31	52000	48235	17642	3900	25300	117877
2	18-Jun	1003	3	こども2人	40	63000	54801	25356	8000	24600	143157
3	18-Jun	1004	7	既婚	22	62000	40996	10479	1900	27700	113475
4	18-Jun	1005	3	未婚	46	58000	35858	15811	6400	26100	109669

⋮

DataFrameを指定された列の値で並び替えます。

<行のソート>

```
DF.sort_values()
```

sort_valuesのオプション

by=必須

ソートする値の行、列を指定

複数の場合リストで指定

ascending=True

Trueのとき、昇順にソート

Falseのとき、降順にソート

デフォルトはTrue

inplace=False

Trueのとき、元のデータを変更

デフォルトはFalse

例題4－6)

127

'年齢'が降順になるように並び替えて表示してください。

```
df_kakeibo.sort_values(by='年齢', ascending=False)
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費	生活費	生活費2
117	18-Jul	1019	50	未婚	6	43000	30604	14229	800	21900	87833	NaN
5	18-Jun	1006	50	既婚	6	49000	47784	12785	2200	10100	109569	109569
18	18-Jun	1019	50	未婚	6	43000	36512	15308	1400	26200	94820	NaN
104	18-Jul	1006	50	既婚	6	49000	36558	13565	400	29800	99123	NaN
132	18-Jul	1034	50	未婚	4	53000	41032	11531	800	13000	105563	105563

⋮

データ加工の注意点(SettingWithCopyWarning)

128

DataFrameから[]や.loc[]などの操作を複数行ったオブジェクトに対して、代入操作を行うとSettingWithCopyWarningと呼ばれる警告が発生する場合があります。データ加工に失敗する可能性があるため、対処が必要な警告になります。

<SettingWithCopyWarningが発生する例>

```
df = pd.DataFrame({'A':[1, '2'], 'B':[3,4]}) #DF作成  
  
df[df['A'] == 1]['A'] = 0 #警告が発生する。  
df
```

<SettingWithCopyWarningへの対応>

[]や、.loc[]などを複数使った表現をやめ、1つのメソッドを使って表現します。

```
df = pd.DataFrame({'A':[1, '2'], 'B':[3,4]}) #DF作成  
  
df.loc[df['A'] == 1, 'A'] = 0 #警告が発生しない。  
df
```

データ加工の注意点(SettingWithCopyWarning)

129

<原因>

DataFrameから複数の抽出操作を行った時、元のDataFrameが参照される場合と、データフレームのコピーが作成される場合があります、後者の時に問題が発生します。


<例>

```
df = pd.DataFrame({'A':[1,2,3,4], 'B':['a','a','b','c']})
```

```
df[df['B'] == 'a']['A'] = 1  
df
```

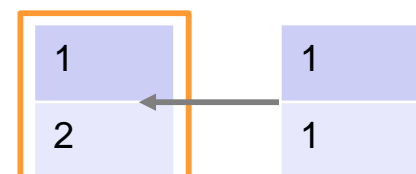
<元のDataFrameが参照される場合>

index	A	B
0	1	a
1	2	a
2	3	b
3	4	c



<元のDataFrameのコピーができる場合>

index	A	B
0	1	a
1	2	a
2	3	b
3	4	c

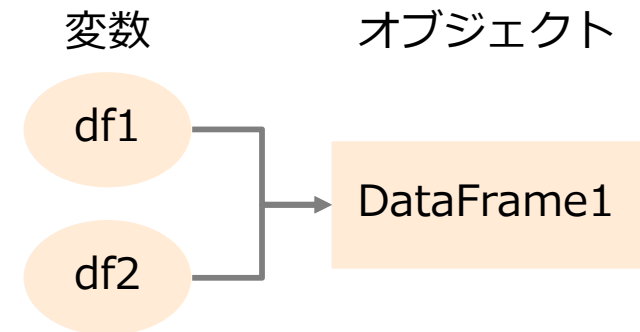


DataFrameをコピーして新しいDataFrameとして利用したい場合があります。この時、変数はオブジェクトに対する参照を保持するだけのため、変数を別の変数に渡しても意味がありません。そのため、copyメソッドを使い明示的にDataFrameオブジェクトを複製する必要があります。

<copyを使わない例>

```
# DFを作成
df1 = pd.DataFrame({'A':[1,2], 'B':[3,4]})
df2 = df1 # 同じオブジェクトを参照

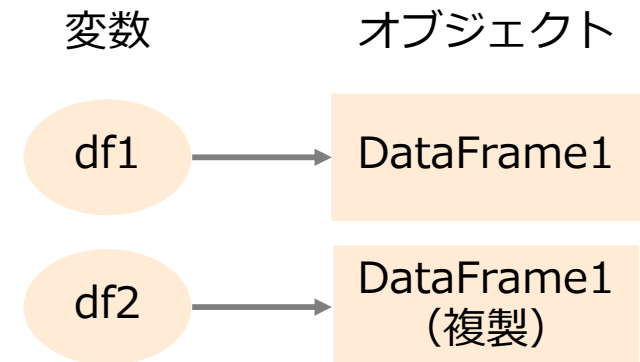
df2['B'] = [5,6] # df2のみ変更したつもり
df1 # df1が変わってしまっている
```



<copyを使う例>

```
# DFを作成
df1 = pd.DataFrame({'A':[1,2], 'B':[3,4]})
df2 = df1.copy() # 新しいオブジェクトを参照

df2['B'] = [5,6] # df2のみ変更
df1 # df1は変更されていない
```



- (1) 演習4－1(2)で作成したdf_test1において、'5教科'の列が'国語'の前になるよう入れ替えたdf_test2を作成してください。
- (2) df_test2を用いて、'5教科'の点数が高い順に並び替えたdf_test3を作成してください。
- (3) df_test3を用いて、'性別'の1を男、2を女にしたdf_test4を作成してください。
(df_test3を変更しないようにしてください。)
- (4) df_test4から'学校ID','性別'列を抽出し、重複を削除した結果を確認してください。

データの結合(concat)

132

複数のDataFrame, Seriesをindex, columnsを元に結合します。

<データの結合>

```
pd.concat(objs)
```

pd.concatのオプション

objs=必須

結合するDFをリストで指定

axis=0

縦(0)または横(1)の結合を選択可能

デフォルトは0

join='outer'

'outer'のとき、全データを結合

データがない行、列はNaNになる

'inner'のとき、両方のDFに存在する

データを結合

データの結合(concat)

133

<例1:縦に結合>

<df1>

	c1	c2	c3
i1	A	B	C
i2	D	E	F

<df2>

	c1	c2	c4
i1	a	b	c
i2	d	e	f

<pd.concat([df1,df2],axis=0)>

	c1	c2	c3	c4
i1	A	B	C	NaN
i2	D	E	F	NaN
i1	a	b	NaN	c
i2	d	e	NaN	f

<例2:横に結合>

<df1>

	c1	c2
i1	A	B
i2	C	D
i3	E	F

<df2>

	c1	c2
i1	a	b
i2	c	d
i4	e	f

<pd.concat([df1,df2],axis=1)>

	c1	c2	c1	c2
i1	A	B	a	b
i2	C	D	c	d
i3	E	F	NaN	NaN
i4	NaN	NaN	e	f

新しい会員の情報が追加されました。縦に結合してdf_kakeibo1を作成してください。

