DATUM STUDIO

データサイエンティストのためのハンズオンセミナー
～Pythonで分析基礎を学ぶ！～

以下の環境を導入済み & 必要なデータをダウンロード済みですか？

- 開発環境
  - Python 3.4.0 以降
  - Pandas + scikit-learn の導入
  - Jupyter Notebook を利用できる環境
- データ（事前配布済み）
  - https://github.com/Salinger/datum_python_tutorial/raw/master/pre.zip

戸嶋　龍哉

Mail: t.tojima@datumstudio.jp

講師紹介

- 名前：戸嶋 龍哉（とじま たつや）
- 年齢：28 歳
- 略歴：
  - 長岡技術科学大学 工学研究科
    電気電子情報工学専攻 修士課程修了(～2014/3)
  - 株式会社ドリコム データ分析G
    (2014/4 ～ 2014/12)
  - DATUM STUDIO 株式会社
    (2015/01 ～ 現在)
- 出版：
  - データサイエンティスト養成読本 機械学習入門編
    2015/09

DATUM STUDIO 紹介

DATUM STUDIO

- DATUM STUDIO（デイタム　スタジオ）は、データを活用しようとするすべての企業を、人工知能を通し支援する会社です。どのデータと、どのツールを、どのように使えば、人工知能ビジネスの中で価値を生むかについて、お客様の状況とニーズに合わせ一緒に検討していきます。人工知能のカスタマイズ、Webクローラの作成、データ分析基盤の構築、その他、社員研修、分析コンサルティングなど、お客様ごとに最適な支援を行います。特に人工知能に基づく、データマイニング・パターン抽出領域において豊富な経験・技術力を有しており、お客様のデータにあわせた、ビジネス活用を支援する人工知能のカスタマイズ開発に強みがあります。

はじめに（１）

- **概要**

  何らかの意思決定を行う際の強い味方「データ分析」。

  今回のチュートリアルでは新たにデータ分析の世界に入門してみたい方向けに Python とその便利なライブラリ Pandas + scikit-learn を用い、統計学の様々な手法を利用してデータを解釈するための方法、可視化、そのための前処理などについてのハンズオンを行います。

はじめに（２）

- **身につく内容**
  - ビジネスにおけるデータ分析の流れ
  - 基礎統計量による分析（平均・分散など）
  - データの可視化（グラフの作成）
  - データの相関と回帰（回帰分析）
  - データの分類（決定木分析）

- **対象者**
  - Python もしくはデータ分析に興味のあるかた。
  - 高校数学の基礎知識を持っている。
  - 何らかの言語によるプログラミング経験があることが望ましい。

はじめに（３）

- **今回の進め方**
  - 基本的にスライドのページ毎に解説と、サンプルコードがあれば
    その実行時間をとりながら進めます。

  - キリの良い所で休憩（10～15分程度）をはさみます。
    この時間にも質問は受け付けます。

  - 質問は解説中に随時受け付けます。手を上げていただければ、
    チューターの方が対応します。

目次

- Jupyter-notebook で “Hello, world!”
- Python 文法基礎

午前の予定

昼休み

- ビジネスにおけるデータ分析の流れ
- 基礎統計量による分析
- Pandas の使い方
- データの可視化
- データの相関と回帰
- データの分類

午後の予定

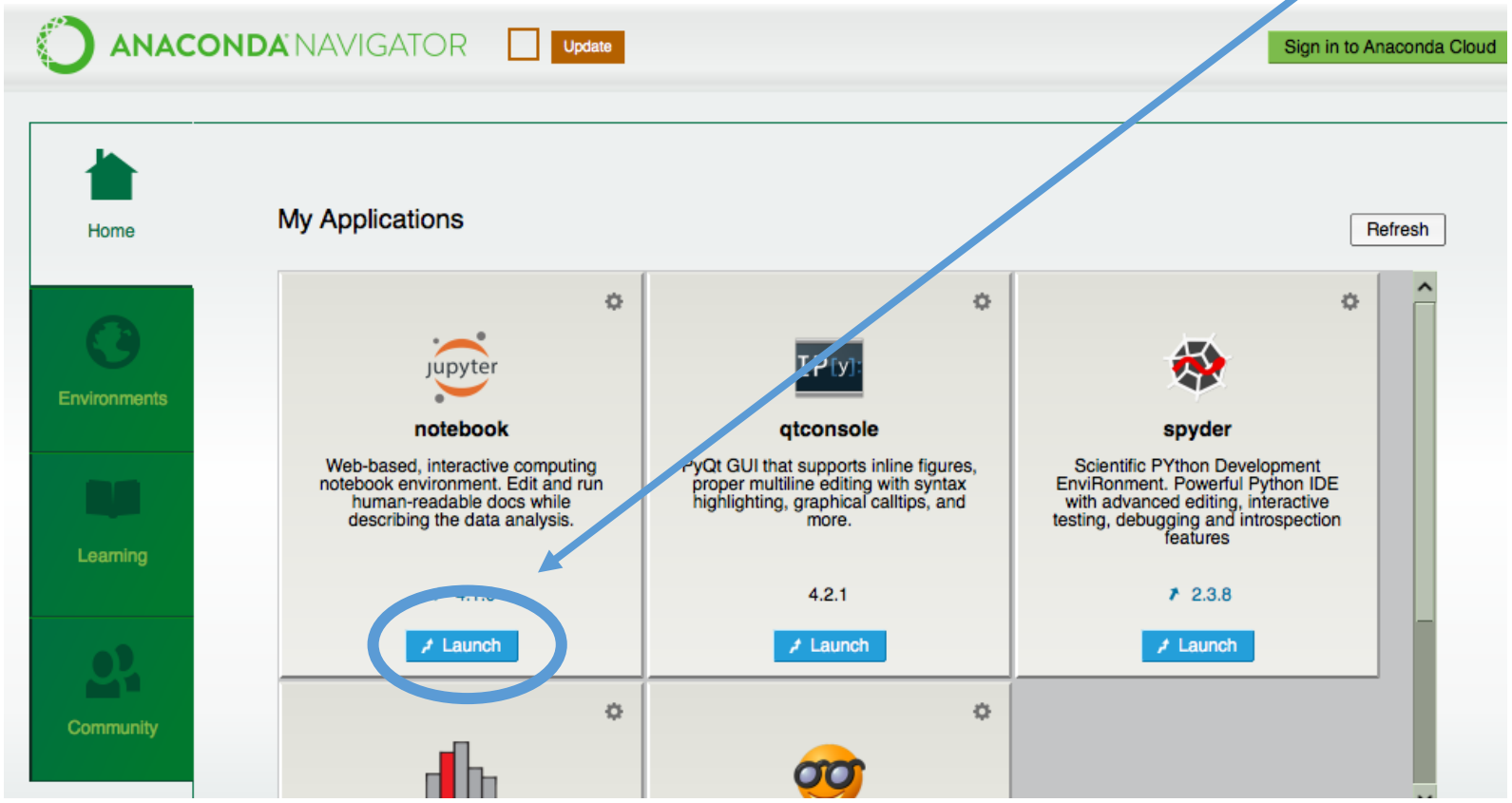
Jupyter Notebook で "Hello, world!"

Jupyter-notebook で “Hello, world!”

- Jupyter-notebook の起動
- Juputer-notebook の操作方法
- Hello, world!

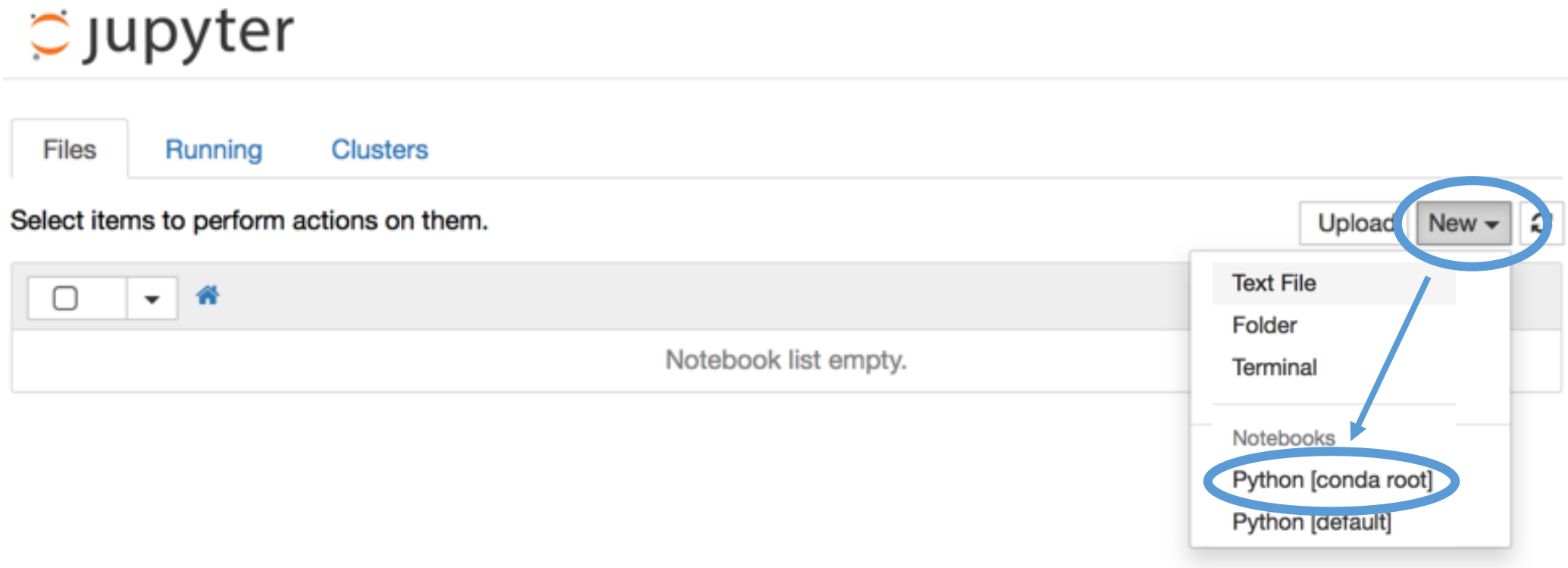
Jupyter-notebook の起動（１）

- Jupyter-notebook を起動してください。
  - ANACONDA NAVIGATOR で Jupyter-notebook 部分の “Launch” をクリック。
  - もしくはターミナルで `$ jupyter notebook` コマンドを実行。



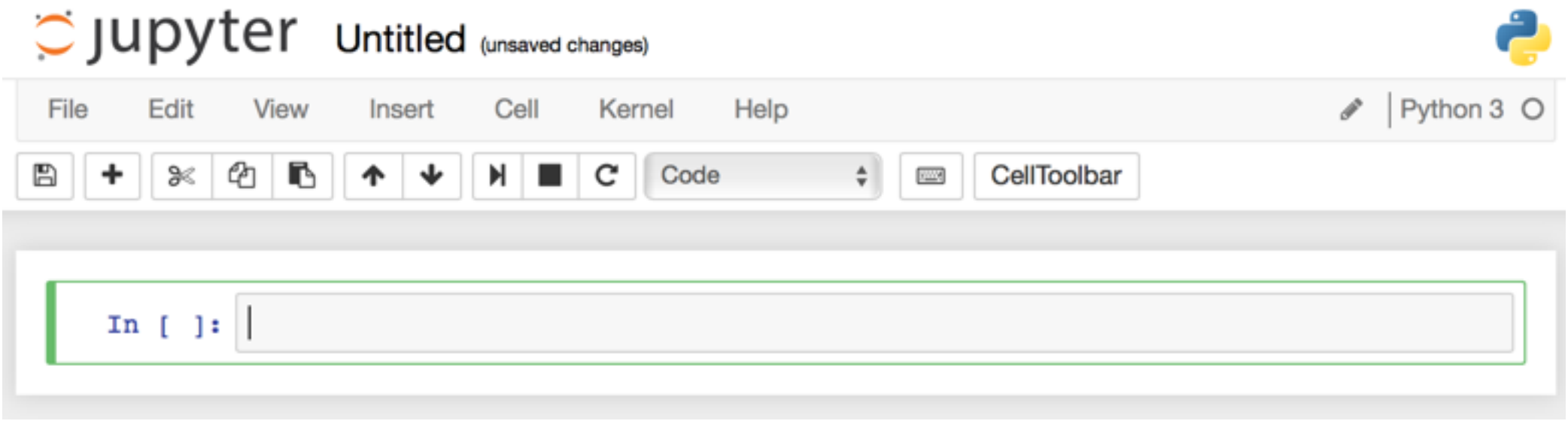
Jupyter-notebook の起動（２）

- 右上の “New” から “Python [conda root]” を選択。



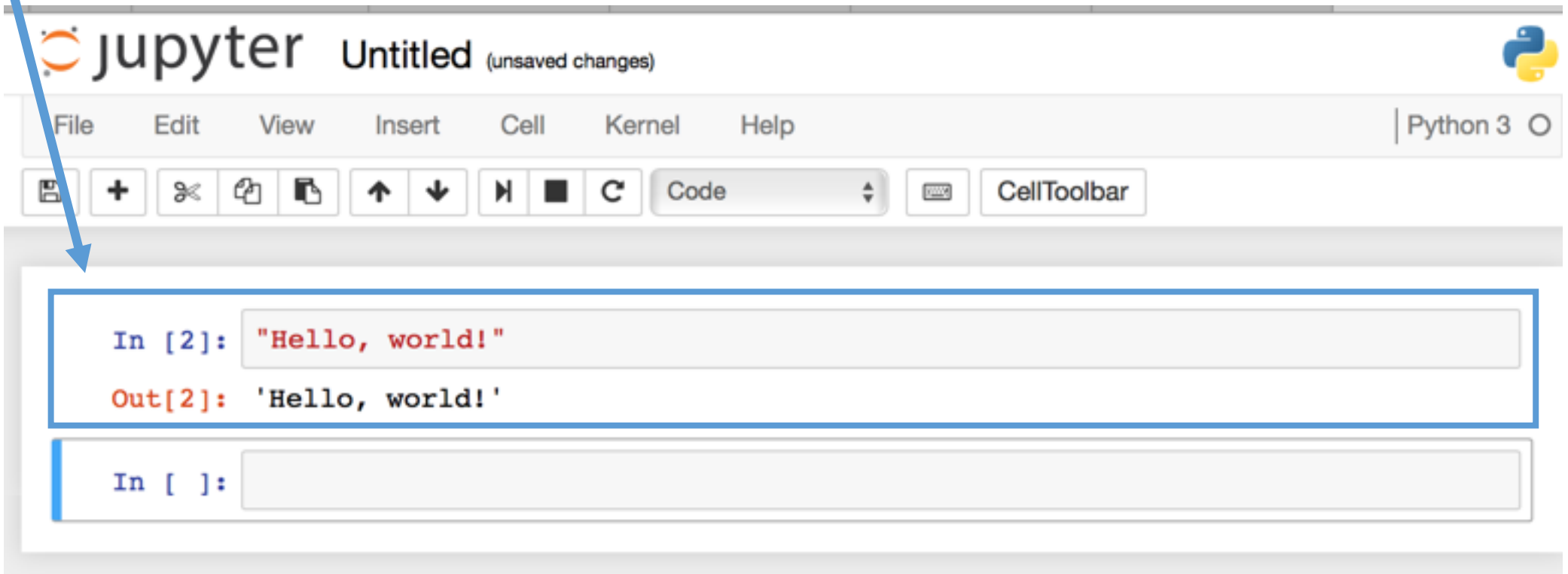
Jupyter-notebook の起動（３）

- このような画面が表示されればOKです。



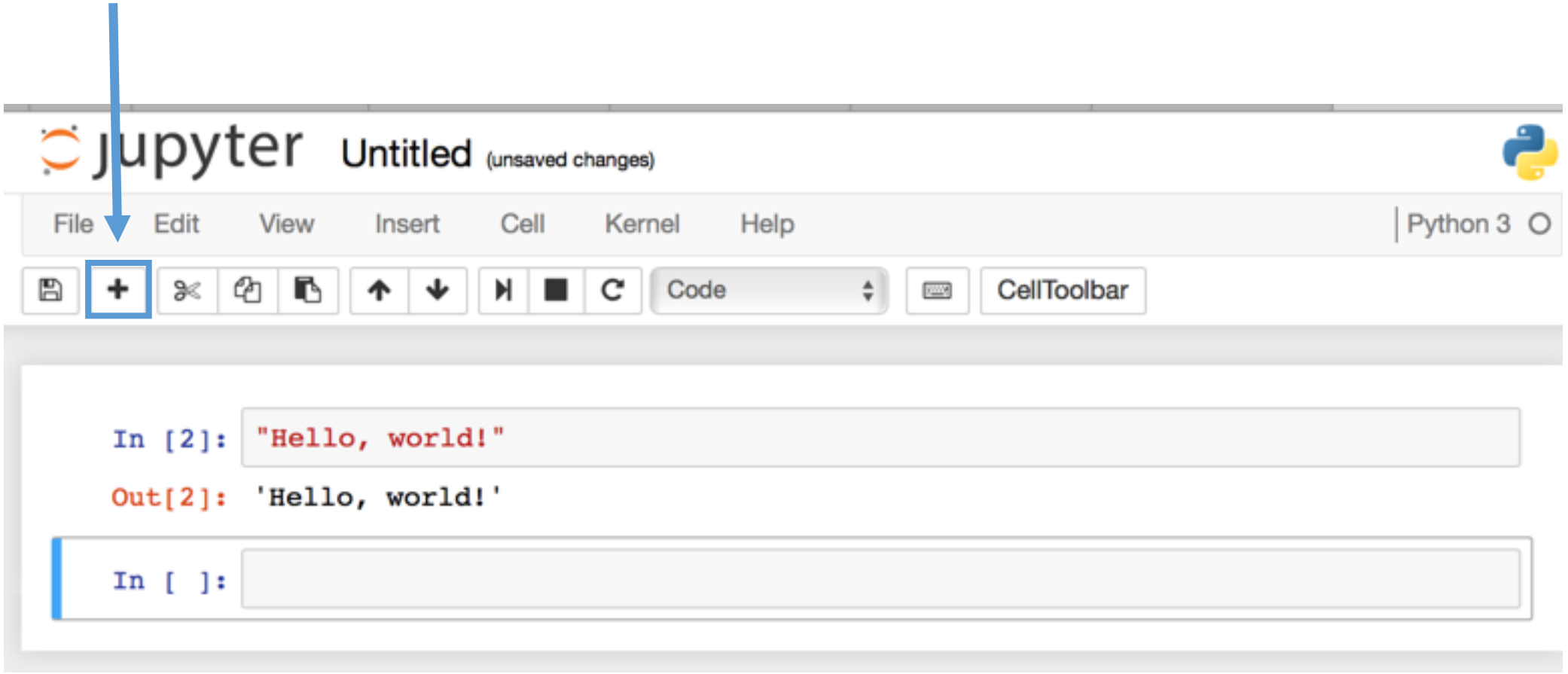
Jupyter-notebook の操作方法（１）

- セル：
  - プログラムを入力する所。実行後は戻り値があれば Out として表示される。



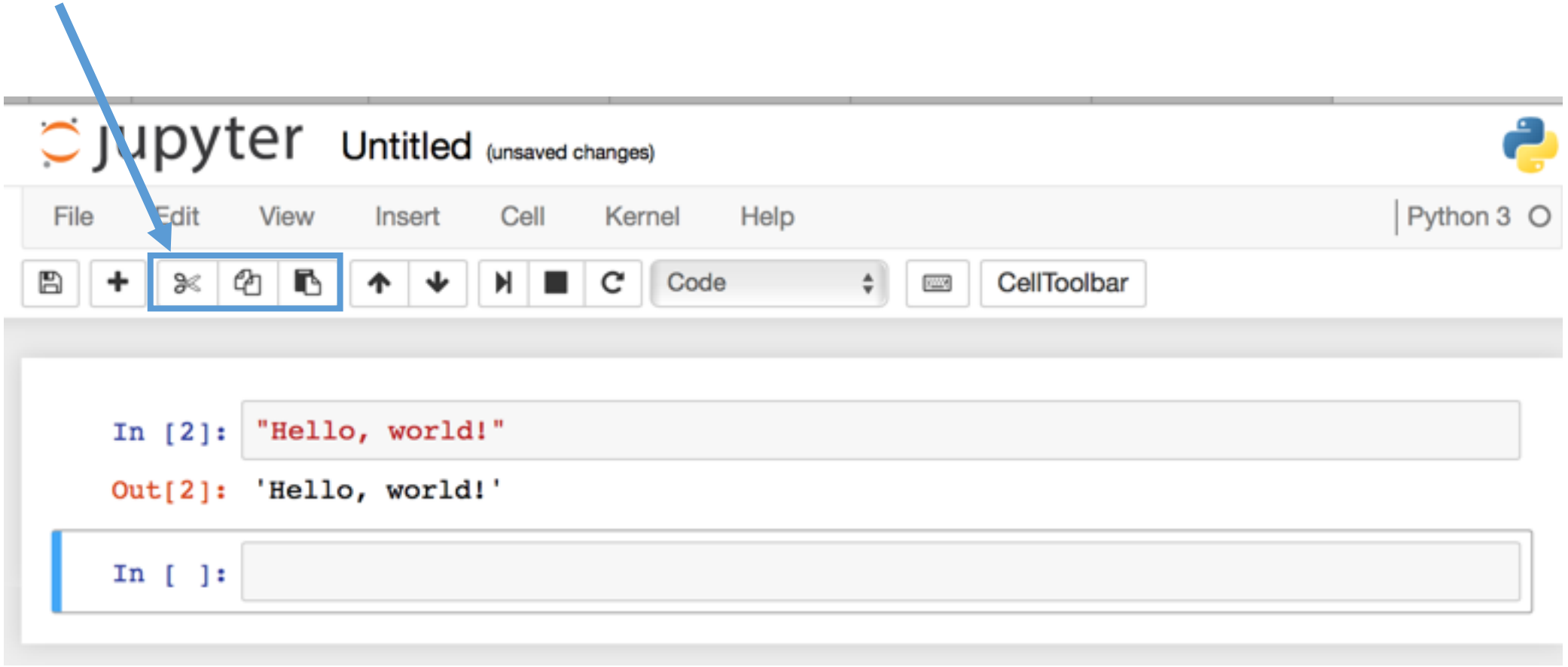
Jupyter-notebook の操作方法（２）

- 新規セルの追加



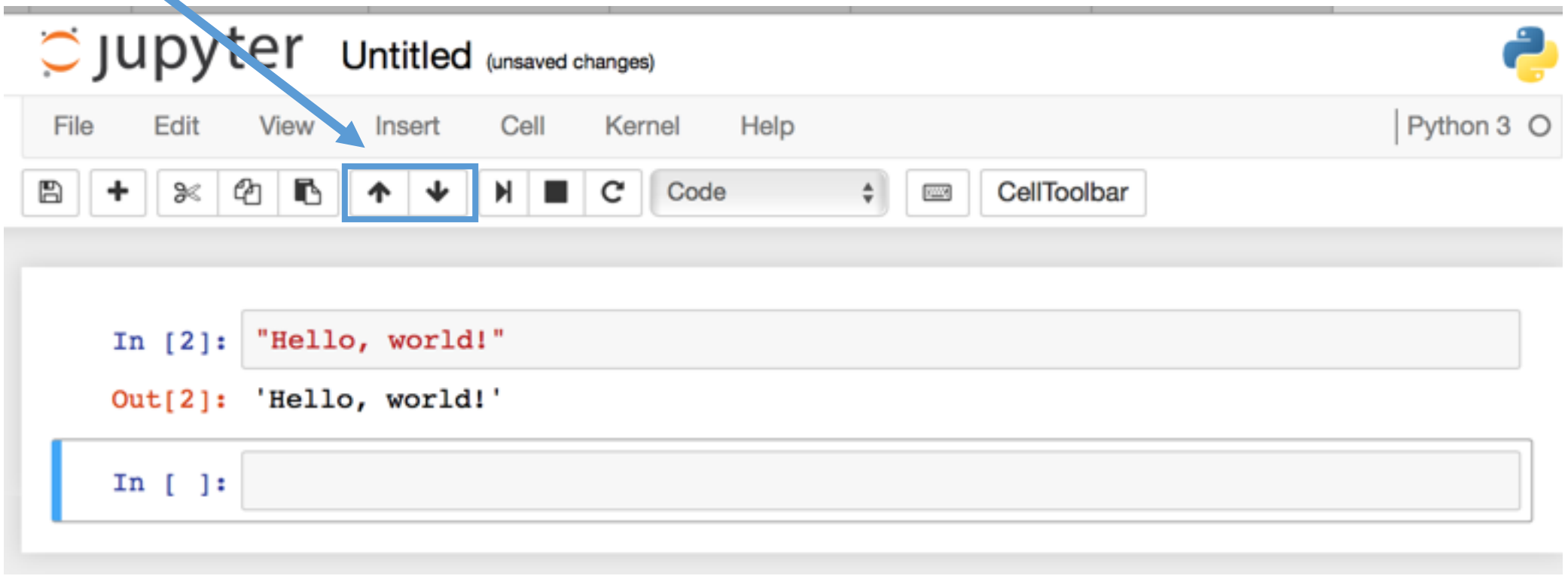
Jupyter-notebook の操作方法（３）

- 左から順にセルの切り取り（削除）・コピー・貼付け



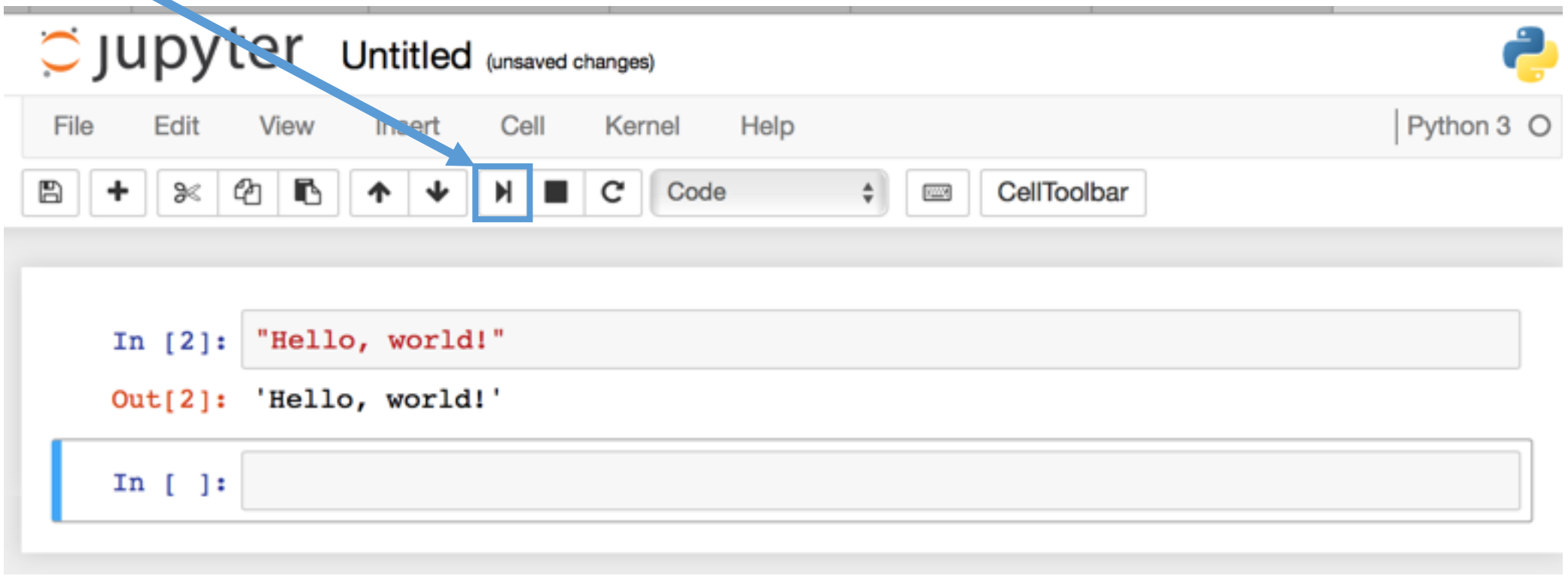
Jupyter-notebook の操作方法（４）

- 指定したセルの上下移動



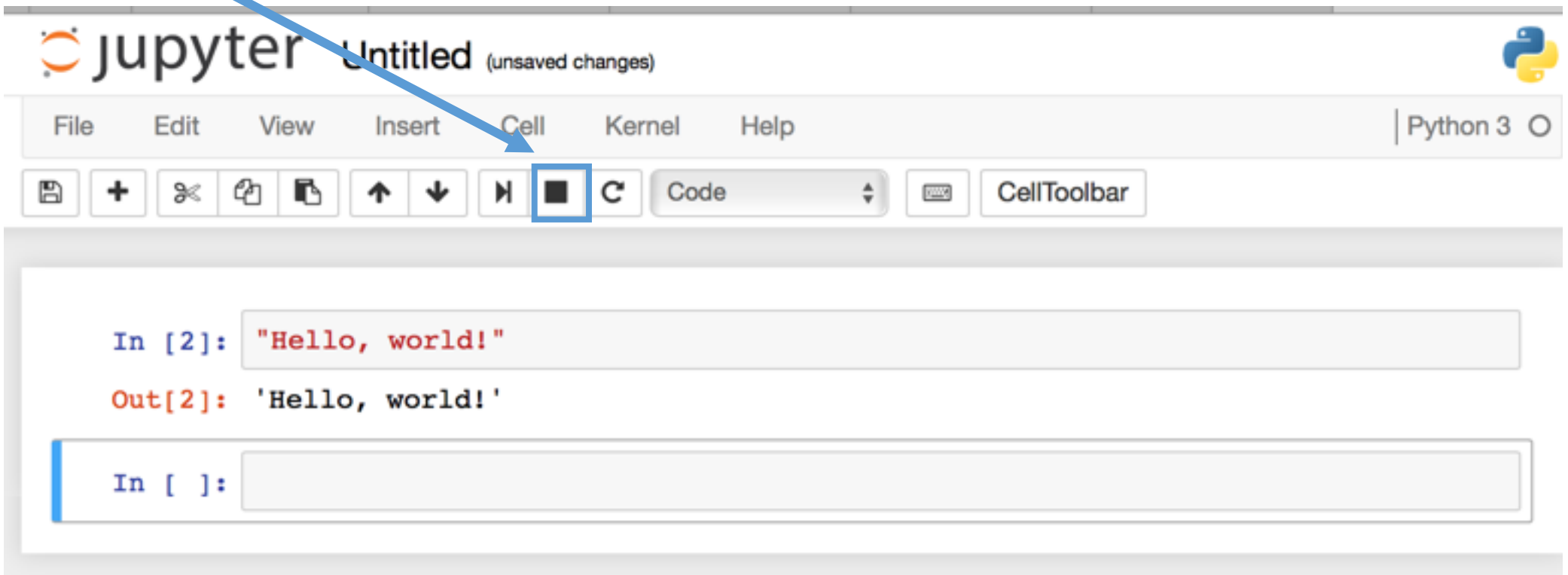
Jupyter-notebook の操作方法（５）

- 選択セルの実行
  - “Shift + Enter” でも実行可能。



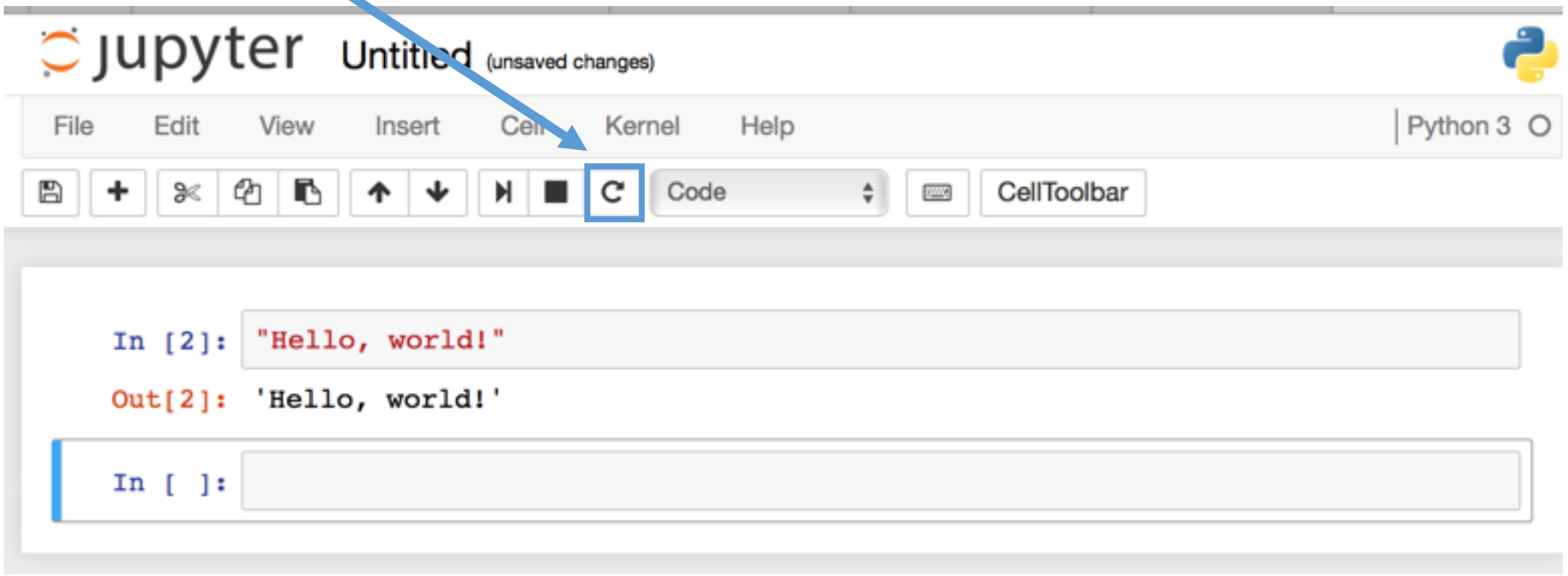
Jupyter-notebook の操作方法（６）

- 実行の停止
  - 途中で処理を終了したい時に。



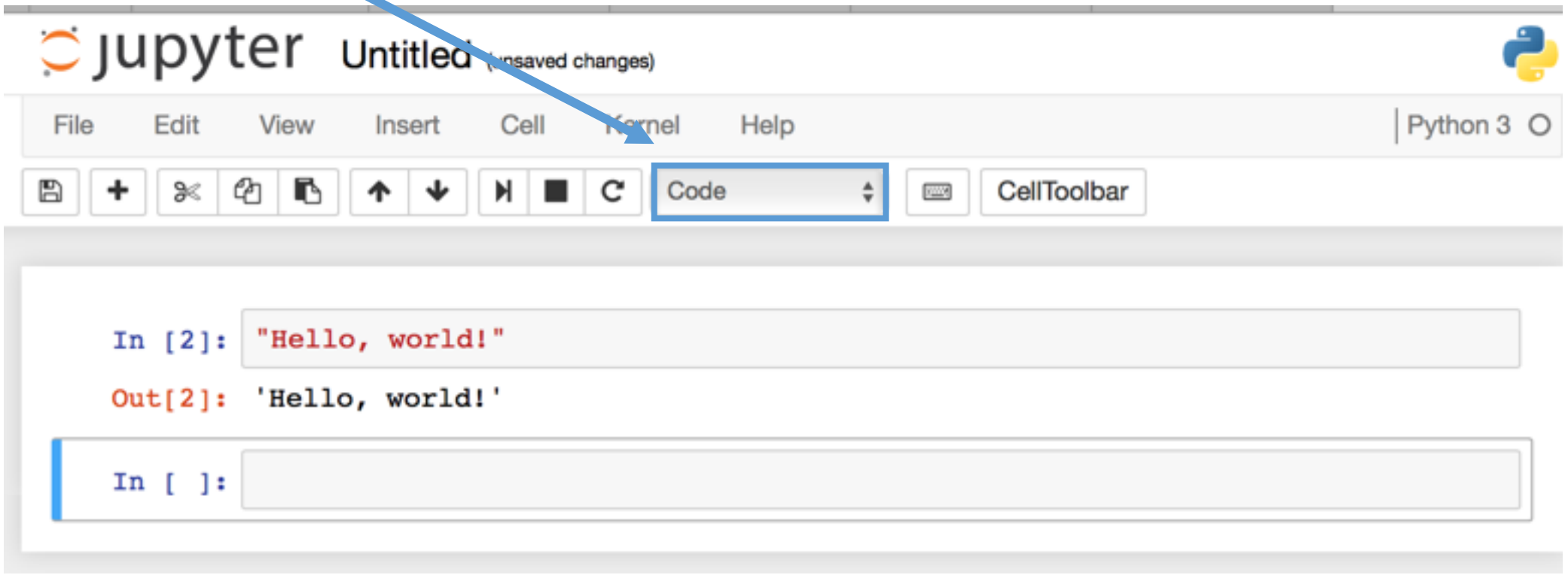
Jupyter-notebook の操作方法（７）

- カーネルの再起動
  - 変数などを初期化できる（セルを再実行するまで表示はそのまま）。



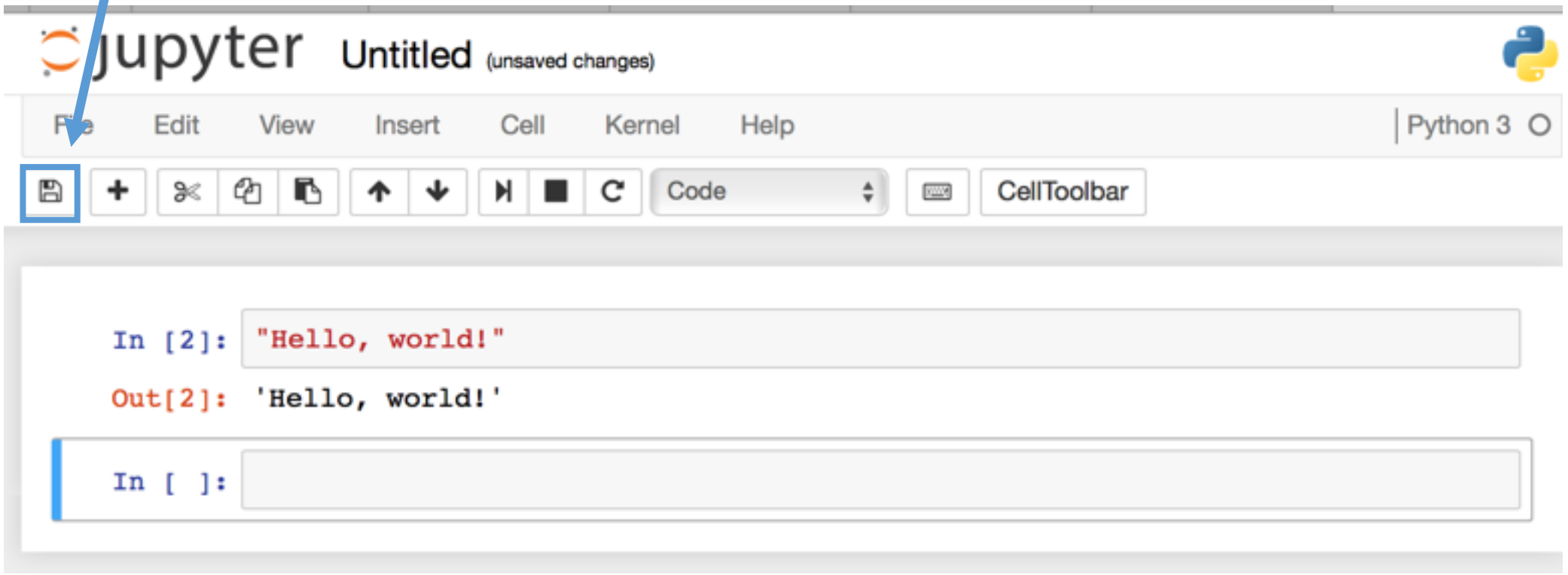
Jupyter-notebook の操作方法（８）

- セルの種類選択
  - Markdown 形式でテキストを記述することができる。



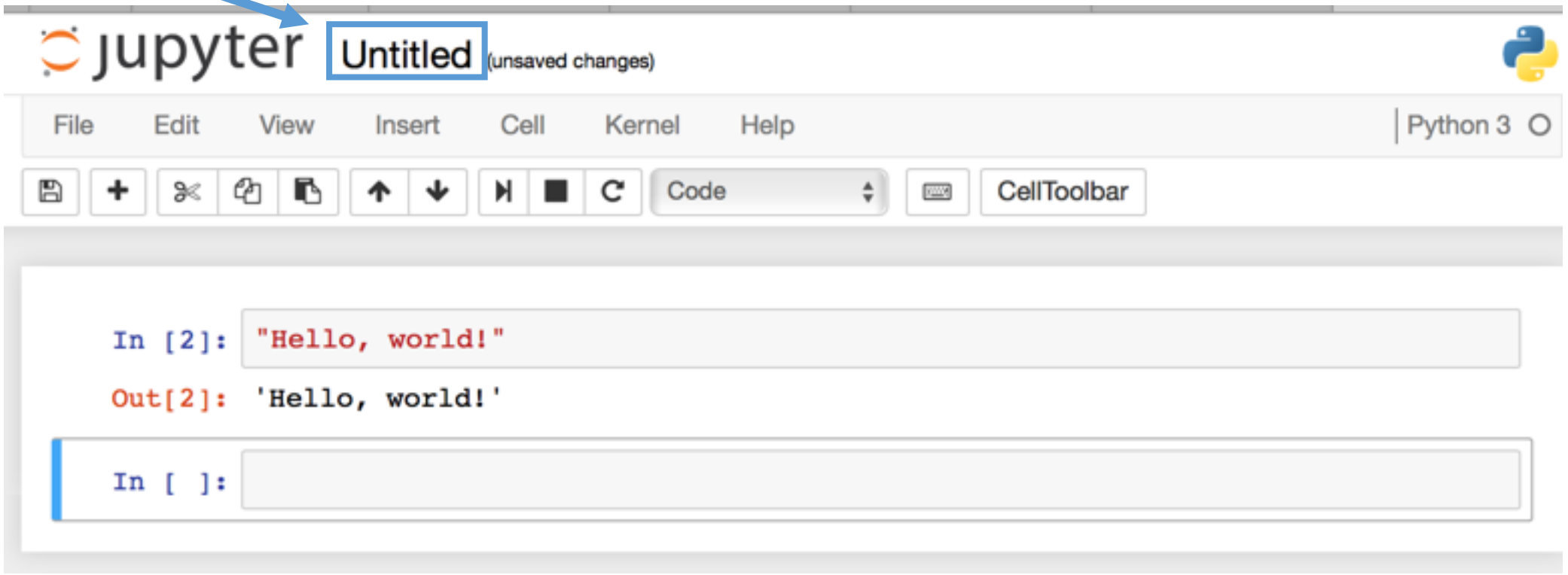
Jupyter-notebook の操作方法（９）

- 作業結果の保存
  - 一定間隔でオートセーブされる。



Jupyter-notebook の操作方法（10）

- タイトルの変更
  - クリックすることでタイトル（ファイル名）を変更できる。



Hello, world!（１）

- 以下操作を実行してみましょう。
  1. タイトルを ”Python データ分析入門” に変更する。
  2. セルに以下の内容を入力して実行し、結果を得る。

```
"Hello, Python!"
```

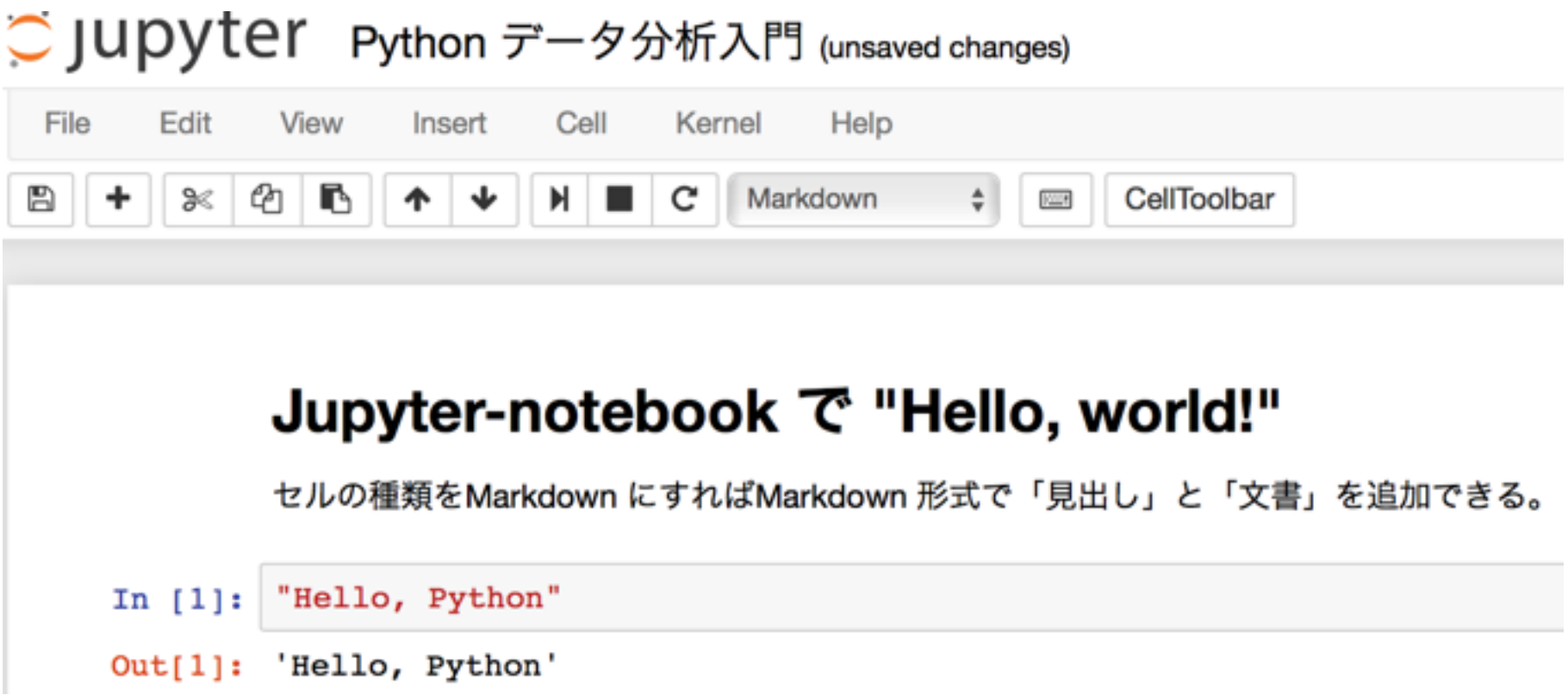
この枠で囲っている
部分を実行してください

  3. 先ほど入力したセルの「上」に新規セルを追加する。
  4. 上記 3. で追加セルの種類を “Markdown” に変更する。
  5. 追加したセルに以下の内容を入力して実行する。

```
# Jupyter-notebook で "Hello, world!"
セルの種類をMarkdown にすればMarkdown 形式で「見出し」と「文書」を追加できる。
```

Hello, world!（２）

- このように説明とプログラムをまとめて書くことが出来ます。



Python 文法基礎

Python 文法基礎

- Python の文法
  - コメント
  - 画面への出力
  - ライブラリのインポート
  - 数値・文字列・ブール値
  - 変数
  - リスト・タプル・ディクショナリ
  - 演算
  - 制御文
  - 関数

コメント

- # （シャープ）から始まる行はコメントです。
- プログラムの説明のために書く。実行時に解釈されない。

```
# '#' から始まる行はコメントとなる
# プログラム実行時に解釈されない
```

```
In [2]: # '#' から始まる行はコメントとなる
        # プログラム実行時に解釈されない
```

画面への出力

- Jupyter Notebook の場合、文字列・数値・関数など値が返るものをを評価した場合は最後のものが画面に出力される。

- print 関数を利用することで任意の要素を画面に出力できる。

```
print("Hello, world!")
```

```
Hello, world!
```

```
# カンマで区切ることで複数要素を同時に出力できる
print("You got", 100, "yen.")
```

```
You got 100 yen.
```

ライブラリの読み込み

- よく使う処理をまとめたものがライブラリ。
- Python 標準のもの（日付処理、ファイル処理 など）と外部のライブラリ（今回利用する Pandas や scikit-learn など）がある。規定の手順に従えば自分で任意のライブラリを導入することが出来る。

```
# pandas というライブラリを pd という名前で読み込む
import pandas as pd
```

```
# あるモジュールの特定の要素のみを読み込む
# datetime モジュールから date クラス
from datetime import date
```

数値

- Python では主に整数・小数・複素数が扱える。

```
# 整数
1000
```

```
# 小数
0.125
```

```
# 複素数
0.5 + 1.3j
```

文字列（１）

- Python では任意の複数個の文字をシングルクォートもしくはダブルクォートで囲むことで文字列として扱える。

```
# ダブルクォートでもシングルクォートでもよい
"こんにちわ"
```

```
In [10]: # ダブルクォートでもシングルクォートでもよい
         "こんにちわ"

Out[10]: 'こんにちわ'
```

```
'みなさん'
```

```
In [11]: 'みなさん'

Out[11]: 'みなさん'
```

```
# 文字列中に文字列を埋め込む
"こんにちは [%s] さん" %("太郎")
```

```
In [12]: # 文字列中に文字列を埋め込む
         "こんにちは [%s] さん" %("太郎")

Out[12]: 'こんにちは [太郎] さん'
```

文字列（２）

- 数値を任意のフォーマットで文字列中に埋め込める。
  - 整数値は "d"、小数値は "f"。

```
# 文字列中に整数を埋め込む"
年齢: %d歳" %(27)
```

```
# 桁数を指定して整数の埋め込み (先頭ゼロ埋め)
"年齢: %02d歳" %(9)
```

```
# 桁数を指定して小数の埋め込み
# (小数点以下3桁まで)
"売上の増加率: %0.3f" %(0.123456)
```

```
In [13]: # 文字列中に整数を埋め込む
         "年齢： %d歳" %(27)

Out[13]: '年齢： 27歳'

In [14]: # 桁数を指定して整数の埋め込み（先頭ゼロ埋め）
         "年齢： %02d歳" %(9)

Out[14]: '年齢： 09歳'

In [15]: # 桁数を指定して小数の埋め込み（小数点以下3桁まで）
         "売上の増加率： %0.3f" %(0.123456)

Out[15]: '売上の増加率： 0.123'
```

ブール値（１）

- 条件分岐などで使用する。
- 条件が成り立つ場合は True、成り立たない場合は False。

```
In [16]: # 条件分岐などで使う
         # 真の時は True
         True

Out[16]: True
```

```
In [17]: # 偽のときは False
         False

Out[17]: False
```

```
In [18]: # not で反転できる
         not True

Out[18]: False
```

```
In [19]: # and
         # a and b で a, b どちらも True のとき True になる
         True and False

Out[19]: False
```

```
In [20]: # or
         # a or b で a, b どちらかが True のとき True になる
         True or False

Out[20]: True
```

ブール値（２）

- 「等しい」かどうか、大小関係が成立するかどうかも扱う。

```
In [21]: # a == b : a, b が等しい
         a = 10
         b = 15
         a == b
```

```
Out[21]: False
```

```
In [22]: # a != b : a, b が等しくない
         a != b
```

```
Out[22]: True
```

```
In [23]: # a < b  : a が b より小さい
         a < b
```

```
Out[23]: True
```

```
In [24]: # a <= b : a が b より小さい or 等しい
         a <= b
```

```
Out[24]: True
```

```
In [25]: # a > b  : a が b より大きい
         a > b
```

```
Out[25]: False
```

```
In [26]: # a >= b : a が b より大きい or 等しい
         a >= b
```

```
Out[26]: False
```

変数

- 先ほどのスライドで a = 10 のように値を定義していましたが、このときの a を変数と呼びます。
- 値を一時的に格納しておくための箱のようなものです。

```
# s という変数に "Hello, world!" という文字列を代入する
s = "Hello, world!"
```

```
# 何度も同じ値を参照できる
s
```

```
Out[28]:  'Hello, world!'
```

リスト・タプル・ディクショナリ（１）

- 複数要素をまとめて取り扱うために以下のものがある。