```
#データの作成
data = [['18-Jul', 1101, 42, 'こども2人', 3, 75000,
        51437, 23192, 1300, 54000],
        ['18-Jul', 1102, 23, '未婚', 3, 61000,
        34194, 12106, 0, 9021]]
columns = ['年月', '会員ID', '年齢', '家族構成',
           '地方ID', '家賃', '食費', '公共料金', '医療費', '雑費']
df_new_kakeibo = pd.DataFrame(data=data, columns=columns)

df_kakeibo1 = pd.concat([df_kakeibo, df_new_kakeibo])
df_kakeibo1.tail()
```

例題4－7)

135

結果

	会員ID	公共料金	医療費	地方ID	家族構成	家賃	年月	年齢	生活費	生活費2	雑費	食費
196	1098	13319	2700	5	未婚	0	18-Jul	24	38550	NaN	32400	25231
197	1099	10875	2100	2	未婚	42000	18-Jul	40	77882	NaN	24300	25007
198	1100	21957	12500	1	こども2人	51000	18-Jul	32	115833	115833	32400	42876
0	1101	23192	1300	3	こども2人	75000	18-Jul	42	NaN	NaN	54000	51437
1	1102	12106	0	3	未婚	61000	18-Jul	23	NaN	NaN	9021	34194

⋮

データの結合(merge)

136

2つのDataFrameを指定された列の値を元に結合します。

<データの結合>

```
pd.merge(left, right, on)
```

pd.mergeのオプション

left=必須

結合するDFを指定

right=必須

結合するDFを指定

on=None

結合キー列を指定

複数の場合はリストで指定

how='inner'

'outer'のとき、全データを結合

'inner'のとき、両方のDFに存在するデータを結合

'left'のとき、leftに存在する全データを結合

'right'のとき、rightに存在する全データを結合

デフォルトは'inner'

データの結合方法(merge)

137

<入力データ>

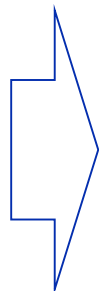
<df1>

	c1	c2
i1	A	1
i2	C	2

<df2>

	c1	c3
i1	A	3
i3	D	4

結合



<pd.merge(left=df1,right=df2,on='c1')>

<how=inner>

	c1	c2	c3
i1	A	1	3

<how=outer>

	c1	c2	c3
i1	A	1	3
i2	C	2	NaN
i3	D	NaN	4

<how=left>

	c1	c2	c3
i1	A	1	3
i2	C	2	NaN

<how=right>

	c1	c2	c3
i1	A	1	3
i3	D	NaN	4

例題4－8)

138

地方IDの対応表があります。これを利用して'地方ID'をキーに'地方名称'を追加したdf_kakeibo2を作成してください。

```
df_dis_name = pd.read_csv('district.csv')

df_kakeibo2 = pd.merge(df_dis_name, df_kakeibo, on='地方ID')
df_kakeibo2
```

結果

	地方ID	地方名称	年月	会員ID	年齢	家族構成	家賃	食費	公共料金	医療費	雑費	生活費	生活費2
0	1	北海道	18-Jun	1009	41	未婚	46000	0	15216	200	20700	61216	NaN
1	1	北海道	18-Jun	1014	33	既婚	45000	36748	12628	1300	9400	94376	NaN
2	1	北海道	18-Jun	1022	36	未婚	62000	28675	10354	1900	16700	101029	101029
3	1	北海道	18-Jun	1030	40	未婚	47000	23370	13560	300	34200	83930	NaN
4	1	北海道	18-Jun	1032	31	こども2人	53000	39971	18946	6300	22500	111917	111917

⋮

(応用)データの結合(merge)

マージのキーとなる列の名前が違う場合、別のオプションを使います。

または、列名を変更することもできます。

<データの結合>

```
pd.merge(left, right, left_on, right_on)
```

pd.mergeのオプション

left_on=None

左の結合キー列名

right_on=None

右の結合キー列名

<列名の変更>

```
DF.rename(index={"変更前": "変更後"}, columns={...})
```

例題4－9)【応用】

140

(1) df_dis_nameの'地方ID'を'地区ID'に変更したdf_dis_name2を作成してください。

```
df_dis_name2 = df_dis_name.rename(columns={'地方ID': '地区ID'})
df_dis_name2
```

結果

	地区ID	地方名称
0	1	北海道
1	2	東北
2	3	関東
3	4	中部
4	5	近畿
5	6	中国
6	7	四国
7	8	九州

例題4－9)【応用】

141

(2) '地方ID'と'地区ID'をキーにdf_dis_name2を追加したdf_mergeを作成してください。

```
df_merge = pd.merge(df_kakeibo, df_dis_name2,  
                    left_on='地方ID', right_on='地区ID')  
df_merge
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費	生活費	生活費2	地区ID	地方名称
0	18-Jun	1001	46	未婚	7	47000	47134	0	3300	12400	94134	NaN	7	四国
1	18-Jun	1002	31	こども1人	7	52000	48235	17642	3900	25300	117877	117877	7	四国
2	18-Jun	1004	22	既婚	7	62000	40996	10479	1900	27700	113475	113475	7	四国
3	18-Jun	1018	24	未婚	7	53000	26520	13370	2000	5800	92890	NaN	7	四国
4	18-Jun	1020	38	こども1人	7	44000	41545	16031	6600	19000	101576	101576	7	四国

⋮

- (1) test_scores_v2.csvを読み込み、演習4－2(3)で作成したdf_test4と縦に結合したdf_test5を作成してください。
- (2) school.csvを読み込み、df_test5を用いて、'学校ID'をキーに'学校名'を追加したdf_test6を作成してください。
- (3) subject.csvを読み込み、df_test6を用いて、'好きな教科'の数値を対応する'教科名'に置き換えたdf_test7を作成してください。
作成したdf_test7をcsvに出力してください。
- (4) df_test7をcsvに出力してください。

(応用)項目の型変換

異なる型同士の演算を行う際に、
項目の型をastype関数を利用し指定の型へ変換します。

<astypeを利用した型変換>

```
【DF or Series】.astype(dtype)
```

astypeのオプション

dtype = 必須

変換したい型(*)を指定、データ内項目をすべて変換

項目ごとに変換させたい場合は、以下のようにディクショナリ型で指定

{'column名1':変換したい型, 'column名2':変換したい型, ...}

(*)指定できる型の例

int : 符号あり整数型

bool : True または False

float : 浮動小数点数

str(※) : 文字列

※これを適用した場合、NaN(欠損値)も'nan'文字列となり、isnullで抽出できません。

例題4－10)【応用】

144

(1) '年齢'を整数型から文字列に型変換してください。

```
print('-----before-----\n', df_kakeibo.dtypes)

#型変換
df_kakeibo = df_kakeibo.astype({'年齢':str})

print('-----after-----\n', df_kakeibo.dtypes)
```

```
結果
-----before-----
...
年齢      int64
...
-----after-----
...
年齢      object
...
```