  1. リスト
  2. タプル
  3. ディクショナリ

リスト・タプル・ディクショナリ（２）

- リストの定義

```
# リストを利用することで複数要素を並べられる
[1, 2, 3, 4, 5]
```

```
[1, 2, 3, 4, 5]
```

```
["a", "b", "c"]
```

```
['a', 'b', 'c']
```

```
# リストは同じ種類（型）のデータでなくても並べられる
[1, "tanaka", "taro", 32, "male"]
```

```
[1, 'tanaka', 'taro', 32, 'male']
```

```
# 入れ子にもできる
[[1, "tanaka", "taro", 32, "male"],
 [2, "suzuki", "jiro", 20, "male"],
 [3, "suzuki", "hanako", 26, "female"]]
```

```
[[1, 'tanaka', 'taro', 32, 'male'],
 [2, 'suzuki', 'jiro', 20, 'male'],
 [3, 'suzuki', 'hanako', 26, 'female']]
```

リスト・タプル・ディクショナリ（３）

- リストの要素へのアクセス

```
# 添字を指定することで n 番目の値にアクセスできる。
# n は 0 から始まるので注意
l = [1, 2, 3, 4, 5]
l[1]
```

```
2
```

```
l[3]
```

```
4
```

```
# マイナスの値を指定することで逆から辿れる
l[-2]
```

```
4
```

```
# 範囲を指定して特定部部分のみ切り出せる
# 終端に指定した場所の一つ手前まで切り出されるので注意
l[0:2]
```

```
[1, 2]
```

```
# 先頭もしくは末尾までの場合は省略できる
l[2:]
```

```
[3, 4, 5]
```

リスト・タプル・ディクショナリ（４）

- リストの要素の書き換え・追加・削除

```
# リストは値を書き換え可能
l[0] = 11
l
```

```
[11, 2, 3, 4, 5]
```

```
# append() で末尾に要素を追加可能
l.append(6)
l
```

```
[11, 2, 3, 4, 5, 6]
```

```
# del で要素の削除が可能
del l[3]
l
```

```
[11, 2, 3, 5, 6]
```

```
# pop() で指定した要素の削除と同時に値の取得ができる
l.pop(0)
```

```
11
```

```
l
```

```
[2, 3, 5, 6]
```

リスト・タプル・ディクショナリ（５）

- タプル
  - 値を書き換えられないリスト。

```
# タプルはほぼリストと同じ操作ができる
t = (1, 2, 3, 4, 5)
t
```

```
(1, 2, 3, 4, 5)
```

```
# ただし値を書き換えることが出来ない
# 例外が発生してプログラムが強制終了する
t[0] = 11
```

```
---------------------------------------------------------------------------
--
TypeError                                 Traceback (most recent call las
t)
<ipython-input-219-5370011e5d65> in <module>()
      1 # ただし値を書き換えることが出来ない
      2 # 例外が発生してプログラムが強制終了する
----> 3 t[0] = 11

TypeError: 'tuple' object does not support item assignment
```

リスト・タプル・ディクショナリ（6）

- ディクショナリは key-value のペアを表現できる。

```
# ディクショナリは key-value のペアを表現できる
d = {
    "a": 1,
    "b": 15,
    "c": "文字もOK"
}
d
```

```
{'a': 1, 'b': 15, 'c': '文字もOK'}
```

```
# 添字に key を指定することで任意の要素を取り出せる
d["c"]
```

```
'文字もOK'
```

```
# 値を書き換え可能
d["b"] = "replaced"
d
```

```
{'a': 1, 'b': 'replaced', 'c': '文字もOK'}
```

演算（1）

- 数値の足し算・引き算

```
# 整数同士の足し算
10 + 15
```

```
25
```

```
# 小数値の引き算
15.6 - 10
```

```
5.6
```

```
# 複素数の足し算
a = 10 + 3j
b = -3 + 5j
a + b
```

```
(7+8j)
```

演算（2）

- 掛け算・割り算

```
# 整数の掛け算
10 * 1000
```

```
10000
```

```
# 累乗
10**3
```

```
1000
```

```
# 小数の割り算（そのまま）
15.6 / 10
```

```
1.56
```

```
# 小数の割り算（商のみ：小数値以下切り捨て）
15.6 // 10
```

```
1.0
```

```
# 小数の割り算の余りの取得
15.6 % 10
```

```
5.6
```

演算（3）

- 文字列の演算

```
# 文字列の結合
"Hello" + ", " + "world!"
```

```
'Hello, world!'
```

```
# 文字列の繰り返し
"Hello! " * 3
```

```
'Hello! Hello! Hello! '
```

制御文（１）

- if： 条件分岐のための構文

```
# if 文で条件分岐ができる
# 一段（space x 4）下げることでブロックを表す
# 条件が真のときだけそのブロックが実行される
s = "good"
if s == "good":
    print("good")
elif s == "bad":
    print("bad")
else:
    print("?")
```

```
good
```

```
# if のみ、if-else のみでもOK
s = "bad"
if s == "good":
    print("good")
```

この場合は条件に合致しないので
なにも出力されいない。

制御文（２）

- for： 繰り返しのための構文

```
# for で処理を複数回繰り返せる
for i in [0, 1, 2, 3, 4]:
    print("i:", i)
```

```
i: 0
i: 1
i: 2
i: 3
i: 4
```

```
# break で途中終了できる
# 基本的に if 文と組み合わせて使う
# range(i, j) で i から j - 1 までの連番を生成できる
# 今回は [0, 1, 2, 3, 4] と同じ結果になる
for i in range(0,5):
    if i == 3:
        break
    print("i:", i)
```

```
i: 0
i: 1
i: 2
```

制御文（３）

```
# range は3つ目の引数で n 個ずつ要素を飛ばせる
for i in range(0, 20, 3):
    print(i)
```

```
0
3
6
9
12
15
18
```

```
# continue で以降の部分をスキップし次のループに入れる
# 基本的に if 文と組み合わせて使う
for i in range(0,5):
    if i == 3:
        continue
    print("i:", i)
```

```
i: 0
i: 1
i: 2
i: 4
```

```
# enumerate 関数でインデックス番号も同時に取得しながらループできる
l = ["a", "b", "c", "d", "e"]
for i, x in enumerate(l):
    print(i, x)
```

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

制御文（４）

- while： 条件が成立するまで処理を繰り返すための構文。

```
# while で条件が真の間ずっとブロック内の処理を繰り返すことが出来る
i = 0
while i < 5:
    print("i:", i)
    i += 1              # i に i + 1 した値を代入
```

```
i: 0
i: 1
i: 2
i: 3
i: 4
```

関数（１）

- 関数を定義することで、同じ処理を使いまわせる。

```
# 関数を定義することで同じ処理を何度も使いまわせる
def hello():
    print("hello, taro-san")

hello()
hello()
```

```
hello, taro-san
hello, taro-san
```

関数（２）

- 引数を定義することで、一部の値を変えた状態で同じ処理を繰り返し実行できる。

```
# 引数を定義することで一部の値を変えた状態で同じ処理を実行できる
# = でデフォルトの引数を指定することが可能
def hello(name="nanashi"):
    print("hello, %s-san" %(name))

hello("taro")
hello("hanako")
hello()
```

```
hello, taro-san
hello, hanako-san
hello, nanashi-san
```

関数（３）

- return を使うことで、値を返すことができる。
- return が呼ばれた時点で関数内の処理は終了する。

```
# return で値を返すことができる
# return 以降は実行されない
def add_10(n):
    n_10 = n + 10
    return n_10

n1 = add_10(10)
n2 = add_10(100)
print(n1, n2)
```

```
20 110
```

やってみよう（1 − 1）

- 1以上の整数を引数に与えた場合、
  1. 整数が3の倍数なら "fizz"
  2. 5 の倍数なら "buzz"
  3. 15の倍数なら "fizzbuzz"
  4. それ以外ならそのままの値

  を返す関数 fizzbuzz を作成してください。

やってみよう（1 − 2）

- 1 ～ 10000 までの素数を出力するコードを作成してください（こちらは余裕があればトライしてください）。

解答例（１－１）

```
def fizzbuzz(n):
  if n % 15 == 0:
      return "fizzbuzz"
   elif n % 5 == 0:
      return "buzz"
   elif n % 3 == 0:
      return "fizz"
   else:
      return n
```

```
for i in range(1, 31):
   print(fizzbuzz(i))
```

```
1               16
2               17
fizz            fizz
4               19
buzz            buzz
fizz            fizz
7               22
8               23
fizz            fizz
buzz            buzz
11              26
fizz            fizz
13              28
14              29
fizzbuzz        fizzbuzz
```

解答例（１－２）

- 「エラトステネスのふるい」と呼ばれるアルゴリズムを使うとこんな形で書ける。

```
# エラトステネスのふるい
p = [2]
for x in range(3, 10000, 2):
    exist = False
    for y in p:
        if x % y == 0:
            exist = True
            break
    if not exist:
        p.append(x)

print(p)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149,
151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229
, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 3
13, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401,
409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491
, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 5
99, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677,
683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787
, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 8
83, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991,
997, 1009, 1013, 1019, 1021, 1031, 1033, 1039, 1049, 1051, 1061, 1063, 10
69, 1087, 1091, 1093, 1097, 1103, 1109, 1117, 1123, 1129, 1151, 1153, 116
3, 1171, 1181, 1187, 1193, 1201, 1213, 1217, 1223, 1229, 1231, 1237, 1249
, 1259, 1277, 1279, 1283, 1289, 1291, 1297, 1301, 1303, 1307, 1319, 1321,
1327, 1361, 1367, 1373, 1381, 1399, 1409, 1423, 1427, 1429, 1433, 1439, 1
447, 1451, 1453, 1459, 1471, 1481, 1483, 1487, 1489, 1493, 1499, 1511, 15
23, 1531, 1543, 1549, 1553, 1559, 1567, 1571, 1579, 1583, 1597, 1601, 160
7, 1609, 1613, 1619, 1621, 1627, 1637, 1657, 1663, 1667, 1669, 1693, 1697
```

ビジネスにおけるデータ分析の流れ

ビジネスにおけるデータ分析の流れ

- データ分析における5つのフロー
- 現状とあるべき姿
- 問題発見
- データの収集と加工
- データ分析
- アクション

まず大前提として…

高度で複雑なモデルによる高精度な分析結果は、必ずしもそれだけでは価値が高いとは言えない。

データ分析における5つのフロー（２）

解決すべき問題に合わせて、「データ分析者」が分析方法の設計・実行ができることが大事。

- ここを誤ると分析の価値が乏しくなってしまう。
- 逆に、ここをきちんと抑えれば、本日学ぶ比較的簡単な手法による分析でも大きな成果を出すことができる。

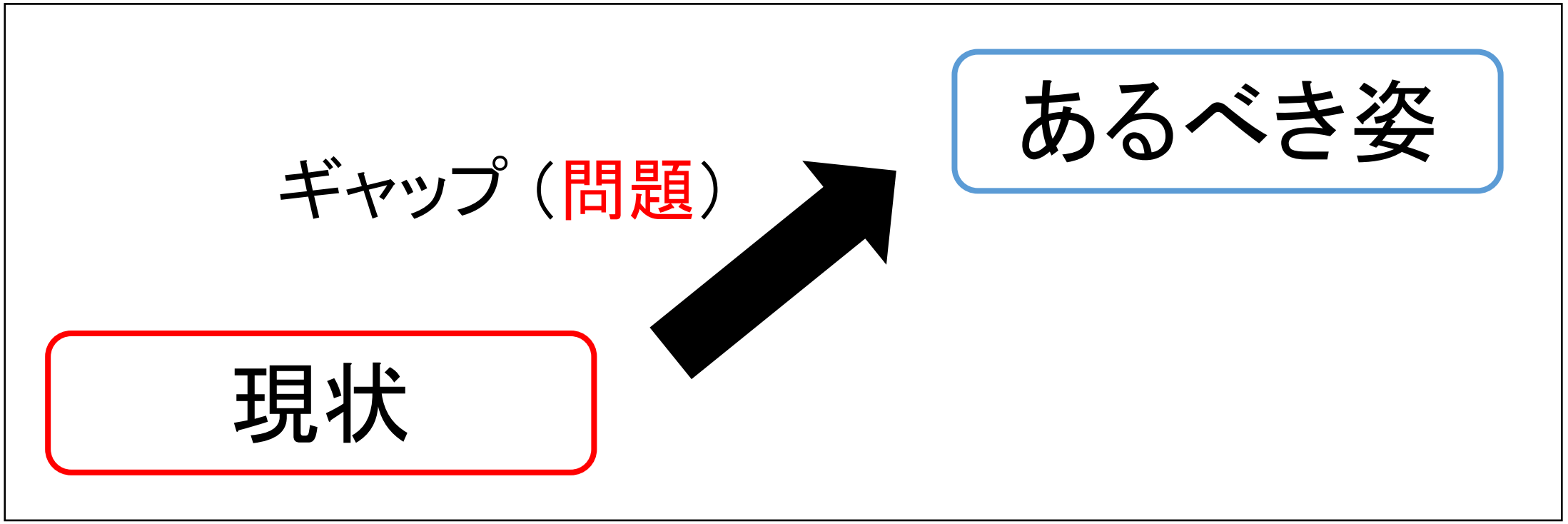
データ分析における5つのフロー（３）

1. 現状とあるべき姿の調査
2. 問題発見
3. データの収集と加工
4. データ分析
5. アクション

現状からあるべき姿に最短ルートで近づくように
問題を抽出し、解決することを目標とする。

現状とあるべき姿

- 以下の様な構造が見つかれば、そこにギャップがある。
- 本来あるべき姿と現状にギャップが合って初めて問題となる。



問題発見（１）

- 「現象」と「問題」の区別が必要

現象	現状	あるべき姿	問題か？
売上が減少した	売上比率が低い	現状でOK	問題ではない
	売上比率が高い	売上を好調時に戻す	問題
売上が増加した	広告費用が高い	広告費用を下げる	問題
	広告費用が適切	現状でOK	問題ではない

- 「現象」と「あるべき姿」から生まれる「ギャップ」がチームで共有された時、はじめてデータ分析を行う土台ができる。
- この共有されたギャップの原因を検証していく作業がデータ分析の最初のステップ。

問題発見（２）

- ギャップ（問題）を見つけるための効果的な方法
  「問題が起きていない状態＝あるべき姿をイメージする」

- 「現状」と「あるべき姿」のギャップは以下のような切り口でみることで見えてくる。
  1. 大きさを見る：　重要度の大きそうなものから
  2. 分解してみる：　要素同士の掛け算を漏れ無く作っていく
     （コントローラブルな要素を含むように）
  3. 比較してみる：　一番あるべき姿に近かった状態と比べる

データの収集と加工（１）

1. 問題を検証するために、どんなデータが必要なのか？
2. 必要なデータは、分析者が使えるところに保存されているのか？
3. 必要なデータは、分析者が申請すれば使えるようになるのか？
4. 必要なデータが保存されていない場合、新たに取得することは可能なのか？
5. 必要なデータが保存されておらず、かつ、新たに取得するのにコストがかかる場合、代用できる他のデータはないか？

基本的には、上記順番で調査していく。

データの収集と加工（２）

- データの収集先
  1. ファイル
     - Excel （各種社内資料など）
     - 生のテキストデータ（Apache のアクセスログなど）
     - CSV （DBから既にダンプしてあるデータなど）
  2. RDBMS
     - サービスで稼働中のサーバ
     - 分析用のリードレプリカ
  3. データウェアハウス
     - Hadoop 環境 （主にオンプレミス環境）
     - RedShfit、BigQuery、TreasureData （クラウド環境）

データの収集と加工（３）

- データの加工
  1. データの結合
     - ログデータとマスタデータの結合など。
  2. 判定用変数の作成
     - 課金した（1）/ 課金していない（0）など
       - このフラグを作っておけば、「課金したフラグ数 / 総ユーザー数」でさくっと課金率がでる。
  3. 変数の離散化
     - 課金額別に、ライトユーザー、ミドルユーザー、ヘビーユーザーとしてフラグ作成。
     - 年齢を10歳刻みでフラグ作成。
       - 意味のある単位毎に分析できるようになる。
       - ノイズに強くなる。

データ分析

- 意思決定支援
  - 問題解決のためのアクションを人間が決定・実行するのを支援。
  - 高度で複雑なモデルよりも、人間が理解しやすいシンプルなモデルを使う。
    - 単純集計、クロス集計、シンプルな統計分析など。

- 自動化・最適化
  - 問題解決のためのアクションをコンピューターに自動実行させる。
  - 理解のしやすさよりも、アルゴリズムの精度と計算量が重要視される。
    - 機械学習など。

アクション

- 「人間が意思決定してなにかをはじめる / やめる」
  - 上司あるいは企画職への説得コストが大きい。
- 「コンピューターに自動実行させる」
  - 開発職、サービス運用職への説得コストが大きい。
- 見込み数値の具体化、実施リスクの調査などが必要。
  - 分析時に作成した予測モデルでシミュレーション。
  - 誤差の範囲（ばらつき）の大きさの確認（許容範囲内か）。

基礎統計量による分析

この章でやる内容

- 統計学とは？
  - 記述統計学と推測統計学
  - 統計学が使えるシーン
  - 統計学の全体像
- 記述統計学
  - 記述統計学とは？
  - 真ん中を知る（平均値・中央値）
  - 構成を知る（比率）
  - ばらつきを知る（分散・標準偏差）
  - 注意点
- 推測統計学
  - 母集団と標本
  - 仮説の検証方法
  - 相関・回帰・分類・クラスタリング
  - 注意点

統計学とは？：記述統計学と推測統計学（１）

- まず次の2つのパターンを想像してみてください。

1. DATUM STUDIO の隣のカフェについて、DATUM STUDIO 全社員を調べたところ前日の利用率が『50%』だった。

2. 現在の内閣支持率は、A 新聞の調査結果によると『50%』だった。

統計学とは？ ：記述統計学と推測統計学（２）

1. DATUM STUDIO の隣のカフェについて、DATUM STUDIO 全社員を調べたところ前日の利用率が『50%』だった。

- こちらは全員に利用の有無について聞いているので、疑いようのない数字。

=> **全件調査**と呼ぶ。

全件調査を扱う統計学 ＝「**記述統計学**」

統計学とは？ ：記述統計学と推測統計学（３）

2. 現在の内閣支持率は、A 新聞の調査結果によると『50%』だった。

- 自分聞かれてないんだけど？
- 誰に聞いたの？どのぐらいの人数に聞いたの？
- 全員に聞いたときと比べてどのぐらいズレるの？

=> **標本調査**と呼ぶ。

標本調査を扱う統計学 ＝「**推測統計学**」

統計学とは？：統計学が使えるシーン

- 統計学が使える具体的なシーン
  - ある小学校の平均体重が知りたい。
  - 日本人とアメリカ人とで、どちらが体重にばらつきがあるか知りたい。
  - サラリーマンのお小遣いが東京と大阪で異なるかどうかを推測したい。
  - A内閣の支持率が都市部と農村部とで異なるかどうかを推測したい。
  - A県とB県の味噌ラーメンの平均価格が異なるかどうかを推測したい。
  - A営業所とB営業所の一部の人の営業成績の結果から、A営業所とB営業所とで営業成績の人によるムラの大きさが異なるかどうかを推測したい。
  - 年齢と好きな洋服ブランドの結びつきの強さを知りたい。
  - 広告費、ライバル店舗との距離、営業担当者数から売上を推測したい。
  - 国語、社会、理科、英語、数学の点数から、得意科目によって人を分類したい。

統計学とは？：統計学の全体像

何のための「道具」？		適用できる手法	
①要約する	真ん中を知る	平均値 / 中央値	記述統計学
	構成を知る	比率	
	ばらつきを知る	分散 / 標準偏差	
②一部分のデータから全体を断言する	断言が間違える確率を知る	検定	推測統計学
	全体の真ん中が同じ確率を知る	母平均の差の検定（t 検定）	
	全体の構成が同じ確率を知る	母比率の差の検定（χ 二乗検定）	
	全体のばらつきが同じ確率を知る	母分散の差の検定（F 検定）	
	データ同士の関係の強さを知る	相関分析	
	関係の強さを参考に、全体を分ける	クラスタリング / 因子分析 / 主成分分析 / コレスポンデンス分析	
	関係の強さを参考に、全体を当てる	回帰 / 分類	

記述統計学：記述統計学とは？（１）

- 記述統計学は「多すぎるデータを要約する」ための統計学。

```
# このようなリストを定義してみる
[10, 20, 20, 30, 20, 10]
```

  - このリストぐらいの量なら、「10が2個、20が3個、30が1個」とすべてのデータを簡単に把握できる。

記述統計学：記述統計学とは？（２）

- 次のように100要素からなるリストの場合は

[64, 90, 145, 152, 214, 93, 98, 66, 116, 129, 62, 139, 135, 127, 43, 123, 154, 21, 91, 76, 144, 138, 115, 189, 89, 86, 152, 199, 135, 55, 87, 109, 159, 96, 162, 145, 138, 70, 107, 205, 17, 102, 161, 77, 30, 66, 186, 47, 101, 229, 139, 155, 4, 59, 85, 155, 147, 152, 75, 105, 145, 40, 182, 34, 25, 122, 119, -30, 202, 142, 148, 147, -11, 74, 65, 152, 86, 53, 48, 67, 75, 118, 142, 105, 110, 186, 92, 26, 7, 93, 106, 153, 111, 99, 148, 7, 147, 115, 165, 123]

ぱっと見て解釈するのは困難 => 要約する必要がある！

- 実務上使うのはほぼ次の3つのみ。
  - 真ん中を知る（平均値・中央値）
  - 構成を知る（比率）
  - ばらつきを知る（分散・標準偏差）

記述統計学：記述統計学とは？（３）

- Pandas
  - Python でデータ解析を行うための支援ライブラリ。

  - CSV などのテーブル構造が扱いやすい。
    - DataFrame 型：　2次元テーブル構造。
    - Series 型：　DataFrame の行、列に対応する1次元の構造。
便利なリストのようなもの。

  - 集計処理に便利な関数もセットになっている。
    - 平均値、分散などが簡単に出せる。

記述統計学：真ん中を知る（１）

- 統計学で最もよく使われるのは平均値
- すべてのデータを足し合わせ、その値をデータの数で割ったもの。

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{x_1 + x_2 + x_3 + \cdots + x_n}{n}$$

x_i: データの各要素　　n: データの要素数

```
# ライブラリの読み込み
import pandas as pd

# Series 型で読み込む
#   事前にダウンロードしておいた "01_100values.csv" を
#   Jupyter-notebook で作成したファイルと同じ場所に保存しておく
s = pd.read_csv("01_100values.csv", header=None, squeeze=True)

# 平均値の計算 & 出力
print("平均値", s.mean())
```

記述統計学：真ん中を知る（２）

- このような形で平均値が出力される。

In [2]:
```
# ライブラリの読み込み
import pandas as pd

# Series 型で読み込む
#   事前にダウンロードしておいた "01_100values.csv" を
#   Jupyter-notebook で作成したファイルと同じ場所に保存しておく
s = pd.read_csv("01_100values.csv", header=None, squeeze=True)

# 平均値の計算 & 出力
print("平均値", s.mean())
```

```
平均値 107.48
```

記述統計学：真ん中を知る（３）

- データによっては平均値が真ん中を表していないことがある。

- 例えば年収。
  - 一部の人がとてつもない金額をもらっている。
  - 平均年収はそれに引きずられて高くなる傾向がある。
  - そのためほとんどの人が平均年収以下。

- 「2：8 の法則」
  - 「20%の人が全体の80%の売上を出している」
  - 80%の人が平均値以下の購買額となる。
  - ECサイト、ソーシャルゲーム など。

記述統計学：真ん中を知る（４）

- このような場合は中央値を使う。
  - データを昇順にならべたとき、ちょうど中央にある値。
  - 要素が偶数個の場合は中央の2つの値の平均値となる。

```
# 中央値の計算 & 出力
print("中央値", s.median())
```

```
In [3]: print("中央値", s.median())
中央値 109.5
```

- このサンプルデータは平均値・中央値がほぼ同じ。
  - 一部の要素がずば抜けて 大きい / 小さい 可能性は少なそう。

記述統計学：真ん中を知る（５）

- 平均と中央値のズレの実例。
  - 実サービスだと平均値は中央値の2〜3倍になることも多い。
  - 「課金ユーザーの平均課金額がXXX円だからXXX円払うユーザーをターゲットに〜」としてしまうと大幅に本来とボリューム層とズレてしまう！

- 指標にするなら中央値をメインに追いかけるほうがサービスの実情をよく表している場合が多い。

記述統計学：構成を知る（１）

- 真ん中を見た次によく行う要約が「構成を知る」。
  - ビジネスのデータでは何らかのグループごとに集計して違いを見ることが多い。
    - 「男性」と「女性」で購買傾向が違うのか？
    - 「平日」と「休日」で来店者数に違いはあるのか？
  - 次になにをすべきか検討する際の強力な材料になる。

- 構成は比率で表す。
  - データの出現回数を集計し、全データ数で割る。
  - データの合計値を集計し、全データ数で割る。

記述統計学：構成を知る（２）

- 社員の喫煙の有無から喫煙率を算出してみる。

```
# 喫煙の有無データ作成
smoking_df = pd.DataFrame(
    {"member": ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"],
     "is_smoking": [1, 1, 0, 0, 0, 0, 1, 0, 0, 1]},
    columns=["member", "is_smoking"]
)

# 喫煙データ表示
smoking_df
```

Out[5]:

	member	is_smoking
0	A	1
1	B	1
2	C	0
3	D	0
4	E	0
5	F	0
6	G	1
7	H	0
8	I	0
9	J	1

```
# 喫煙率の算出
smoking_df["is_smoking"].sum() / len(smoking_df.index)
```

Out[6]: 0.4

記述統計学：ばらつきを知る（１）

- 分散
  - どの程度値がばらついているのかを表す。
  - データの各要素に対して、平均値からどれぐらい離れているのかを計算し、その値を２乗したものの平均。

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 = \frac{(x_1-\mu)^2 + (x_2-\mu)^2 + \cdots + (x_n-\mu)^2}{n}$$

x_i: **データの各要素**

n: **データの要素数**

μ: **平均値**

- 標準偏差
  - 分散の平方根。
  - 分散は２乗しているので元のデータと単位が異なってしまうので、平方根をとり単位を元に戻す。

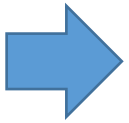
$$\sigma = \sqrt{\sigma^2}$$

記述統計学：ばらつきを知る（2）

- 今回はあやめの花（Iris）のデータをサンプルとして使用する。
  - SepalLength：がく片の長さ
  - SepalWidth：　がく片の幅
  - PetalLength：花びらの長さ
  - PetalWidth：　花びらの幅
  - Name：　　品種名

- まずデータを読み込んでみる。

```
# Iris データの読み込み
iris_df = pd.read_csv("03_iris.csv")

# データの先頭5行を確認
iris_df.head()
```



Out[7]:

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

記述統計学：ばらつきを知る（3）

- 分散算出のプログラム

```
# Iris データを「名前でグループ化」->「分散を計算」
iris_df.groupby("Name").var()
```

Out[8]:

	SepalLength	SepalWidth	PetalLength	PetalWidth
Name				
Iris-setosa	0.124249	0.145180	0.030106	0.011494
Iris-versicolor	0.266433	0.098469	0.220816	0.039106
Iris-virginica	0.404343	0.104004	0.304588	0.075433

- 標準偏差算出のプログラム

```
# Iris データを「名前でグループ化」->「標準偏差を計算」
iris_df.groupby("Name").std()
```

Out[9]:

	SepalLength	SepalWidth	PetalLength	PetalWidth
Name				
Iris-setosa	0.352490	0.381024	0.173511	0.107210
Iris-versicolor	0.516171	0.313798	0.469911	0.197753
Iris-virginica	0.635880	0.322497	0.551895	0.274650