例題4－10)【応用】

145

(2) '年齢'と'家族構成'について(_)をはさんで結合し、新規変数を作成してください。
また、'年齢'を文字列から整数型に戻してください。

```
df_kakeibo['年齢_家族'] = df_kakeibo['年齢']  
                                + '_' + df_kakeibo['家族構成']  
df_kakeibo.head()
```

結果

	年月	会員ID	年齢	家族構成	地方ID	家賃	食費	公共料金	医療費	雑費	生活費	生活費2	年齢_家族
0	2018/6	1001	46	未婚	7	47000	47134	0	3300	12400	94134	NaN	46_未婚
1	2018/6	1002	31	こども1人	7	52000	48235	17642	3900	25300	117877	117877	31_こども1人
2	2018/6	1003	40	こども2人	3	63000	54801	25356	8000	24600	143157	143157	40_こども2人
3	2018/6	1004	22	既婚	7	62000	40996	10479	1900	27700	113475	113475	22_既婚
4	2018/6	1005	46	未婚	3	58000	35858	15811	6400	26100	109669	109669	46_未婚

#型変換した年齢を元に戻す

```
df_kakeibo = df_kakeibo.astype({'年齢':int}, copy=True)
```

csvからpd.read_csvで読み込んだ日付項目はデフォルトでobject型(str型)となっているため、datetime型に変換する必要があります。

datetime型にすることで年月日の抽出ができ、月毎の集計等が可能となります。

<to_datetimeを利用した型変換>

```
# 日付カラムobject型からdatetime型への変換
df[<日付カラム>] = pd.to_datetime(df[<日付カラム>])

# 月の抽出
Df['month'] = df.dt.strftime('%m')
```

pd.to_datetimeのオプション

format

読み込むカラムが標準的な書式でない場合、指定する。

書式はPythonドキュメント参照

<https://docs.python.org/ja/3.7/library/datetime.html>

例題4－11)

147

(1)uriage.csvを読み込み、日付カラムをdatetime型に変換してください。

```
df_uriage = pd.read_csv('uriage.csv')  
# データ型の確認  
df_uriage.dtypes
```

```
日付 object  
ハンバーガー売上件数 int64  
...略...  
dtype: object
```

```
# datetime型に変換  
df_uriage['日付'] = pd.to_datetime(df_uriage['日付'])  
# データ型の確認  
df_uriage.dtypes
```

```
日付      datetime64[ns]  
ハンバーガー売上件数 int64  
...略...  
dtype: object
```

例題4－11)

148

(2)日付カラムから月を抽出して、カラムに追加してください。

```
df_uriage['月'] = df_uriage['日付'].dt.strftime('%m')  
df_uriage.head()
```

	日付	ハンバーガー売上件数	ハンバーガー売上金額	ドリンク売上件数	ドリンク売上金額	月
0	2001-04-01	70	17780	70	14120	04
1	2001-04-02	90	24340	84	17240	04
2	2001-04-03	78	20060	76	15220	04
3	2001-04-04	72	20460	64	13580	04
4	2001-04-05	64	16600	62	12420	04

(応用)行、列への関数適用(apply)

applyを使うことで、DF.sum()やDF.var()などと同じようにいろいろな関数を行単位、列単位で適用できます。自分で定義した関数も利用できます。関数の引数は行もしくは列のSeriesとなります。

<行、列への関数適用>

```
DF.apply(func)
```

applyのオプション

func=必須

適用する関数を指定

axis=0

行単位(1)または列単位(0)の適用を選択可能

デフォルトは0

例題4-12)(参考)行、列への関数適用(apply)

自作のapply関数へはaxisの指定によって、列単位(axis=0)、行単位(axis=1)でデータを渡し、処理できます。

処理する軸(axis)を変えて処理がどう変わるか観察してみてください。

どちらの単位で集計関数などが必要になるかで、処理する軸を変更する必要があります。

Ex)列単位で平均をとって各項目の値と差分を取得したい場合は列単位を指定

```
df = pd.DataFrame({'品名': ['リンゴ', 'バナナ', 'みかん', '梨'], '値段': [100, 150, 50, 300], '重さ': [100, 300, 100, 200]})  
def aptest(Series):  
    print("=====")  
    print(f"{Series}")  
    print(f"{Series.mean()}")  
    print("=====")  
    return Series - Series.mean()
```

軸を変えてみよう

```
df[['値段', '重さ']].apply(aptest, axis=0)
```

例題4－13)【応用】

151

'食費'、'公共料金'、'医療費'、'雑費'について消費税を変更したdf_c_taxを作成してください。

```
def c_tax_8_10(Series):  
    result = Series * 1.10 / 1.08  
    return result  
  
df_c_tax = df_kakeibo[['食費', '公共料金', '医療費',  
                      '雑費']].apply(c_tax_8_10,axis=0)  
df_c_tax
```


例題4－13)【応用】

152

結果

	食費	公共料金	医療費	雑費
0	48006.851852	0.000000	3361.111111	12629.629630
1	49128.240741	17968.703704	3972.222222	25768.518519
2	55815.833333	25825.555556	8148.148148	25055.555556
3	41755.185185	10673.055556	1935.185185	28212.962963
4	36522.037037	16103.796296	6518.518519	26583.333333
5	48668.888889	13021.759259	2240.740741	10287.037037
6	55277.037037	20437.592593	11916.666667	15787.037037
7	40909.814815	17162.037037	4074.074074	11000.000000

⋮

(応用)各要素の関数適用(applymap)

applymapもapplyと同じように任意の関数を適用することができます。しかし、applyは各行・列に対して適用されるのに対して、applymapは各要素に適用されるという違いがあります。関数の引数は各要素です。

<各要素への関数適用>

```
DF.applymap(func)
```

applymapのオプション

func=必須

適用する関数を指定

<applyの適用単位(列)>

Index	col1	col2	col3
1	A1	A2	A3
2	B1	B2	B3
3	C1	C2	C3

<applymapの適用単位>

Index	col1	col2	col3
1	A1	A2	A3
2	B1	B2	B3
3	C1	C2	C3

例題4－12)【応用】

154

'年齢'もしくは'食費'が7の倍数のとき'Seven'、そうでないとき'NotSeven'とするdf_is_sevenを作成してください。

```
def is_seven(num):  
    if(num % 7) == 0:  
        result = 'Seven'  
    else:  
        result = 'NotSeven'  
    return result  
  
df_is_seven = df_kakeibo[['年齢', '食費']].applymap(is_seven)  
df_is_seven
```

例題4－12)【応用】

155

結果

	年齢	食費
0	NotSeven	NotSeven
1	NotSeven	NotSeven
2	NotSeven	NotSeven
3	NotSeven	NotSeven
4	NotSeven	NotSeven
5	NotSeven	NotSeven
6	NotSeven	NotSeven
7	NotSeven	Seven

⋮

- (1) df_test7を用いて、教科別の点数が80以上であれば'優'、そうでなければ '-' に置き換えた df_yu_hanteiを作成してください。
- (2) df_test7の'学校ID'と'教科ID'を整数型から文字列に型変換してください。

データの要約・集計

- 要約統計量
- 度数集計
- pivot_table

変数を集計して1元度数分布表や要約統計量を求めます。

<変数の度数集計>

```
DF['集計したい列'].value_counts()
```

<変数の要約>

```
DF.集計用関数()
```

集計用関数

count(): 非欠損値の数を計算

max(): 最大値を計算

mean(): 平均値を計算

min(): 最小値を計算

describe(): 主要な統計量を計算

size(): データの件数をカウント

std(): 標準偏差を計算

sum(): 総和を計算

var(): 分散を計算

例題5－1)

160

(1) 数値変数について、平均を求めてください。

```
#平均  
df_kakeibo.mean()
```

結果

```
会員ID    1050.251256  
年齢       36.331658  
地方ID     4.658291  
家賃      50924.623116  
食費      40070.396985  
公共料金  16701.175879  
医療費     4729.648241  
雑費      23025.628141  
dtype: float64
```

例題5－1)

161

(2) '家族構成'について、度数集計を行ってください。

```
#度数集計  
df_kakeibo['家族構成'].value_counts()
```

結果

```
未婚      92  
既婚      32  
こども1人  26  
こども3人  26  
こども2人  23  
Name: 家族構成, dtype: int64
```

グループ化した変数の要約・集計

162

列の値ごとにグループ化し、各グループで度数分布表や要約統計量を求めます。

<グループごとの度数集計>

```
DF.groupby(by='グループ分け列名')['集計列名(1列)'].value_counts()
```

<グループごとの要約>

```
DF.groupby(by='グループ分け列名')['集計列名'].集計用関数()
```

groupbyのオプション

by=None

グループ分けする列名を指定

複数の場合はリストで指定

as_index=True

グループ分けした列の値をindexにする

Falseの場合、indexにせず列に残る

value_counts 使用時はFalseにできない

※groupbyを使用して結果が省略された場合は、

pd.set_option("display.max_rows",任意の行数)で表示可能

例題5－2)

163

'家族構成'ごとに'年齢'の度数集計と分散を求めてください。

※get_group('こども1人')を追加する事で、家族構成のこども1人だけを表示

```
# 度数集計
```

```
df_kakeibo.groupby(by='家族構成')['年齢'].value_counts()
```

```
# 分散
```

```
df_kakeibo.groupby(by='家族構成')['年齢'].var()
```

度数集計の結果

家族構成	年齢	
こども1人	30	4
	37	4
	38	4
	31	2
	36	2
	39	2

分散の結果

家族構成	
こども1人	35.064615
こども2人	34.873518
こども3人	30.424615
既婚	75.983871
未婚	69.565217

Name: 年齢, dtype: float64

(応用)複数の要約統計量

164

複数の要約統計量を求めたい場合、集計用関数にagg()関数を使用します。

<複数の要約>

```
DF.groupby(by='グループ分け列名')['集計列名'].agg(func)
```

aggのオプション

func=必須

集計用の関数を指定

複数の場合はリストにして指定

※辞書形式にして列ごとに指定することもできる

例)B列のmin,maxとC列のsumを求める場合

```
df.groupby('A').agg({'B': ['min', 'max'], 'C': 'sum'})
```

例題5－3)【応用】

165

(1) '地方ID'ごとに'家賃'の平均、最大、最小を求めてください。

```
df_kakeibo.groupby(by='地方ID')['家賃'].agg(['mean', 'max', 'min'])
```

結果

	mean	max	min
地方ID			
1	52368.421053	62000.0	45000.0
2	49823.529412	58000.0	42000.0
3	54666.666667	63000.0	47000.0
4	49444.444444	67000.0	0.0
5	49464.285714	58000.0	0.0
6	51538.461538	58000.0	43000.0
7	51400.000000	62000.0	44000.0
8	50153.846154	56000.0	45000.0

例題5－3)【応用】

166

(2) '地方ID'ごとに'食費'の平均、'年齢'の最大、最小を求めてください。

```
df_kakeibo.groupby(by='地方ID').agg({'食費':'mean',  
                                     '年齢':['max', 'min']})
```

結果

	食費		年齢
	mean	max	min
地方ID			
1	34961.157895	48	31
2	41763.382353	48	25
3	44155.722222	48	27
4	39316.611111	50	22
5	38922.321429	49	22
6	40542.615385	50	25
7	39841.300000	49	22
8	40312.230769	45	23

(応用)pivot_tableを使った集計

167

まとめたい列が複数あるとき、pivot_tableを使うと見やすい形で集計ができる。

<複数の要約>

```
pd.pivot_table('データフレーム', index='index項目',  
               columns='カラム項目', values=['集計項目1', '集計項目2'],  
               aggfunc='集計関数')
```

agg_func

agg_funcに集計方法(デフォルトはmean)を指定可能

例題5－4)【応用】

168

(1) '地方ID'、'家族構成'毎に'雑費'、'食費'の平均値を算出してください。

```
pvtest = pd.pivot_table(df_kakeibo, index='地方ID',  
                        columns='家族構成',  
                        values=['雑費', '食費'], aggfunc='mean')  
pvtest.head()
```

結果

雑費						食費				
家族構成	こども1人	こども2人	こども3人	既婚	未婚	こども1人	こども2人	こども3人	既婚	未婚
地方ID										
1	26800.000000	28666.666667	NaN	15300.000000	18363.636364	44068.0	40779.666667	NaN	39217.500000	34122.909091
2	18416.666667	29125.000000	33787.500000	25300.000000	19314.285714	44782.5	53082.000000	49194.125	41250.500000	33062.714286
3	NaN	23075.000000	21450.000000	14883.333333	21300.000000	NaN	50145.000000	56838.500	44568.166667	35522.833333
4	14350.000000	45200.000000	34600.000000	12925.000000	15112.500000	37302.0	53444.500000	51279.000	43683.250000	31952.625000
5	13050.000000	NaN	35166.666667	19775.000000	23240.000000	49970.5	NaN	51550.000	40580.500000	32868.733333

- (1) 演習4-3(4)で作成したcsvを読み込み、df_test8を作成してください。なお、不要な列('Unnamed: 0')は削除してください。
- (2) それぞれの学校に所属する生徒数と、学校ごとに'5教科'の平均を求めてください。
- (3) 生徒ごとに、それぞれの教科の偏差値を求めてください。なお、計算のために作成した列は残さず、偏差値は整数で表してください。
(参考)
 - ・偏差値 = (点数 - 平均) / 標準偏差 × 10 + 50
 - ・applyの引数に代入するfuncの作成方法は(例題4-12、13)参照
 - ・applyで渡されたSeriesに対して.sum()等の関数が利用できる

データの可視化

- データ可視化について(matplotlib)
- 各種グラフ生成

headやdescribeでデータの形式、最大値・最小値を把握することができますが、棒グラフやヒストグラムといったグラフで可視化することでデータの傾向を把握し易くなります。データの可視化にはmatplotlibというライブラリを使用するのが一般的です。

<matplotlib使用の準備>

matplotlibでjupyter notebookに出力できるように設定

日本語が文字化けしないよう、フォントを設定

```
%matplotlib inline

# 日本語用フォント設定
import matplotlib
font = {'family': 'Yu Mincho'}
matplotlib.rc('font', **font)
```

データのプロット(matplotlib.pyplot)