記述統計学（14）

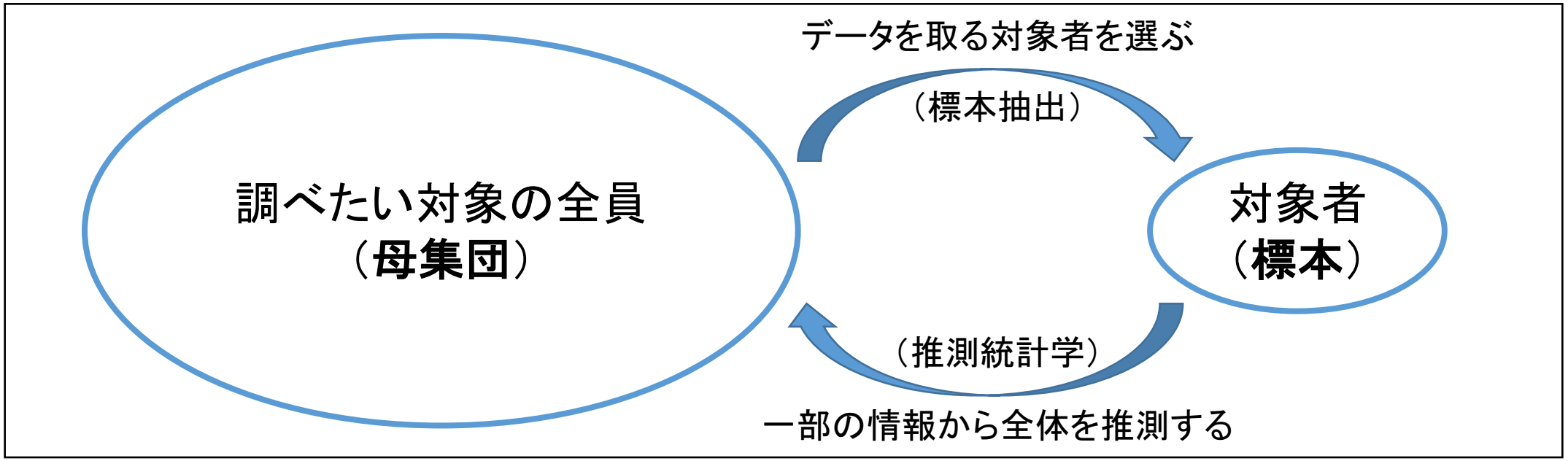
- 記述統計学による「要約」の注意点
  - 「要約」には情報を捨てていく作業が含まれている。
  - 「要約」したものにはきれいな情報が多くなりやすい。
  - 「要約」したものは分析する前からわかっている結果になりやすい。
  - 「要約」で捨てられる情報に、ビジネスのヒントがあることも多い。

全体の傾向がわかる ≒ 貴重な少数意見を捨てる

推測統計学：母集団と標本

- 一部分の情報から調べたい対象の全体を推測するのが推測統計学。
  - 全員を調べるのが理想だが、時間的にも金銭的にも不可能な場合。

- 一部分の取り方によって結果が大きく変わってしまう。
  - 調べたい全対象の精巧なミニチュアになっていること（**ランダム抽出**）が大事。



推測統計学：仮説の検証方法（１）

- 「ある原因」が「ある結果」に影響する

1. 経験や既知の情報を参考に因果関係を考える。
2. 因果関係について、情報把握できるようにデータを集める。
3. 集めたデータを見る。
4. 集めたデータを見る限り、想像したよう因果関係がありそうだ。

上記条件が満たされた場合のみ、「道具」を使い、
調べたい調査対象の全員について調べていく。

推測統計学：仮説の検証方法（２）

- 仮説の４パターン

		結果	
		種類	**数量**
原因	**種類**	性別が男性 -> 利用意欲が高い	広告を打つ -> ユーザー数が増加する
	数量	価格が下がる -> 利用意欲が上がる	広告数が増える -> 認知率が増加する

推測統計学：仮説の検証方法（３）

- 仮説を考える方法

		結果	
		種類	数量
原因	種類	クロス集計表	種類データ毎の真ん中、 ばらつきを比較
	数量	散布図	散布図

推測統計学：仮説の検証方法（４）

- 仮説を検定する方法

		結果	
		種類	数量
原因	種類	χ二乗検定	t検定、F検定
	数量	判別分析 ロジスティック回帰分析 係数の有意確率を利用	相関分析 回帰分析 係数の有意確率を利用

推測統計学：仮説の検証方法（５）

- 断言が間違える確率を知る：検定
  - 一部のデータであることによるブレを考慮しても
    このデータ間に差はあるといえるか？

- 平均の差を検定したい => t 検定
- 比率の差を検定したい => χ 二乗検定
- 分散の差を検定したい => F 検定

推測統計学： 仮説の検証方法（６）

- t 検定を行なってみる

```
# 科学技術計算用ライブラリ scipy を利用する
from scipy import stats

# t 検定
setosa_petal_width = iris_df[iris_df.Name == "Iris-setosa"].PetalWidth
virginica_petal_width = iris_df[iris_df.Name == "Iris-virginica"].PetalWidth

# t値、p値 を取得
t, p = stats.ttest_rel(setosa_petal_width,  virginica_petal_width)
print("t:", t, "p:", p)

# p < 0.05 なら有意とする
if p < 0.05:
    print("有意な差があります")
else:
    print("有意な差がありません")
```

$$t\text{ 値} = \frac{\text{標本平均の差}}{\text{標本平均の差の標準誤差}}$$

p 値：
仮説（帰無仮説）が間違う確率。
0.05 or 0.01 を基準にする場合が多い。

```
t: -44.5852727142
p: 2.51205884218e-41
有意な差があります
```

推測統計学：相関・回帰・分類・クラスタリング

- 相関
  - ２つの変数の間の直線的な関係を数値化したもの。
- 回帰
  - 因果関係がある変数の組み合わせを利用して、ある変数（1 ～ n 個）から特定の変数の将来的な値を予測する。
- 分類
  - ある要素が既知のクラス(2 ～ n 種類)のどれに分類されるか予測する。
- クラスタリング
  - 与えられた要素のうち、似ている物同士を自動的にまとめる。

これらは実務でよく使うので、後の章で詳しく解説します。

推測統計学：注意点

- 検定では「原因」と「結果」との厳密な因果分析はできない。
  - にわとりが先か、卵が先か。
- データが少ないと役に立たない。
- 検定結果が（数学的に）正しいとまでは言えない。
  - 差があることまでしかわからない。

取得できたデータから全体について断言する際に、対象者が偏っているリスクを踏まえたうえで、全体への断言を間違える可能性を定量化するときに使う「道具」。

やってみよう（２－１）

1. "02_10000values.csv" を Series 形式で読み込む。
2. 平均値を算出する。
3. 中央値を算出する。

やってみよう（２－２）

1. "03_iris.csv" を DataFrame 形式で読み込む。
2. 品種別の平均値を計算する。
3. 品種別の中央値を計算する。
4. 品種別の分散を計算する。
5. 品種別の標準偏差を算出する。

やってみよう（２－３）

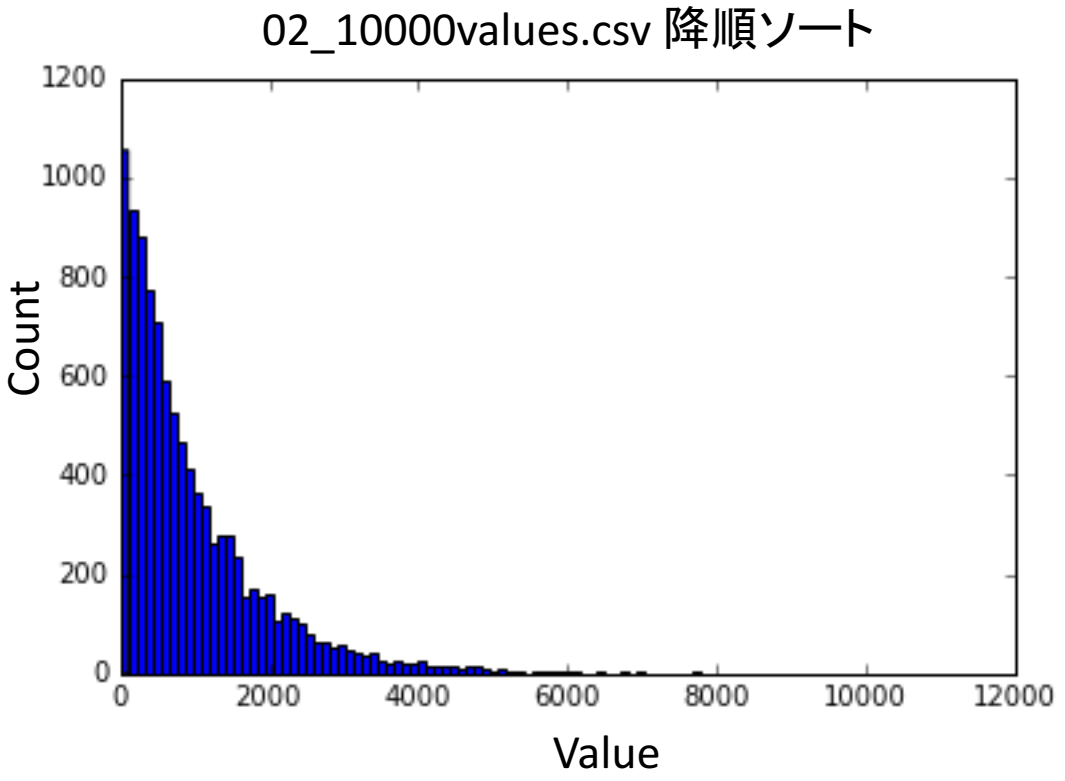
1. "03_iris.csv" を DataFrame 形式で読み込む。
2. Iris-versicolor の SepalWidth 列を Series 形式で取り出し、変数に代入する。
3. Iris-virginica の SepalWidth 列を Series 形式で取り出す、変数に代入する
4. 2. と 3. で取り出した２つの列に対して t 検定を実施する。
5. 有意差があるかどうか確かめる。

解答例（２－１）

```
In [4]: s = pd.read_csv("02_10000values.csv", header=None, squeeze=True)
        print("平均値", s.mean())
        print("中央値", s.median())
```

```
平均値 983.9983804474648
中央値 659.0869630605753
```

指数分布（右のグラフ）になっている場合は、このように平均値、中央値がズレる。



解答例（２－２）

先ほど計算しなかったところのみ

In [15]:
```
# やってみよう
iris_df.groupby("Name").mean()
```

Out[15]:

	SepalLength	SepalWidth	PetalLength	PetalWidth
Name				
Iris-setosa	5.006	3.418	1.464	0.244
Iris-versicolor	5.936	2.770	4.260	1.326
Iris-virginica	6.588	2.974	5.552	2.026

In [16]:
```
iris_df.groupby("Name").median()
```

Out[16]:

	SepalLength	SepalWidth	PetalLength	PetalWidth
Name				
Iris-setosa	5.0	3.4	1.50	0.2
Iris-versicolor	5.9	2.8	4.35	1.3
Iris-virginica	6.5	3.0	5.55	2.0

解答例（２－３）

```
# t 検定
versicolor_sepal_width = iris_df[iris_df.Name == "Iris-versicolor"].SepalWidth
virginica_sepal_width = iris_df[iris_df.Name == "Iris-virginica"].SepalWidth

# t値、p値 を取得
t, p = stats.ttest_rel(
    versicolor_sepal_width,
    virginica_sepal_width
)

print("t:", t)
print("p:", p)

# p < 0.05 なら有意
if p < 0.05:
    print("有意な差があります")
else:
    print("有意な差がありません")
```

```
t: -3.07552983666
p: 0.00343220956329
有意な差があります
```

Pandas の使い方

Pandas の使い方

- Series・DataFrame の作成
- データの入力
- データの確認
- Column・Index の操作
- データの抽出
- データのソート
- データの結合
- 集約処理
- その他よくつかう操作
- データの出力

Series・DataFrame の作成（１）

- リストからSeries の作成
  - Series は Pandas に存在する、集計のための機能を色々付加した高機能リストのようなもの。
  - 通常のリストから変換することが出来る。

```
# Pandas の読み込み
import pandas as pd

# Series の作成（リストから）
l = [1, 2, 3, 4, 5]
s = pd.Series(l)
s
```

```
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

Series・DataFrame の作成（２）

- リスト・Series から DataFrame 作成。
  - DataFrame は Pandas に存在する、集計のための機能を色々付加したテーブル構造を扱うための道具。

```
# DataFrame の作成（２重のリストから）
m = [
    [1, "yamada", "taro", "man"],
    [2, "yamada", "hanako", "woman"],
    [3, "suzuki", "ichiro", "man"],
    [4, "suzuki", "jiro", "man"],
    [5, "suzuki", "saburo", "man"]
]

names = ["id", "family_name", "first_name", "gender"]

# columns 引数でカラム名を指定できる
name_df = pd.DataFrame(m, columns=names)
name_df
```

	id	family_name	first_name	gender
0	1	yamada	taro	man
1	2	yamada	hanako	woman
2	3	suzuki	ichiro	man
3	4	suzuki	jiro	man
4	5	suzuki	saburo	man

Series・DataFrame の作成（３）

- ディクショナリから DataFrame 作成
  - ディクショナリから変換することも出来る。

```
# DataFrame の作成（ディクショナリから）
d = {
    "id": [1, 2, 3, 4, 5],
    "is_smoking": [True, False, True, False, True],
    "gender": ["m", "m", "m", "f", "f"]
}

names = ["id", "is_smoking", "gender"]

# columns 引数でカラムの順番を指定できる
# 指定しないと順番は保証されない
smoking_df = pd.DataFrame(d, columns=names)
smoking_df
```

	id	is_smoking	gender
0	1	True	m
1	2	False	m
2	3	True	m
3	4	False	f
4	5	True	f

データの入力

- Series 形式で読み込み（1カラムのみの場合）

```
# n 行 × 1 列のデータを Series 形式で読み込む
# header がなければ None を指定する
s = pd.read_csv("02_10000values.csv", header=None, squeeze=True)
```

- DataFrame 形式で読み込み

```
# n 行 × m 列のデータを DataFrame 形式で読み込む
#   sep: 区切り文字の指定。デフォルトは ","
df = pd.read_csv("03_iris.csv", sep=",")
```

データの確認（１）

- head メソッド

```
# head() で先頭6行表示
df.head()
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

```
# head(n) で 先頭 n 行表示
df.head(1)
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
0	5.1	3.5	1.4	0.2	Iris-setosa

- tail メソッド

```
# tail() で末尾から表示もできる
df.tail(3)
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

データの確認（２）

- カラム・インデックスの情報の確認

```
# カラム名を取得
df.columns
```

```
Index(['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth
', 'Name'], dtype='object')
```

```
# インデックス名を取得
df.index
```

```
Int64Index([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,
            ...
            140, 141, 142, 143, 144, 145, 146, 147, 148, 149],
           dtype='int64', length=150)
```

- 行数・列数の確認

```
# 列数の確認
len(df.columns)
```

```
5
```

```
# 行数の確認
len(df.index)
```

```
150
```

データの確認（３）

- 基礎統計量の確認

```
# describe() で基礎統計量の確認
df.describe()
```

	SepalLength	SepalWidth	PetalLength	PetalWidth
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

Column・Index の操作

- 名称の書き換え

```
# インデックスを同じ長さの上書き
#（カラムも同様の方法で上書きできる）
df.index = range(101, 251)
df.index
```

```
Int64Index([101, 102, 103, 104, 105, 106, 107, 108, 109, 110,
            ...
            241, 242, 243, 244, 245, 246, 247, 248, 249, 250],
           dtype='int64', length=150)
```

データの抽出（１）

- 列名で Series としてアクセス

```
# 特定の1列を Series 型で取得
df["SepalLength"].head()
```

```
101    5.1
102    4.9
103    4.7
104    4.6
105    5.0
Name: SepalLength, dtype: float64
```

データの抽出（2）

- 複数列名で DataFrame としてアクセス

```
# 1 〜 n 列を DataFrame 型で取得
df[["SepalLength", "SepalWidth"]].head()
```

	SepalLength	SepalWidth
101	5.1	3.5
102	4.9	3.0
103	4.7	3.2
104	4.6	3.1
105	5.0	3.6

データの抽出（３）

- ix を使った列選択

```
# ix を使えば行番号・列番号・列名・列番号いずれかで指定できる
df.ix[110:115, ["SepalLength", "PetalLength"]]
```

	SepalLength	PetalLength
110	4.9	1.5
111	5.4	1.5
112	4.8	1.6
113	4.8	1.4
114	4.3	1.1
115	5.8	1.2

データの抽出（４）

- [ブール値]による抽出

```
# 条件文でデータのフィルタリング
# df[ ブール値のリスト ] という形で指定する
df[df.SepalLength > 7.5]
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
206	7.6	3.0	6.6	2.1	Iris-virginica
218	7.7	3.8	6.7	2.2	Iris-virginica
219	7.7	2.6	6.9	2.3	Iris-virginica
223	7.7	2.8	6.7	2.0	Iris-virginica
232	7.9	3.8	6.4	2.0	Iris-virginica
236	7.7	3.0	6.1	2.3	Iris-virginica

データの抽出（５）

- query メソッドによる抽出

```
# query メソッドを使うと列名をそのまま文字列で書ける
df.query("Name == 'Iris-virginica'").head()
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
201	6.3	3.3	6.0	2.5	Iris-virginica
202	5.8	2.7	5.1	1.9	Iris-virginica
203	7.1	3.0	5.9	2.1	Iris-virginica
204	6.3	2.9	5.6	1.8	Iris-virginica
205	6.5	3.0	5.8	2.2	Iris-virginica

データの抽出（6）

- query メソッドでは変数を埋め込むことが出来る

```
# query メソッドは @ で変数を埋め込むこともできる
name = "Iris-versicolor"
df.query("Name == @name").head()
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
151	7.0	3.2	4.7	1.4	Iris-versicolor
152	6.4	3.2	4.5	1.5	Iris-versicolor
153	6.9	3.1	4.9	1.5	Iris-versicolor
154	5.5	2.3	4.0	1.3	Iris-versicolor
155	6.5	2.8	4.6	1.5	Iris-versicolor

データの抽出（7）

- query メソッドでは index を指定することもできる

```
# index を指定することも出来る
df.query("index in [130, 131, 132]")
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
130	4.7	3.2	1.6	0.2	Iris-setosa
131	4.8	3.1	1.6	0.2	Iris-setosa
132	5.4	3.4	1.5	0.4	Iris-setosa

データのソート（１）

- ある一つの列の値による並べ替え

```
# sort_values でデータをソートできる
df.sort_values(by=["SepalLength"]).head()
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
114	4.3	3.0	1.1	0.1	Iris-setosa
143	4.4	3.2	1.3	0.2	Iris-setosa
139	4.4	3.0	1.3	0.2	Iris-setosa
109	4.4	2.9	1.4	0.2	Iris-setosa
142	4.5	2.3	1.3	0.3	Iris-setosa

データのソート（2）

- 複数列の値による並べ替え

```
# 複数要素でソートできる
df.sort_values(by=["PetalLength", "PetalWidth"]).head()
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
123	4.6	3.6	1.0	0.2	Iris-setosa
114	4.3	3.0	1.1	0.1	Iris-setosa
115	5.8	4.0	1.2	0.2	Iris-setosa
136	5.0	3.2	1.2	0.2	Iris-setosa
103	4.7	3.2	1.3	0.2	Iris-setosa

データのソート（3）

- 降順ソート

```
# 降順ソート
df.sort_values(by=["PetalLength", "PetalWidth"], ascending=False).head()
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
219	7.7	2.6	6.9	2.3	Iris-virginica
218	7.7	3.8	6.7	2.2	Iris-virginica
223	7.7	2.8	6.7	2.0	Iris-virginica
206	7.6	3.0	6.6	2.1	Iris-virginica
232	7.9	3.8	6.4	2.0	Iris-virginica

データの結合（1）

- concat による縦方向の単純結合