173

matplotlib.pyplotを利用して表示します。

<準備>

```
import matplotlib.pyplot as plt
```

<記述方法>

plt.プロット用関数

プロット用関数

bar(x=,height=):棒グラフ

boxplot(x=):箱ひげ図

hist(x=,bins=):ヒストグラム

plot(x=,y=):折れ線プロット

scatter(x=,y=):散布図

プロット用関数(グラフ装飾用)

title():タイトル

xlabel():x軸の名前

ylabel():y軸の名前

xlim():x軸の範囲

ylim():y軸の範囲

<棒グラフ>

```
plt.bar(x='x軸の値', height='xに対応する棒の長さの値')
```

plt.barのオプション

x=必須

棒グラフを配置する横の座標。リストにして複数の棒を表示可能。

height=必須

棒の長さの値。リストにして複数の棒を表示可能。(xと対応させること)

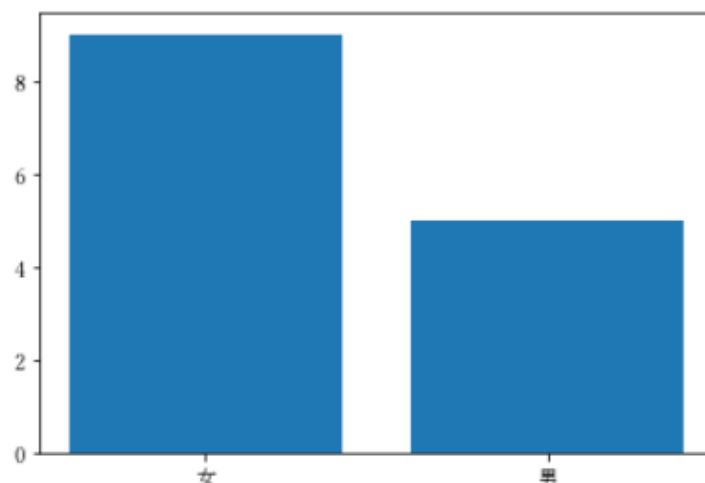
例題6－1)

175

(1) 「test_scores_V2.csv」を読み込み、性別毎の件数で棒グラフを作成してください。

```
df_test = pd.read_csv('test_scores_V2.csv')
sex_count = df_test['性別'].value_counts()
plt.bar(x=sex_count.index, height=sex_count.values)
plt.show()
```

結果



<箱ひげ図>

```
plt.boxplot(x='対象のデータ')
```

plt.boxplotのオプション

x=必須

対象データのリスト。多項目の場合、リストのリスト。

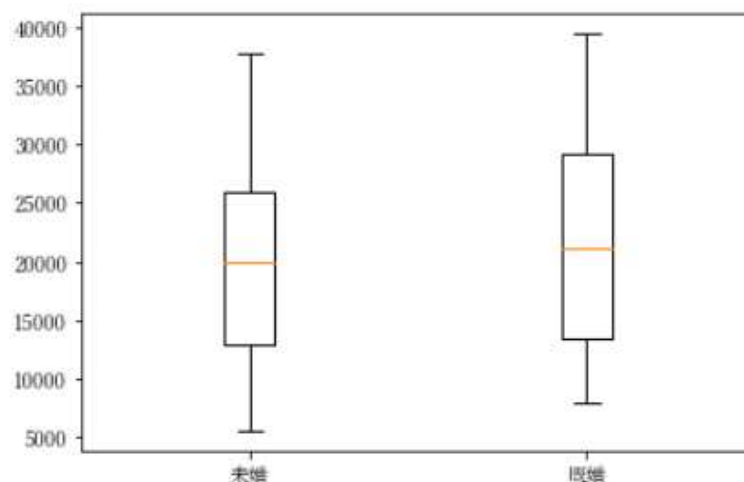
labels=None

各箱ひげに対するラベルを設定。リストにして指定する。

(1) 「kakeibo.csv」を読み込み、家族構成が既婚と未婚の雑費を比較する箱ひげ図を作成してください。

```
df_kakeibo = pd.read_csv('kakeibo.csv')
df_kakeibo.dropna(inplace=True)
points = ((df_kakeibo[df_kakeibo['家族構成'] == '未婚']['雑費'],
            df_kakeibo[df_kakeibo['家族構成'] == '既婚']['雑費']))
plt.boxplot(points, labels=["未婚", "既婚"])
plt.show()
```

結果



<ヒストグラム>

```
plt.hist(x='対象のデータ')
```

plt.histのオプション

x=必須

対象のデータのリスト。複数の場合、リストのリスト。

bins=None

整数を設定すると、分割する数を指定。リストを使うと分割境界を指定。

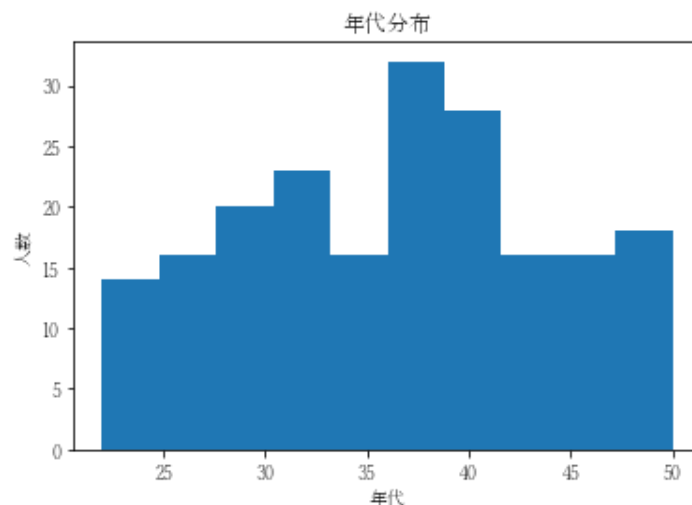
例題6－3)

179

(1) 年齢を10等分したヒストグラムを作成してください。タイトルは「年代分布」、x軸に「年代」、y軸に「人数」というラベルをつけてください。

```
plt.hist(df_kakeibo['年齢'], bins=10)  
plt.title('年代分布')  
plt.xlabel('年代')  
plt.ylabel('人数')
```

結果



<折れ線プロット>

```
plt.plot(x='横の値',y='縦の値')
```

plt.plotのオプション

x=任意

空白の場合、x軸は0からの整数になる。

y=必須

y軸の値、対象データのリストを設定する。

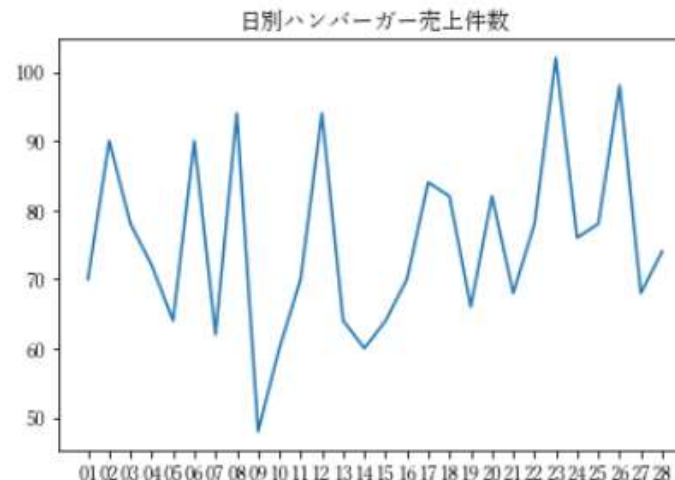
例題6－4)

181

(1) 日別のハンバーガー売上件数を折れ線グラフで表示してください。

```
df_uriage = pd.read_csv('uriage.csv')
df_uriage['日付'] = pd.to_datetime(df_uriage['日付'])
plt.plot(df_uriage['日付'].dt.strftime('%d'),
         df_uriage['ハンバーガー売上件数'])
plt.title("日別ハンバーガー売上件数")
plt.show()
```

結果



<散布図>

```
plt.scatter(x='横の値',y='縦の値')
```

plt.scatterのオプション

x, y = 必須
グラフに描画するデータ。

(2) 年齢ごとの平均家賃を集計したdf_kakeibo_meanから、年齢と平均家賃の散布図を作成してください。タイトルは「年齢と平均家賃」、x軸に「年齢」、y軸に「平均家賃」のラベルをつけてください。また、x軸は[30,50]、y軸は[30000,60000]の範囲で表示してください。

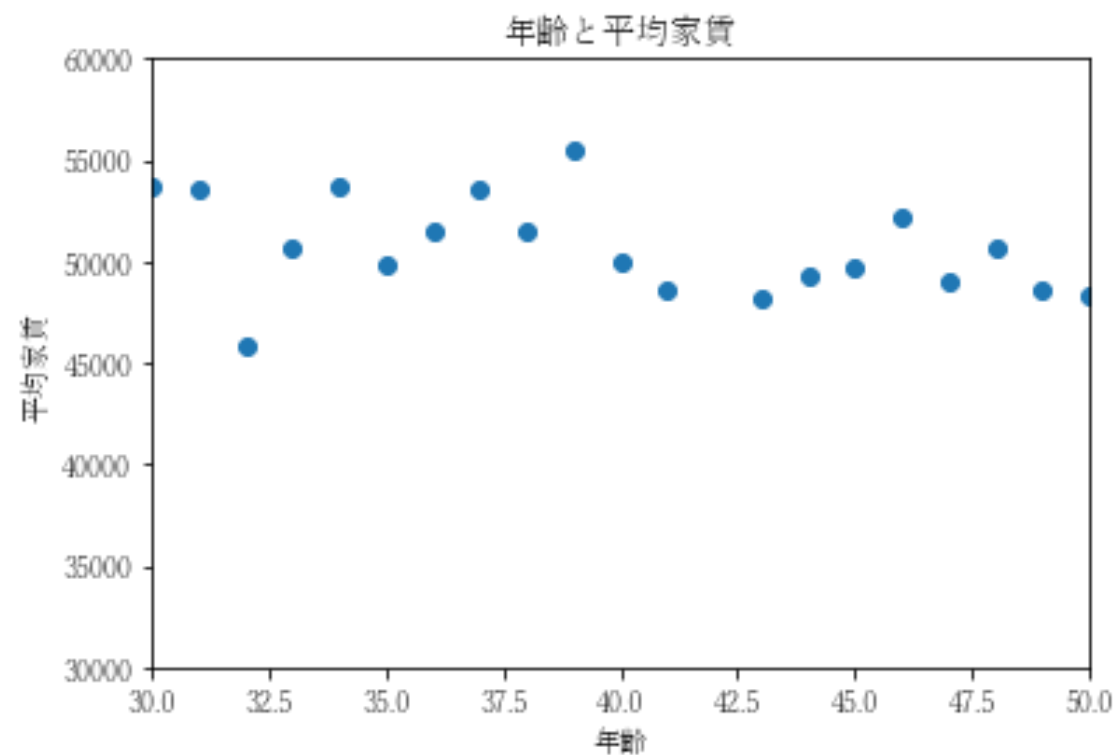
#データ作成

```
df_kakeibo_mean = df_kakeibo[['年齢', '家賃']].groupby(  
    by='年齢', as_index=False).mean()
```

#描画

```
plt.scatter(x=df_kakeibo_mean['年齢'], y=df_kakeibo_mean['家賃'])  
plt.title('年齢と平均家賃')  
plt.xlabel('年齢')  
plt.ylabel('平均家賃')  
plt.xlim(30,50)  
plt.ylim(30000, 60000)
```


結果



- (1) 演習5(1)で作成したdf_test8を用いて、各教科ごとに箱ひげ図を適切なタイトルとx,y軸ラベル付きで作成してください。
- (2) (1)で作成した箱ひげ図にそれぞれ教科名のラベルを付けてください。

(補足)matplotlibの日本語対応

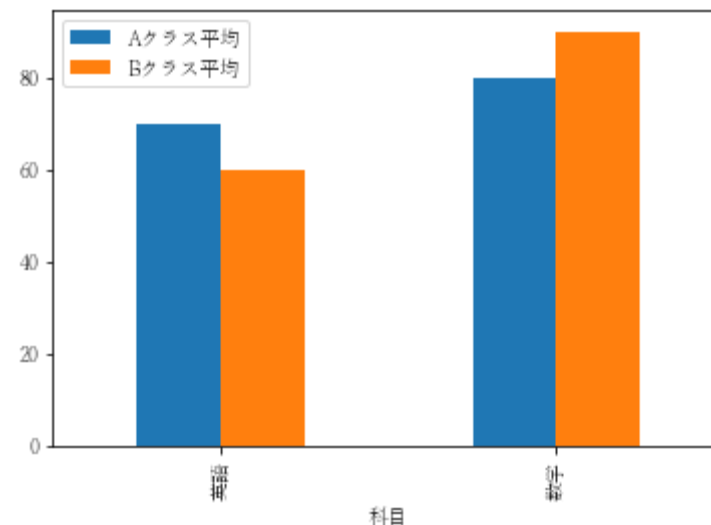
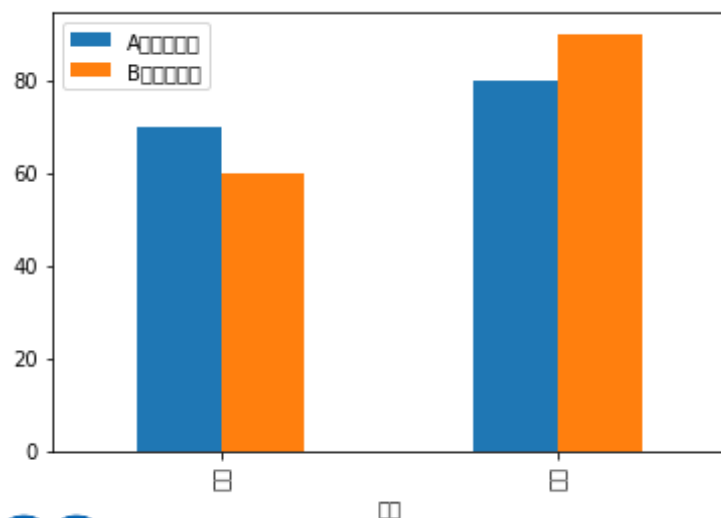
186

matplotlibのデフォルトのフォントは日本語に対応していません。そのためプロットに日本語を含むと、左下の図のように正しく表示されなくなってしまいます。

フォントを日本語に対応したものに切り替えることで日本語も正しく表示されます。今回は遊明朝体に切り替えます。

<記述方法>

```
import matplotlib
font = {'family':'Yu Mincho'}
matplotlib.rc('font', **font)
```