```
# 2つの DataFrame を concat メソッドで縦方向に結合する
smoking1_df = pd.DataFrame(
    {"member": ["A", "B", "C", "D", "E"],
     "is_smoking": [1, 1, 0, 0, 0]},
    columns=["member", "is_smoking"]
)

smoking2_df = pd.DataFrame(
    {"member": ["F", "G", "H", "I", "J"],
     "is_smoking": [0, 1, 0, 0, 1]},
   columns=["member", "is_smoking"]
)

smoking_df = pd.concat([smoking1_df, smoking2_df])
smoking_df
```

	member	is_smoking
0	A	1
1	B	1
2	C	0
3	D	0
4	E	0
0	F	0
1	G	1
2	H	0
3	I	0
4	J	1

データの結合（2）

- concat による横方向の単純結合

```
# 2 つの DataFrame を横方向に結合する (axis = 1 を指定)
smoking3_df = pd.DataFrame(
    {"gender": ["male", "female", "male", "male", "female"],
    "age": [24, 55, 30, 42, 28]},
    columns=["gender", "age"]
)

smoking_df = pd.concat([smoking1_df, smoking3_df], axis=1)
smoking_df
```

	member	is_smoking	gender	age
0	A	1	male	24
1	B	1	female	55
2	C	0	male	30
3	D	0	male	42
4	E	0	female	28

データの結合（3）

- merge による内部結合

```
df1 = pd.DataFrame(
    {'A': ['A0', 'A1', 'A2', 'A3'],
     'B': ['B0', 'B1', 'B2', 'B3'],
     'C': ['C0', 'C1', 'C2', 'C3'],
     'D': ['D0', 'D1', 'D2', 'D3']},
    index=[0, 1, 2, 3]
)

df1
```

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

```
df2 = pd.DataFrame(
    {'B': ['B2', 'B3', 'B6', 'B7'],
     'F': ['F2', 'F3', 'F6', 'F7']},
    index=[2, 3, 6, 7]
)

df2
```

	B	F
2	B2	F2
3	B3	F3
6	B6	F6
7	B7	F7

```
# B 列 をキーにして INNER JOIN
inner_df = pd.merge(df1, df2, on=["B"], how="inner")
inner_df
```

	A	B	C	D	F
0	A2	B2	C2	D2	F2
1	A3	B3	C3	D3	F3

データの結合（4）

- merge による外部結合

```
# id をキーにして LEFT JOIN
# 値がない所は NaN になる
left_df = pd.merge(df1, df2, on=["B"], how="left")
left_df
```

	A	B	C	D	F
0	A0	B0	C0	D0	NaN
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	F2
3	A3	B3	C3	D3	F3

データの集約処理（１）

- 基本的な集計

```
iris_df = pd.read_csv("03_iris.csv")

# 平均 .mean()
iris_df.mean()
```

```
SepalLength    5.843333
SepalWidth     3.054000
PetalLength    3.758667
PetalWidth     1.198667
dtype: float64
```

```
# 中央値 .median()
iris_df.median()
```

```
SepalLength    5.80
SepalWidth     3.00
PetalLength    4.35
PetalWidth     1.30
dtype: float64
```

```
# 分散 .var()
iris_df.var()
```

```
SepalLength    0.685694
SepalWidth     0.188004
PetalLength    3.113179
PetalWidth     0.582414
dtype: float64
```

```
# 標準偏差 .std()
iris_df.std()
```

```
SepalLength    0.828066
SepalWidth     0.433594
PetalLength    1.764420
PetalWidth     0.763161
dtype: float64
```

```
# 合計 .sum()
iris_df.sum()
```

```
SepalLength                                                876.5
SepalWidth                                                 458.1
PetalLength                                                563.8
PetalWidth                                                 179.8
Name           Iris-setosaIris-setosaIris-setosaIris-setosaIr...
dtype: object
```

```
# カウント .count()
iris_df.count()
```

```
SepalLength    150
SepalWidth     150
PetalLength    150
PetalWidth     150
Name           150
dtype: int64
```

データの集約処理（２）

- groupby による集計

```
# グループ別にカウント
iris_df.groupby("Name").count()
```

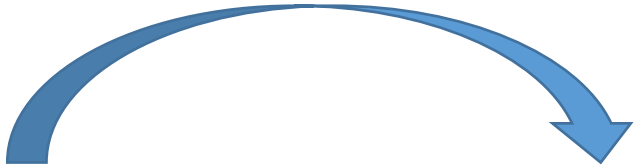
	SepalLength	SepalWidth	PetalLength	PetalWidth
Name				
Iris-setosa	50	50	50	50
Iris-versicolor	50	50	50	50
Iris-virginica	50	50	50	50

データの集約処理（３）

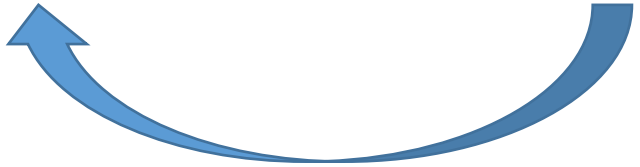
- クロス集計（データの縦持ち・横持ち変換）

性別	政党 A を支持する？
男	支持する
女	支持しない
男	支持する
女	支持する
男	支持する
女	支持しない

横持ち形式



性別	支持する	支持しない
男	3	0
女	1	2



縦持ち形式

データの集約処理（４）

```
# クロス集計のためのデータ作成
iris_df["idx"] = iris_df.index.to_series() % 3
iris_df.head(7)
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name	idx
0	5.1	3.5	1.4	0.2	Iris-setosa	0
1	4.9	3.0	1.4	0.2	Iris-setosa	1
2	4.7	3.2	1.3	0.2	Iris-setosa	2
3	4.6	3.1	1.5	0.2	Iris-setosa	0
4	5.0	3.6	1.4	0.2	Iris-setosa	1
5	5.4	3.9	1.7	0.4	Iris-setosa	2
6	4.6	3.4	1.4	0.3	Iris-setosa	0

```
iris_df.pivot_table(
   ["SepalLength", "SepalWidth"], # 集計する変数の指定
   aggfunc="mean",                # 集計の仕方の指定
   fill_value=0,  # 該当する値がない場合の埋め値の指定
   index="idx",         # 行方向に残す変数の指定
   columns="Name" # 列方向に展開する変数の指定
)
```

	SepalLength			SepalWidth		
Name	Iris-setosa	Iris-versicolor	Iris-virginica	Iris-setosa	Iris-versicolor	Iris-virginica
idx						
0	5.052941	5.770588	6.756250	3.470588	2.676471	2.981250
1	5.011765	6.018750	6.447059	3.417647	2.906250	3.023529
2	4.950000	6.023529	6.570588	3.362500	2.735294	2.917647

その他よく使う操作

- ユニーク化

```
# ユニークにする
no_dup_df = iris_df.drop_duplicates()
print("そのまま", len(iris_df))
print("重複なし", len(no_dup_df))
```

```
そのまま 150
重複なし 147
```

- NaN を補完

```
# NaN を補完
left_df.fillna("Fill")
```

	A	B	C	D	F
0	A0	B0	C0	D0	Fill
1	A1	B1	C1	D1	Fill
2	A2	B2	C2	D2	F2
3	A3	B3	C3	D3	F3

- NaN を削除

```
# NaN が含まれている行を削除
left_df.dropna()
```

	A	B	C	D	F
2	A2	B2	C2	D2	F2
3	A3	B3	C3	D3	F3

データの出力（1）

- ファイルへ

```
# CSV として出力
left_df.to_csv(
    "left_df.csv",              # ファイルのパス
    sep=",",                    # 区切り文字
    header=True,                # header 行を出力するかどうか
    index=False,                # index 列を出力するかどうか
    encoding="utf-8",           # 文字コードの指定
    line_terminator="¥n"        # 改行文字
)
```

データの出力（2）

- DBへ（参考まで）

```
# (参考) MySQL へ出力
from pandas.io import sql
import pymysql
con = pymysql.connect(
    host="localhost",
    user="root",
    passwd="your_password",
    db="your_db_name"
)
sql.write_frame(
    df,                         # 書き込みたい DataFrame
    con=con,                    # コネクションオブジェクト
    name='table_name',          # テーブル名
    if_exists='replace',        # "fail", "replace" or "append"
    flavor='mysql'
)
con.close()
```

データの可視化

データの可視化

- 使用するグラフの選択。
- ヒストグラムの作成。
- 棒グラフの作成。
- 散布図の作成。
- 折れ線グラフの作成。

使用するグラフの選択

- 1変数のみ
  - 要素を数え上げたい -> ヒストグラム

- 2変数
  - 種類と数量の関係性を見たい -> 棒グラフ
  - 数量同士の関係性を見たい -> 散布図
  - 時間と数量の関係性を見たい -> 折れ線グラフ

ヒストグラムの作成（１）

- 対象が1変数のみで、要素の数え上げを行いたい場合
  ＝> ヒストグラム

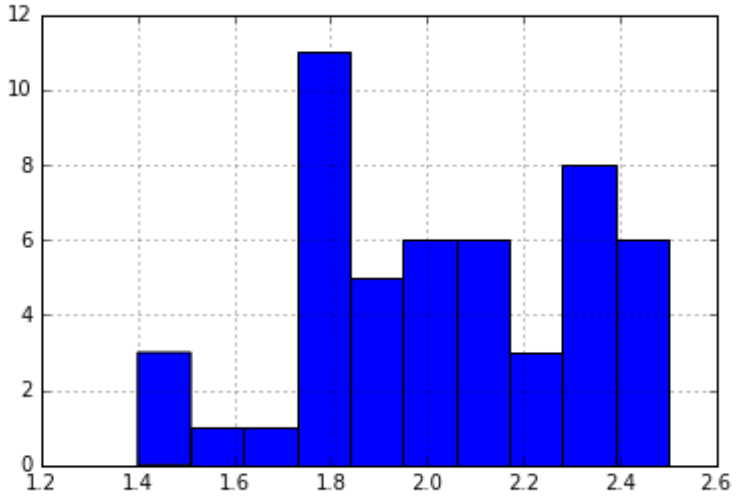
- 数量の場合
  - 適切な区間（レンジ）に区切り、区間毎に要素の数を数え上げます。

- 種類の場合
  - 種類毎に要素を数え上げます。

ヒストグラムの作成（２）

- 実際に描いてみよう

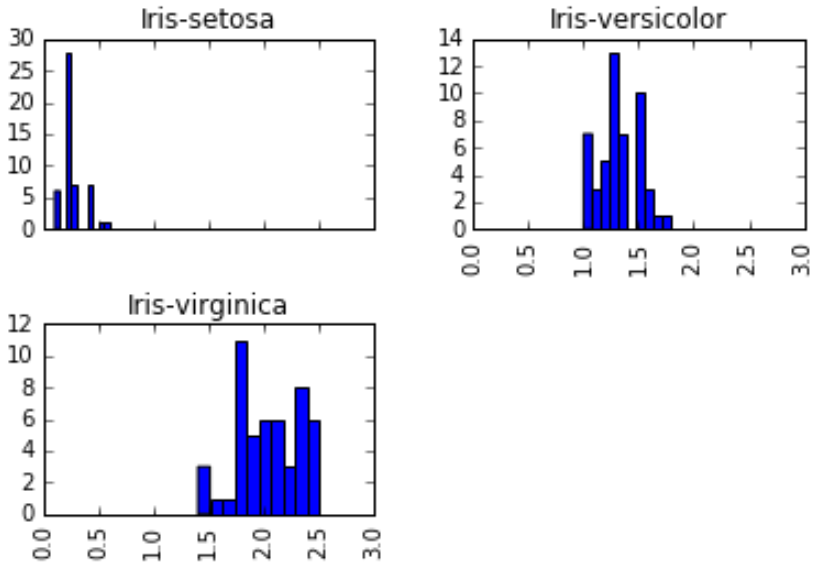
```
# グラフのインライン表示命令 & 必要なライブラリの読み込み
%matplotlib inline
import matplotlib.pyplot as plt

# Iris-virginica の PetalWidth をプロットしてみる
virginica_petal_width = iris_df[iris_df.Name == "Iris-virginica"].PetalWidth
virginica_petal_width.hist()
```



ヒストグラムの作成（３）

- Iris 3種類同時にPetalWidth をプロットしてみる（別グラフに）

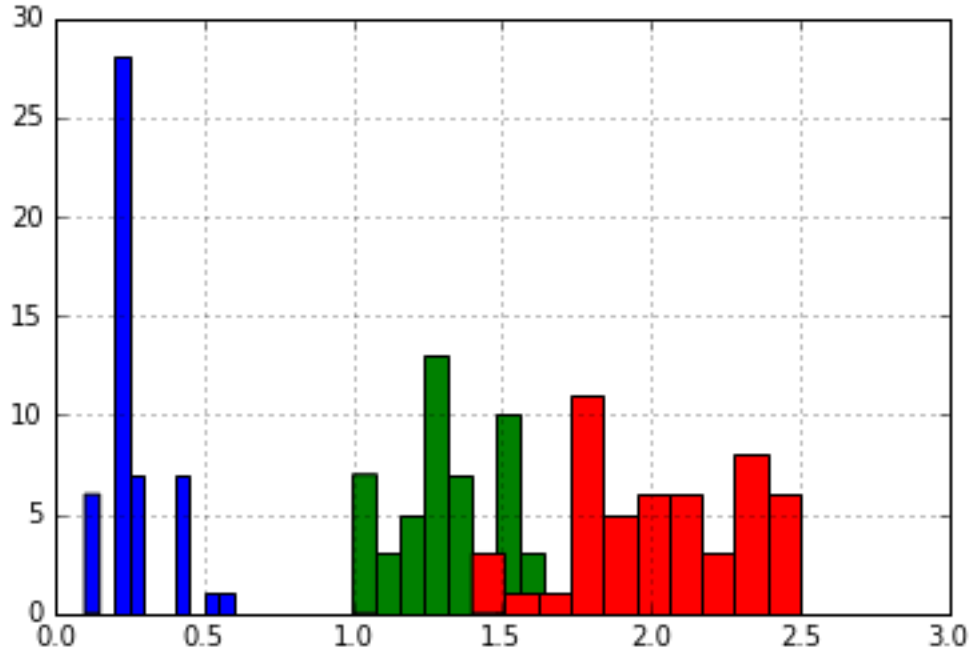
```
# PetalWidth と Name からなるデータフレームのサブセットを作成
petal_width_df = iris_df[["PetalWidth", "Name"]]

# hist メソッド by 引数を指定するとそれぞれ別エリアにプロットできる
# また sharex 引数で同じレンジの x 軸を使用するかどうか指定できる
petal_width_df.hist(by="Name", sharex=True)
```



ヒストグラムの作成（４）

- Iris 3種類同時にPetalWidth をプロットしてみる（同じグラフに）

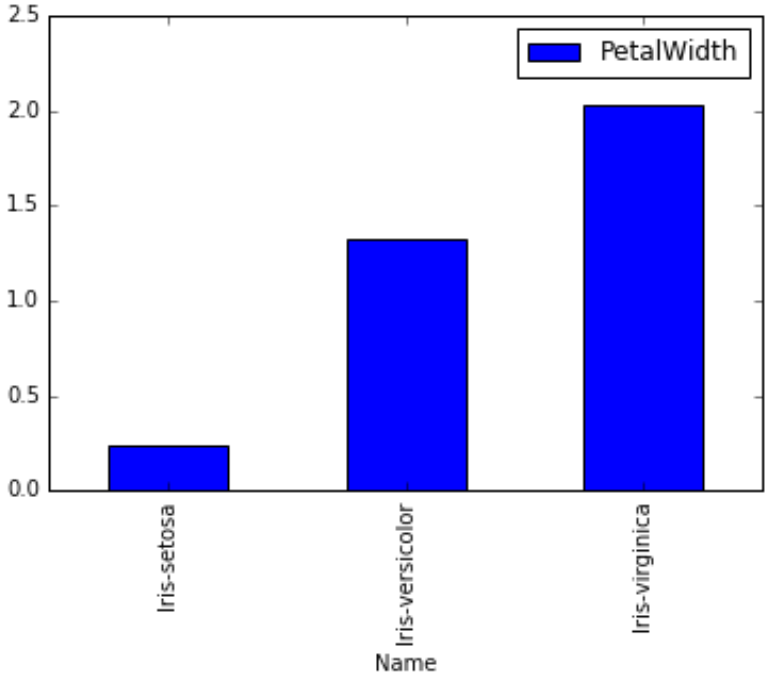
```
# groupby 後に 要素を選択した後 hist メソッドを使用する
petal_width_df.groupby("Name").PetalWidth.hist()
```



棒グラフの作成（１）

- 種類と数量の関係性を見たい場合は棒グラフを利用する。
- 基本的に種類を x 軸（横軸）、数量を y 軸（縦軸）に取る。

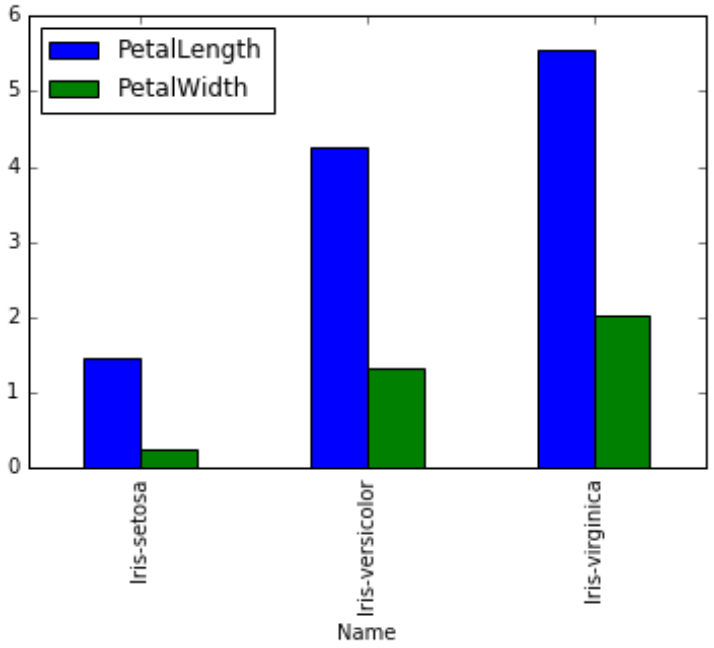
```
# 品種別 PetalWidth 平均値の比較
# kind 引数に bar を指定することで棒グラフを描ける
petal_width_df.groupby("Name").mean().plot(kind="bar")
```



棒グラフの作成（2）

- 複数要素でも同じようにプロットできる。

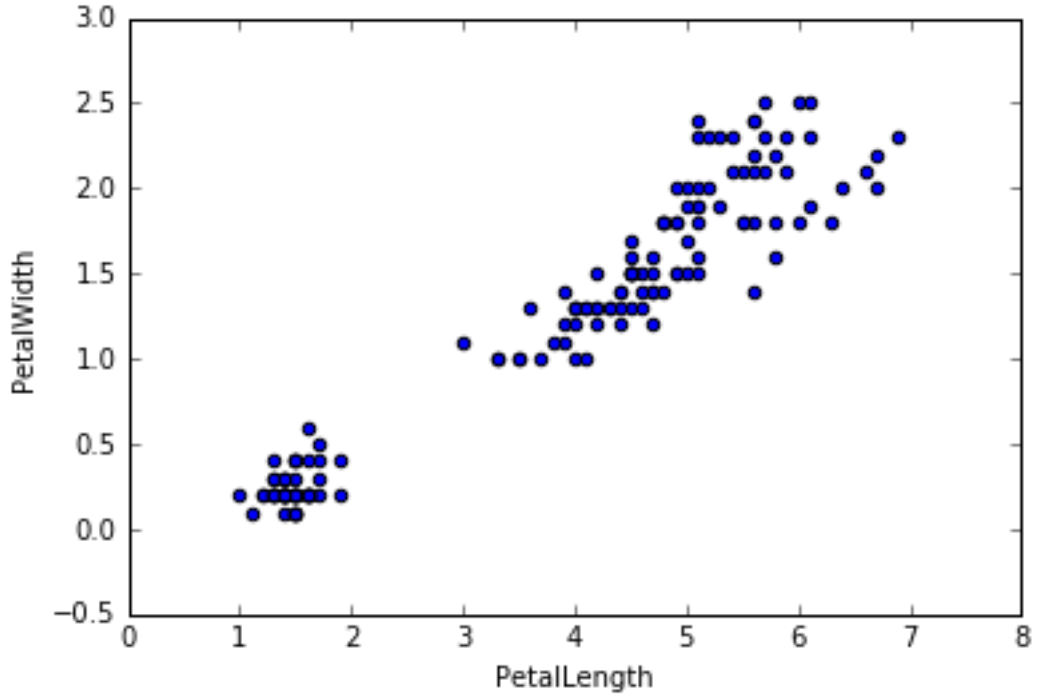
```
# PetalLength、PetalWidth と Name からなるデータフレーム作成
petal_df = iris_df[["PetalLength", "PetalWidth", "Name"]]

# 品種別 PetalLength, PetalWidth 平均値の比較
petal_df.groupby("Name").mean().plot(kind="bar")
```