総合演習

Appendix

外部ライブラリのインストール、更新

189

Anacondaのcondaコマンドを利用して、外部ライブラリを管理することができます。

<外部ライブラリのインストール(tqdmをインストール)>

```
conda install tqdm
```

<外部ライブラリの更新(tqdmを更新)>

```
conda update tqdm
```

<インストール済み外部ライブラリの確認>

```
conda list
```

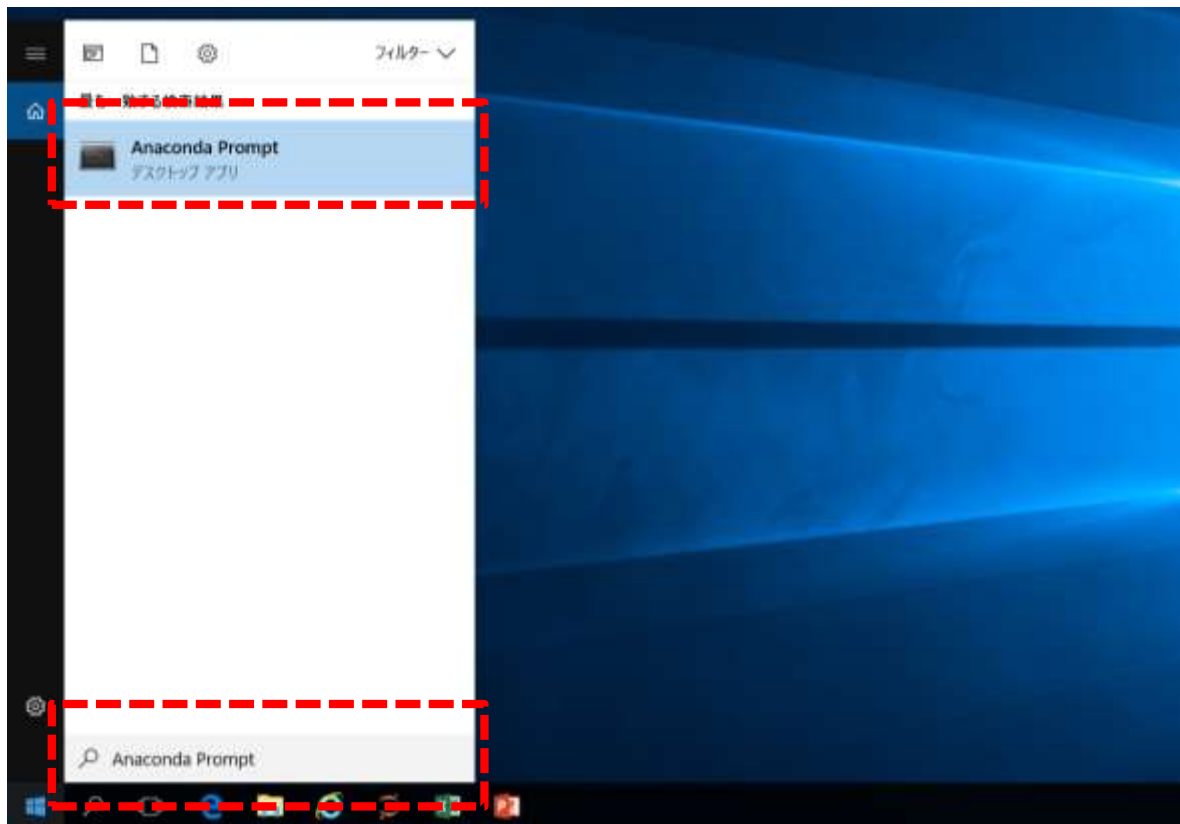
<外部ライブラリの削除(tqdmを削除)>

```
conda uninstall tqdm
```

※install、updateはインターネット接続が必要

1. Anaconda Promptの起動

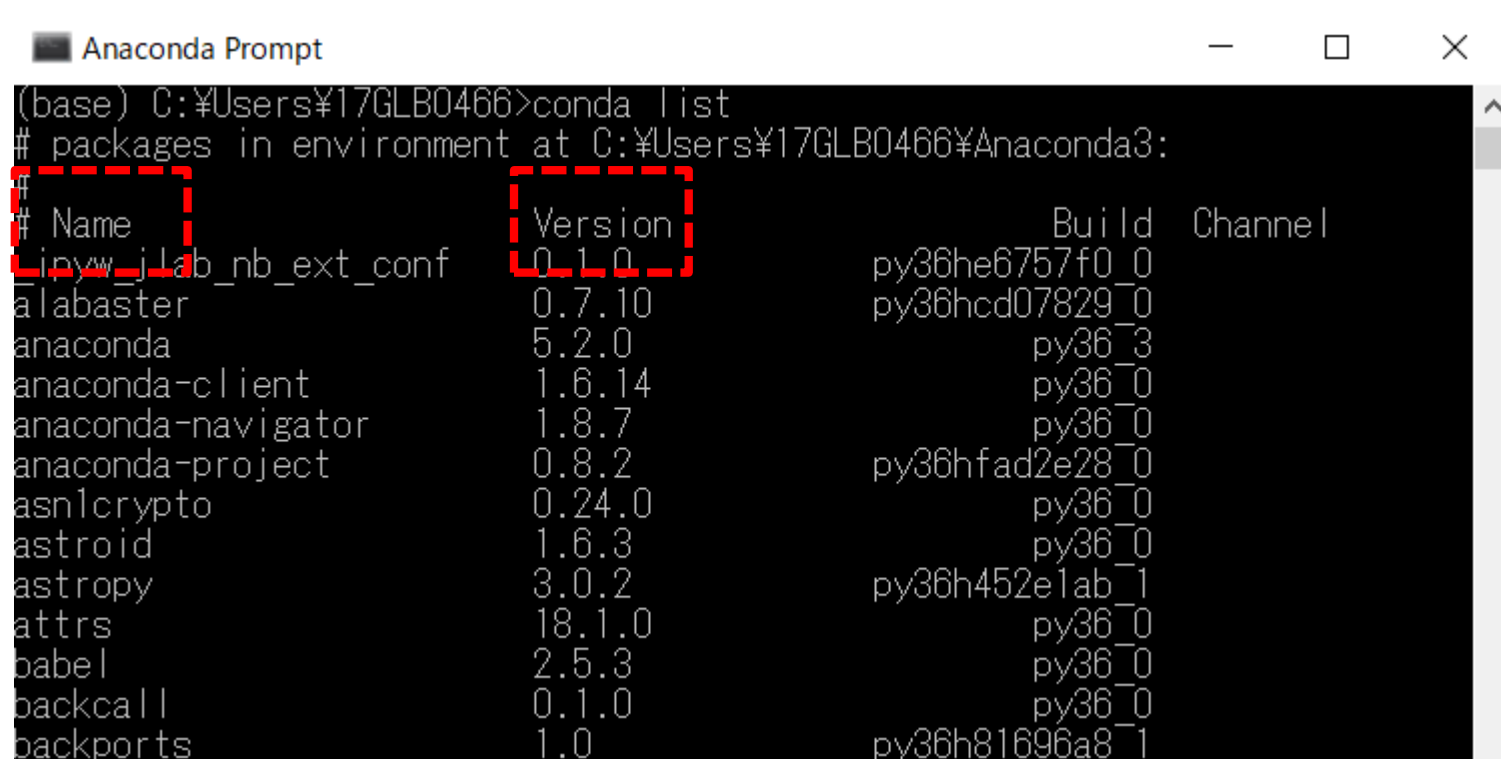
コマンドを入力するために、スタートメニューからAnaconda Promptを選択します。