散布図の作成（１）

- 数量同士の関係性を見たい場合は散布図を利用する。

```
# kind 引数に scatter を指定することで散布図を描ける
# x, y それぞれに x 軸要素、y 軸要素の名称を指定する
petal_df.plot(kind="scatter", x="PetalLength", y="PetalWidth")
```

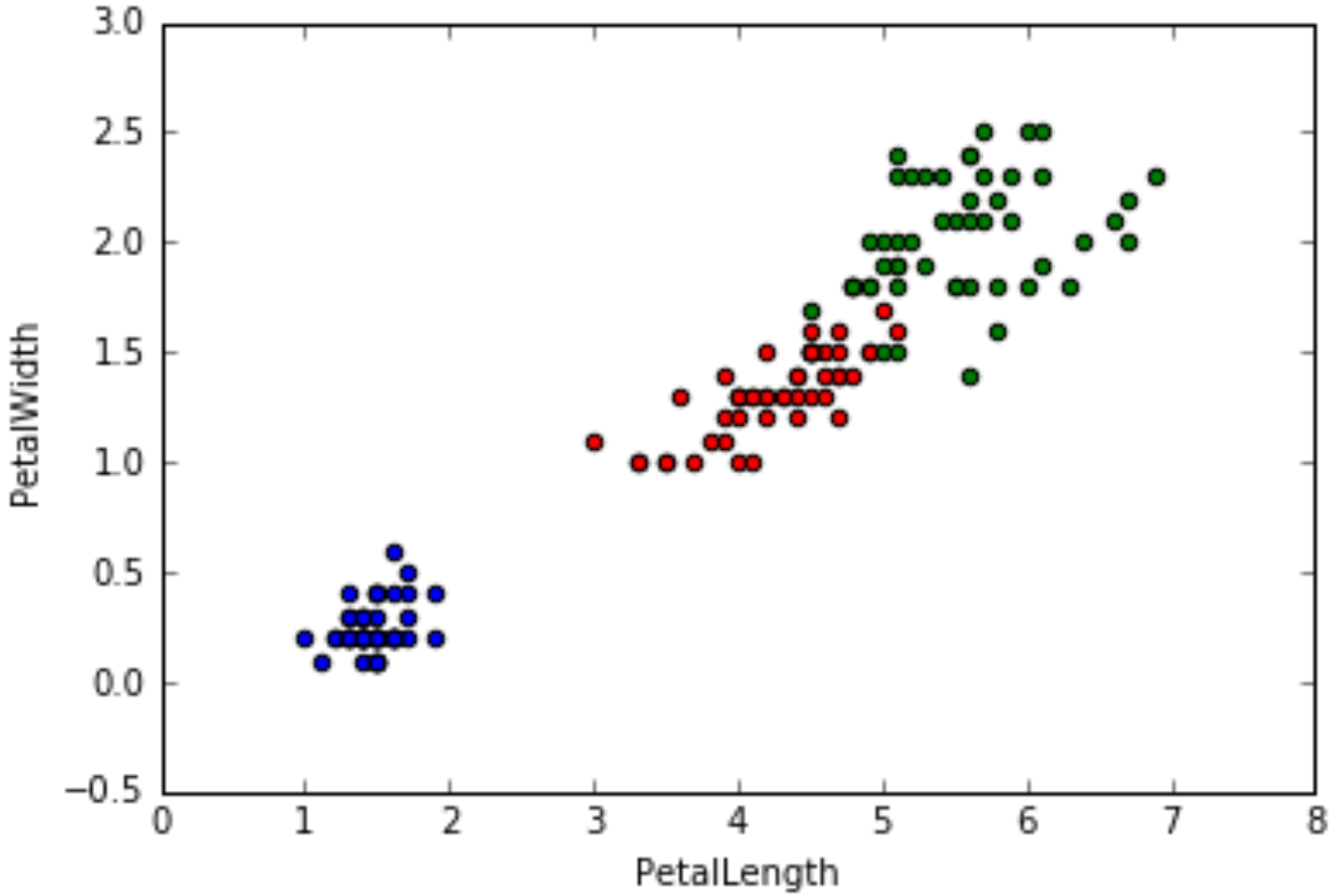


散布図の作成（２）

- ３品種別に描いてみる

```
# 3品種別に描いてみる
p_setosa_df = petal_df[petal_df.Name == "Iris-setosa"]
p_versicolor_df = petal_df[petal_df.Name == "Iris-versicolor"]
p_virginica_df = petal_df[petal_df.Name == "Iris-virginica"]

# ax を次の plot 時に渡せば同じグラフに重ねて描ける
ax = p_setosa_df.plot(kind="scatter", x="PetalLength", y="PetalWidth", color="blue")
ax = p_versicolor_df.plot(kind="scatter", x="PetalLength", y="PetalWidth", color="red", ax=ax)
p_virginica_df.plot(kind="scatter", x="PetalLength", y="PetalWidth", color="green", ax=ax)
```

散布図の作成（３）



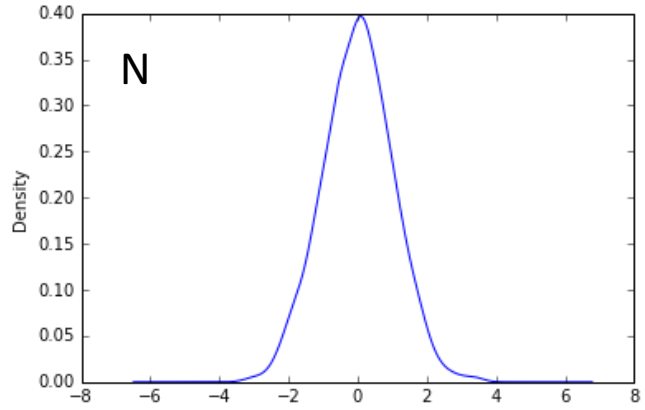
折れ線グラフの作成（１）

- 時間と数量の関係性を見たい場合は折れ線グラフを利用する。
- 基本的に時間をx 軸（横軸）に、数量を y 軸（縦軸）にとる。

- 今回は擬似的な株価データ（ランダムウォーク）を生成し、プロットしてみる。
  直前の値に正規分布に従う乱数が加算され続けるモデルを考えてみる。

$$S(t) = S(t-1) + aN$$

$S(t)$: **ある時刻 t における株価**
a: **定数値**
N: **正規分布に従う乱数**



（上の式の意味がわからなくてもグラフ描くのには問題ないです）

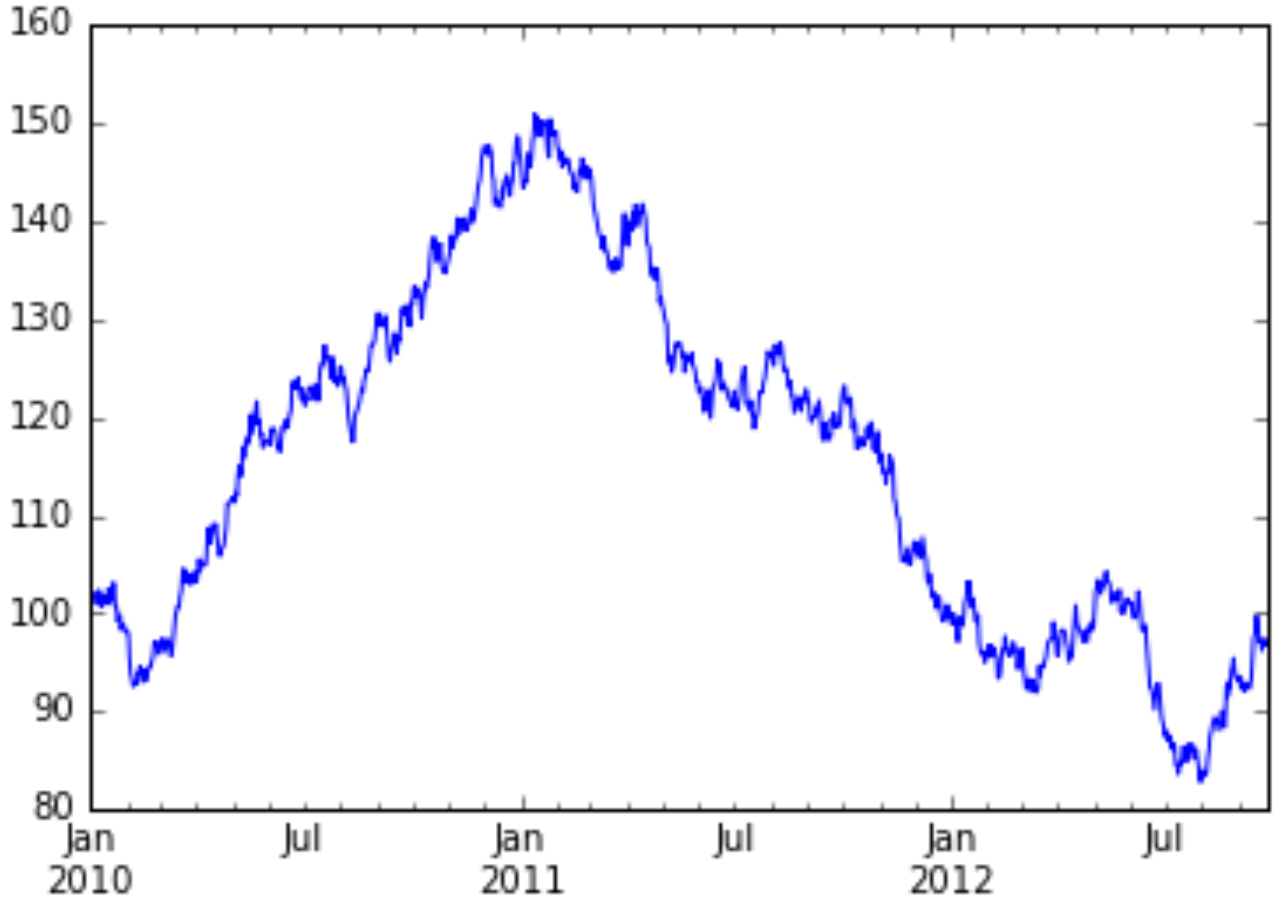
折れ線グラフの作成（２）

- サンプル株価データの作成

```
# 数値計算用ライブラリ numpy の読み込み
import numpy as np

# 擬似的な株価データ（＝ランダムウォーク）を生成してみる
# np.random.randn(1000) で平均 0, 標準偏差 1 の乱数1000個を生成
# pd.date_range('2010-01-01', periods=1000) で '2010-01-01' から 1000 日分のラベル生成
#   => 毎日の株価変動
ts_a = 1.0 * pd.Series(np.random.randn(1000), index=pd.date_range('2010-01-01', periods=1000))

# 初期値 を 100 として、ある日までの変化分を cumsum() メソッドで足し上げる（累積和）
stock_a_price = 100 + ts_a.cumsum()

# プロットしてみる
stock_a_price.plot(kind="line")
```

折れ線グラフの作成（３）

- 株価推移っぽいデータができた。



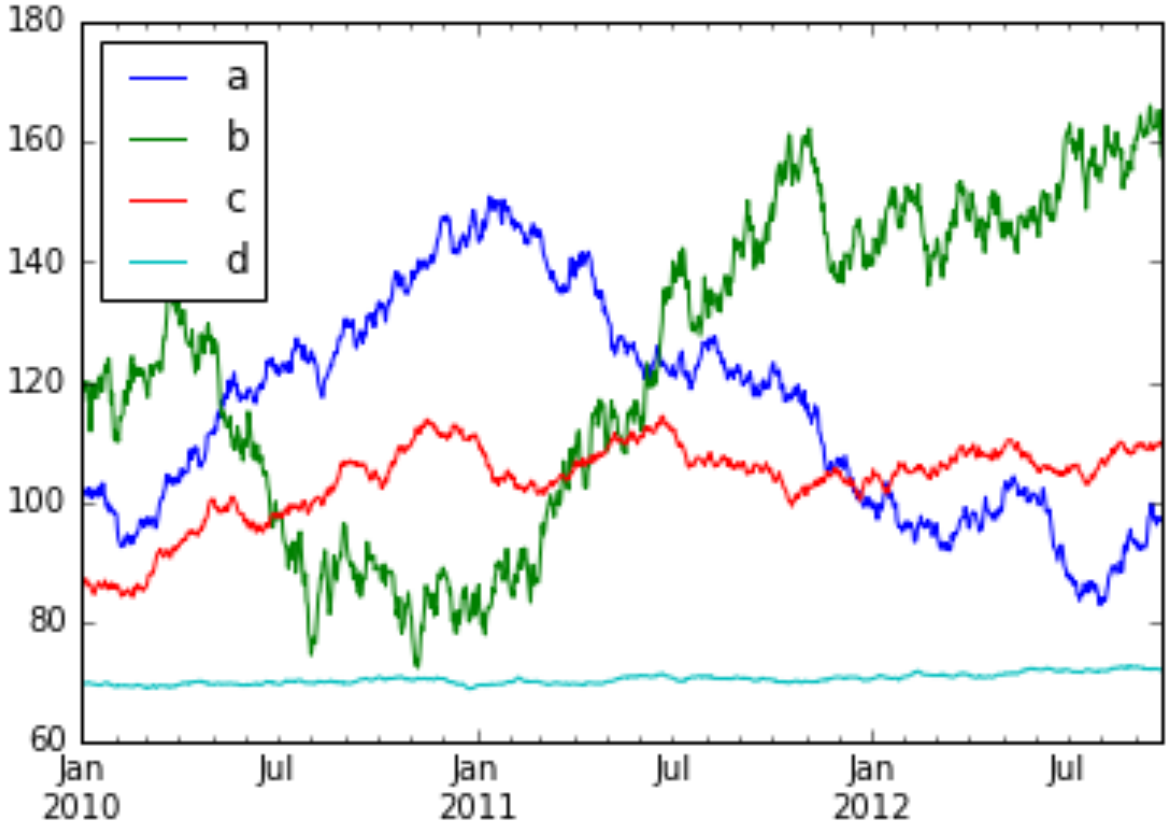
折れ線グラフの作成（４）

- 条件を変えて複数系列作ってみる。

```
# 変化量と初期値が違うものを幾つか生成してみる
ts_b = 2 * pd.Series(np.random.randn(1000), index=pd.date_range('2010-01-01', periods=1000))
ts_c = 0.5 * pd.Series(np.random.randn(1000), index=pd.date_range('2010-01-01', periods=1000))
ts_d = 0.1 * pd.Series(np.random.randn(1000), index=pd.date_range('2010-01-01', periods=1000))
stock_b_price = 120 + ts_b.cumsum()
stock_c_price = 85 + ts_c.cumsum()
stock_d_price = 70 + ts_d.cumsum()

# DataFrame にする
stock_df = pd.DataFrame(
    {"a": stock_a_price, "b": stock_b_price, "c": stock_c_price, "d": stock_d_price},
   columns=["a", "b", "c", "d"]
)

# プロット
stock_df.plot(kind="line")
```

折れ線グラフの作成（５）

- 複数系列でも同じようにプロットできる。



やってみよう（3）

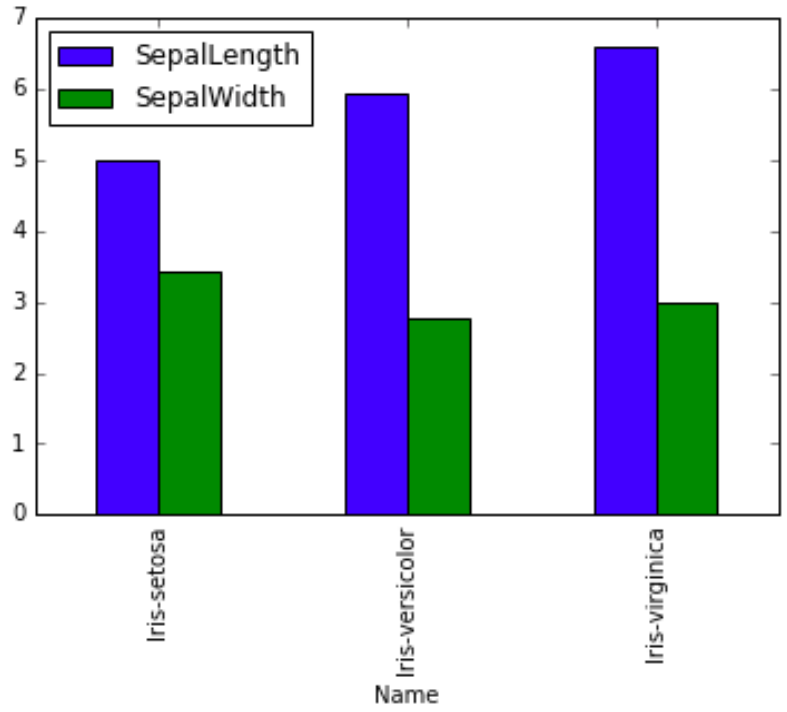
1. "03_iris.csv" を DataFrame 形式で読み込む。
2. "SepalLength", "SepalWidth", "Name" からなるデータフレーム sepal_df を作成する。
3. 品種別の "SepalLength", "SepalWidth" 中央値の棒グラフを作成する。
4. 品種別に色分けした "SepalLength", "SepalWidth" の散布図を作成する。

- 時間が余った人は他の組み合わせでもグラフを描いてみよう。

回答例（3）

In [108]:
```
# Iris データの読み込み
iris_df = pd.read_csv("03_iris.csv")

# "SepalLength", "SepalWidth", "Name" からなるデータフレーム sepal_df の作成
sepal_df = iris_df[["SepalLength", "SepalWidth", "Name"]]

# 品種別 "SepalLength", "SepalWidth" 中央値の棒グラフ作成
sepal_df.groupby("Name").mean().plot(kind="bar")
```

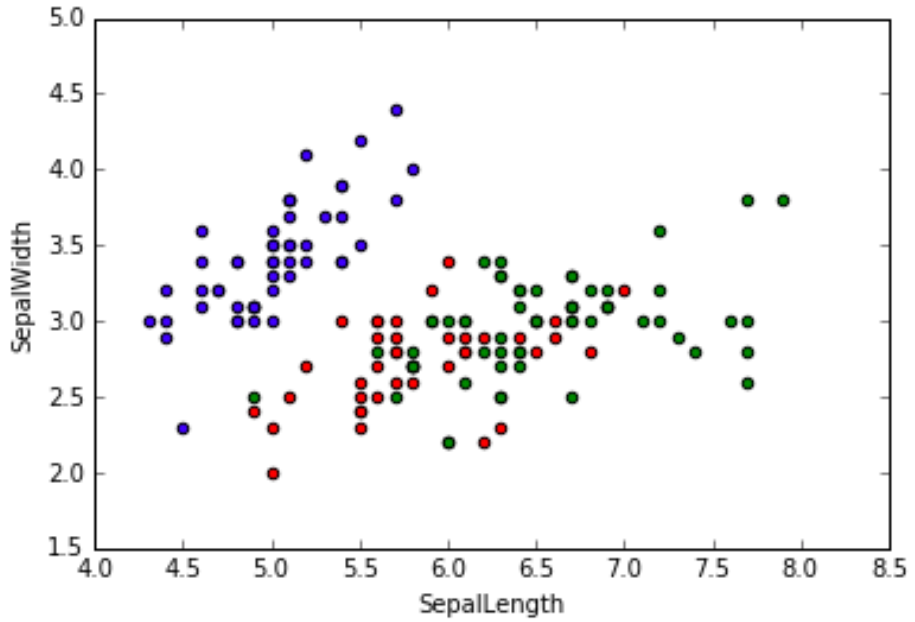
Out[108]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1267015c0>
```



解答例（３）

In [107]:
```
# 品種別に色分けした "SepalLength", "SepalWidth" の散布図作成
# 3品種別 DataFrame 作成
s_setosa_df = sepal_df[petal_df.Name == "Iris-setosa"]
s_versicolor_df = sepal_df[petal_df.Name == "Iris-versicolor"]
s_virginica_df = sepal_df[petal_df.Name == "Iris-virginica"]