2. 既存ライブラリの確認

conda list コマンドを入力し、インストール済みのライブラリを確認します。

Nameがライブラリ名で、Versionがライブラリのバージョンです。

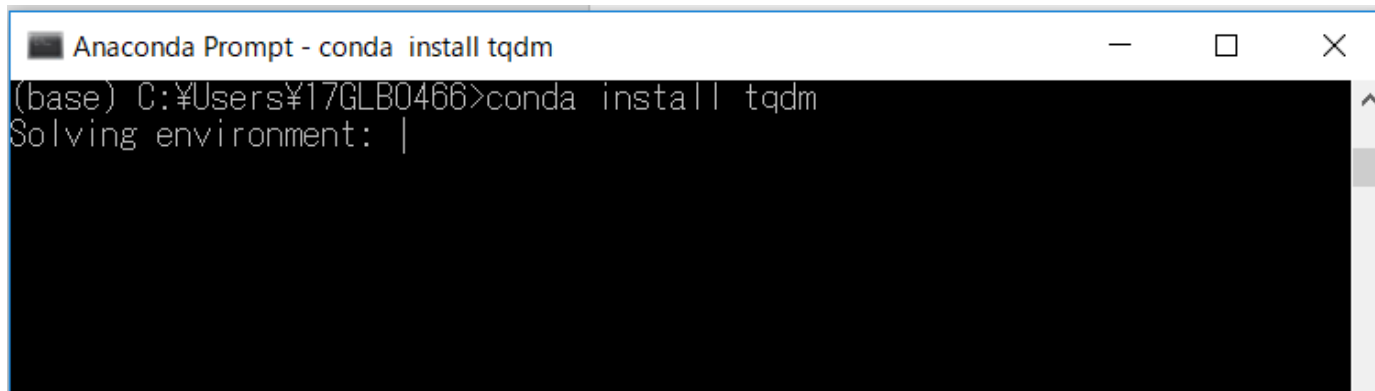


```
(base) C:\Users\¥17GLB0466>conda list
# packages in environment at C:\Users\¥17GLB0466\Anaconda3:
# Name                                Version                                Build                                Channel
#-----
ipyw_jlab_nb_ext_conf                0.1.0                                py36he6757f0_0                      py36hcd07829_0
alabaster                            0.7.10                               py36_3                              py36_0
anaconda                             5.2.0                                py36_0                              py36_0
anaconda-client                      1.6.14                               py36_0                              py36_0
anaconda-navigator                   1.8.7                                py36_0                              py36_0
anaconda-project                     0.8.2                                py36hfad2e28_0                      py36hfad2e28_0
asn1crypto                           0.24.0                               py36_0                              py36_0
astroid                              1.6.3                                py36_0                              py36_0
astropy                              3.0.2                                py36h452e1ab_1                      py36h452e1ab_1
attrs                                18.1.0                               py36_0                              py36_0
babel                                 2.5.3                                py36_0                              py36_0
backcall                             0.1.0                                py36_0                              py36_0
backports                             1.0                                  py36h81696a8_1                      py36h81696a8_1
```


3. 新規ライブラリのインストール

conda install コマンドを入力し、新規ライブラリをインストールします。
ここでは進捗バーライブラリのtqdmをインストールします。

ファイアーウォールやプロキシの設定は必要？

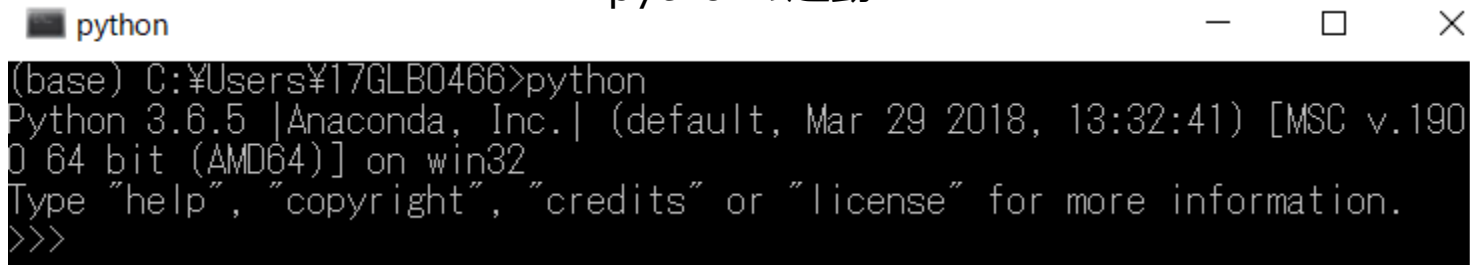


```
Anaconda Prompt - conda install tqdm
(base) C:\Users\¥17GLB0466>conda install tqdm
Solving environment: |
```

4. インストールの確認

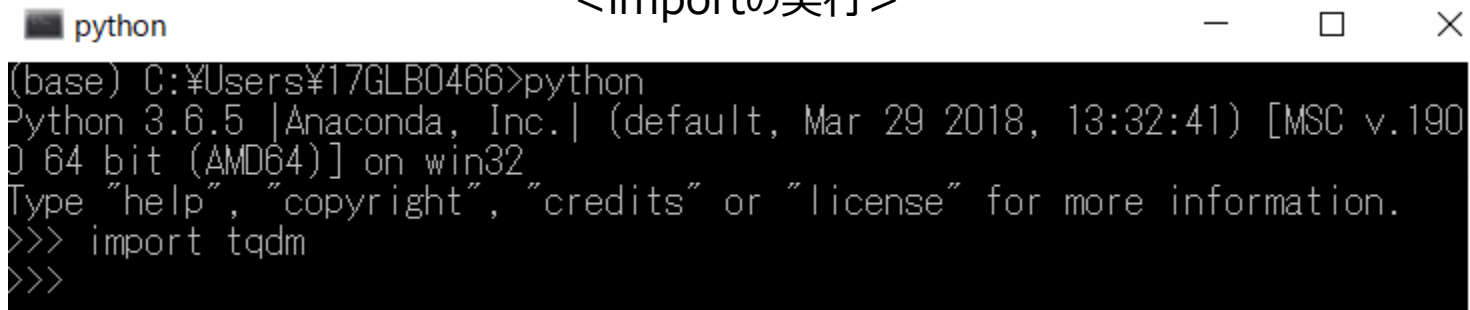
インストールが成功した場合、pythonでimportができるようになります。
importを試して、インストールが成功したか確認します。

<pythonの起動>



```
python
(base) C:\Users\¥17GLB0466>python
Python 3.6.5 [Anaconda, Inc.] (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

<importの実行>



```
python
(base) C:\Users\¥17GLB0466>python
Python 3.6.5 [Anaconda, Inc.] (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import tqdm
>>>
```