# ax を次の plot 時に渡す
ax = s_setosa_df.plot(kind="scatter", x="SepalLength", y="SepalWidth", color="blue")
ax = s_versicolor_df.plot(kind="scatter", x="SepalLength", y="SepalWidth", color="red", ax=ax)
s_virginica_df.plot(kind="scatter", x="SepalLength", y="SepalWidth", color="green", ax=ax)
```

Out[107]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1150f79b0>
```



データの相関と回帰

データの相関と回帰

- 相関分析
  - 相関分析とは
  - 相関係数の算出
  - 自己相関係数の算出
  - 擬似相関に注意

- 回帰分析
  - 回帰分析とは
  - 線形回帰
  - 重回帰
  - ロジスティック回帰

相関分析とは

- 相関分析
  - ２つの変数の間の直線的な関係を数値化したもの。

- 以下の関係の強弱を相関係数で数値化できる。
  - 身長が高ければ、体重も重たい。
  - 来店者数が多ければ、売上も増える。
  - Iris の PetalLength が長ければ、PetalWidth も長い。

相関係数の算出（１）

- 定義
  - ２組の数値からなるデータ X, Y がある。

$$\boldsymbol{X} = (x_1, x_2, x_3, \cdots, x_i) \qquad \boldsymbol{Y} = (y_1, y_2, y_3, \cdots, y_i)$$

  - 相関係数 r は次のように計算できる。

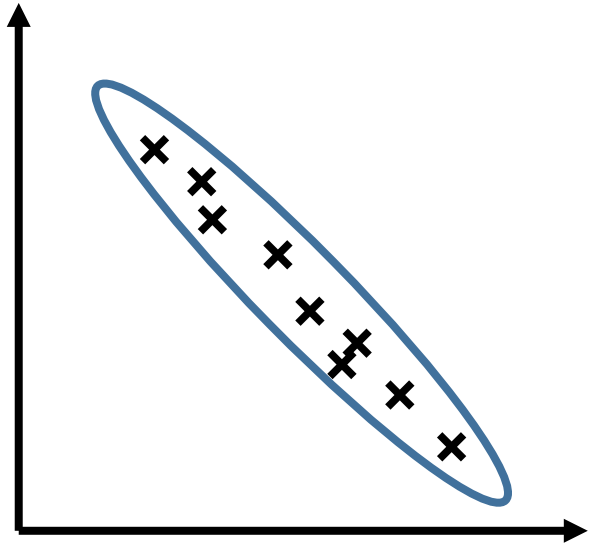
$$r = \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_X)(y_i - \mu_Y)}{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_X)^2}\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \mu_X)^2}} = \frac{(\text{共分散}: x \text{ の偏差と } y \text{ の偏差の平均値})}{(\boldsymbol{X} \text{ の標準偏差})(\boldsymbol{Y} \text{ の標準偏差})}$$

μ_X, μ_y: $\boldsymbol{X}$, $\boldsymbol{Y}$ の平均値

相関係数の算出（２）

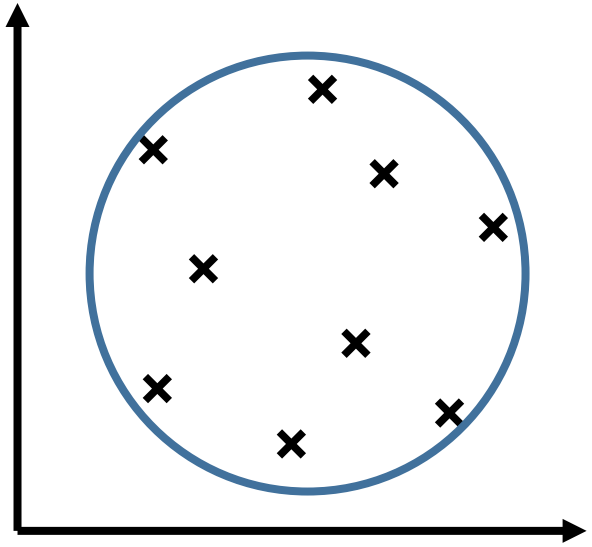
- 相関係数 r は -1 ～ 1 の間の値をとり、以下の様な傾向となる。

$r \fallingdotseq -1$



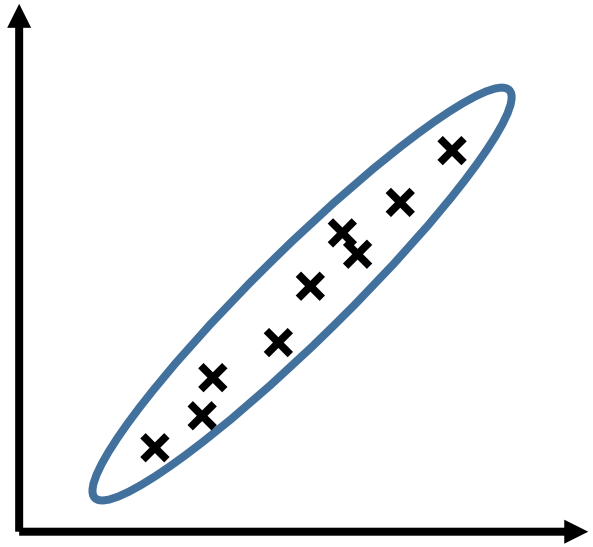
強い負の相関

$r \fallingdotseq 0$



無相関

$r \fallingdotseq 1$



強い正の相関

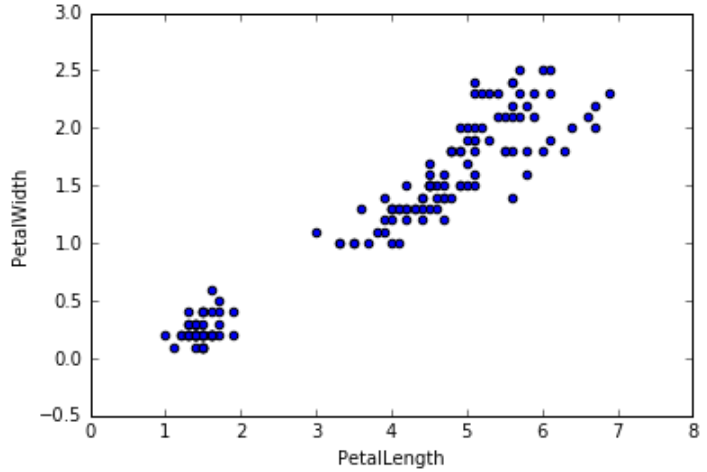
相関係数の算出（３）

- Iris の PetalLength と PetalWidth について計算してみる。

```
# PetalLength、PetalWidth と Name からなるデータフレーム作成
petal_df = iris_df[["PetalLength", "PetalWidth", "Name"]]

# 相関係数の算出
petal_df.corr()
```

Out[28]:

	PetalLength	PetalWidth
PetalLength	1.000000	0.962757
PetalWidth	0.962757	1.000000



（散布図だとこんな感じ）

自己相関係数の算出（１）

- 時系列データの場合
  - 元の時系列データと、 n 単位時間分シフトさせたデータ間で相関係数を算出。 => 自己相関係数
  - 周期成分がある場合、一定間隔で相関係数が高い部分が現れる。
    - 例えば、毎日1データずつ取得できる系列で、7点毎に自己相関係数が高い。 => 1週間単位の周期がある
    - 株価のランダムウォークの場合、直前の値との自己相関係数が高くなる。

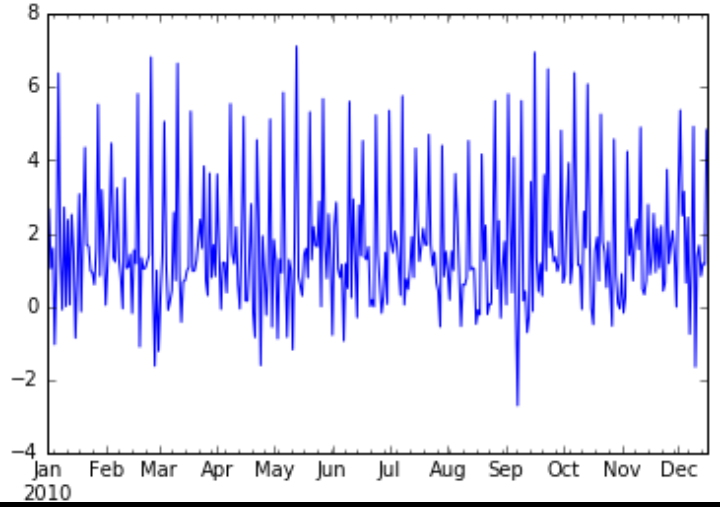
$S(t) = S(t-1) + aN$

$S(t)$: ある時刻 t における株価
a: 定数値
N: 正規分布に従う乱数

自己相関係数の算出（２）

- 7点周期のデータを作成してみる。

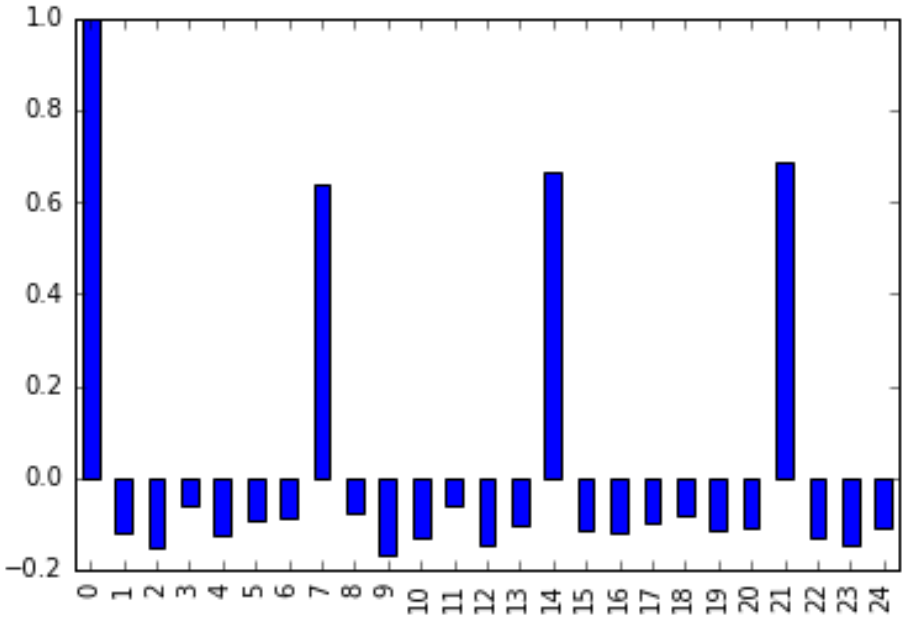
```
# [1, 1, 1, 1, 1, 1, 5] を 50 個分結合した Series (350 要素) 作成
s = pd.Series([1, 1, 1, 1, 1, 1, 5] * 50, index=pd.date_range('2010-01-01', periods=350))

# 作成した Series に乱数追加
s += pd.Series(np.random.randn(350), index=pd.date_range('2010-01-01', periods=350))

# プロット
s.plot(kind="line")
```



自己相関係数の算出（３）

- きれいに7点間隔の周期が現れる。

```
# 自己相関係数の算出
autocorr_series = pd.Series([s.autocorr(lag=i) for i in range(25)])

# プロット
autocorr_series.plot("bar")
```



擬似相関に注意

- 擬似相関
  - 2つの出来事に因果関係がないにもかかわらず、何らかの要因によって2つの集団に意味のある関係があるように見えてしまうこと。
  - A が B を発生させる => 相関関係成立
  - C が A, B を発生させる => 擬似相関
- 例：
  - 小学生100人に算数のテストを受けてもらったところ、身長が高いほど点数が良かった。
    => 身長が高いほど算数が得意？？？
  - 年齢という要素を考える。年齢が高いほうが当然難しい問題を解くことが出来る。年齢と身長、年齢とテストの点数、それぞれに相関関係はあるが、身長とテストの点数の間の関係は擬似相関。

回帰分析とは

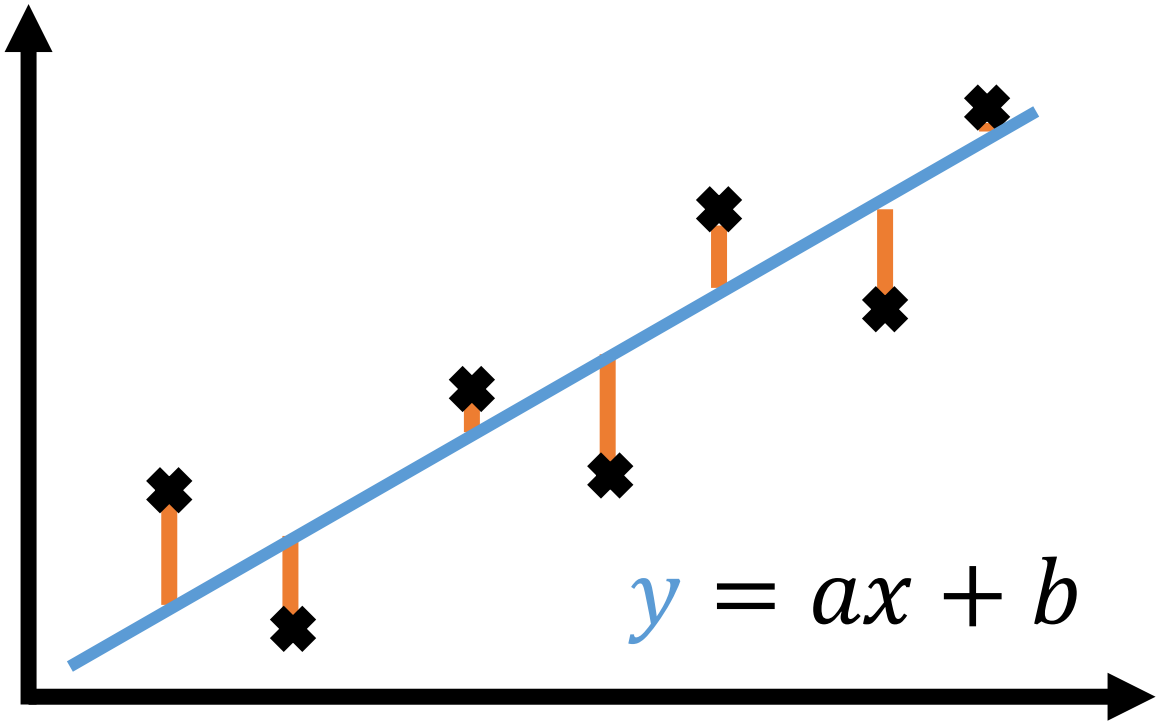
- 回帰分析
  - 因果関係がある変数の組み合わせを利用して、ある変数（1 ～ n 個）から特定の変数の将来的な値を予測する。
  - 予測に使用する変数（入力）　：説明変数 ***X***
    予測される変数（出力）　　：目的変数 ***Y***

$$\boldsymbol{Y} = f(\boldsymbol{X})$$

- 使用する手法
  - 線形回帰： 説明変数が 1 個。目的変数は数量（連続値）。
  - 重回帰： 説明変数が 2 個以上。目的変数は数量（連続値）。
  - ロジスティック回帰：説明変数は 1 個以上、目的変数は 0 or 1 の離散値。

線形回帰（１）

- 下の図の ― の誤差を 2 乗したものの合計が最小になるところを探すことで、適切な直線を探す。



=> 最小二乗法

y: 目的変数　a: 係数(傾き)
x: 説明変数　b: 切片

線形回帰（２）

- scikit-learn
  - Python のための機械学習ライブラリ。
  - 様々な機械学習や統計的手法に対応できる。
    - 回帰
    - 分類
    - クラスタリング
  - 手法毎にインターフェースが統一されていて、とても便利。

線形回帰（３）

- Iris の PetalLength と PetalWidth についてモデルを構築。
  - a: 直線の傾き
  - b: 切片
  - R^2: 決定係数（0 ~ 1 で、1に近づくほど当てはまりが良い）

```
# PetalLength、PetalWidth からなるデータフレーム作成
X = iris_df[["PetalLength"]] # DataFrame として取り出す
Y = iris_df["PetalWidth"]    # Series として取り出す

# 必要なライブラリ読み込み
from sklearn.linear_model import LinearRegression

# 線形回帰モデルを構築
model = LinearRegression()
model.fit(X, Y)
print("a:", model.coef_, "b:", model.intercept_)
print("R-squared:", model.score(X, Y))
```

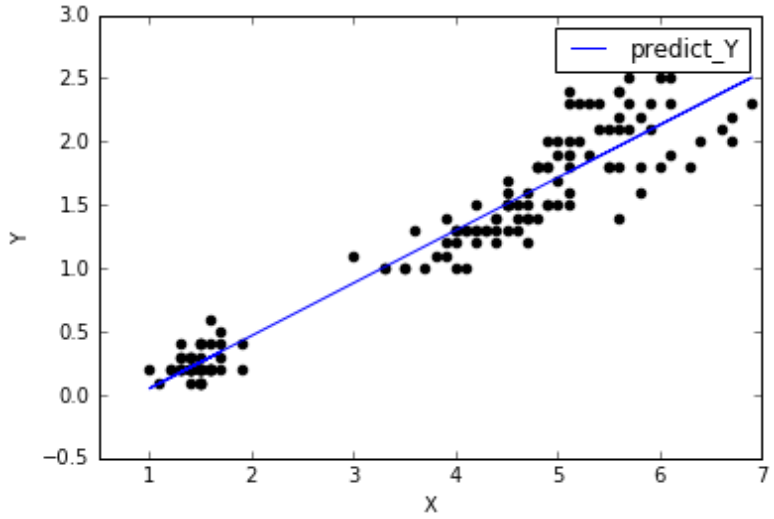
```
a: [ 0.41641913] b: -0.366514045217
R^2: 0.926901227922
```

決定係数が 0.9 を超えているので、
なかなか当てはまりが良い。

線形回帰（４）

- 結果をプロットしてみる
  - model.predict() の引数に説明変数となる X を渡せば、予測結果が帰ってくる。
  - 下記のように Series を渡せば、全ての値に対して、$y = ax + b$ の式にしたがって計算した結果が返って来る。

```
# 与えた X、本当の Y、予測した Y からなる DataFrame 作成
linear_df = pd.DataFrame(
    {"X": X["PetalLength"],"Y": Y, "predict_Y": model.predict(X)},
    columns=["X", "Y", "predict_Y"]
)

# ax を次の plot 時に渡せば同じグラフに重ねて描ける
ax = linear_df.plot(kind="scatter", x="X", y="Y", color="black")
linear_df.plot(kind="line", x="X", y="predict_Y", color="blue", xlim=(0.5, 7.0), ax=ax)
```



重回帰（１）

- 線形回帰の説明変数が2つ以上になったもの。

$$y = a_1x_1 + a_2x_2 + a_3x_3 + \cdots + a_ix_i + b$$

- 各変数の影響を比較したい場合は変数を正規化する。
  - 平均 = 0、標準偏差 = 1 になるように。
  - scikit-learn を利用する場合はオプションで指定できる。

- 説明変数同士は独立（無相関）を仮定している。
  - 相関が高い変数を同時に入れると解釈に困る場合が出てくる。
  - 事前にクラスタリングなどで、独立変数になるように変換しましょう。

重回帰（２）

- Iris の SepalLength を PetalLength と PetalWidth で予想してみる。

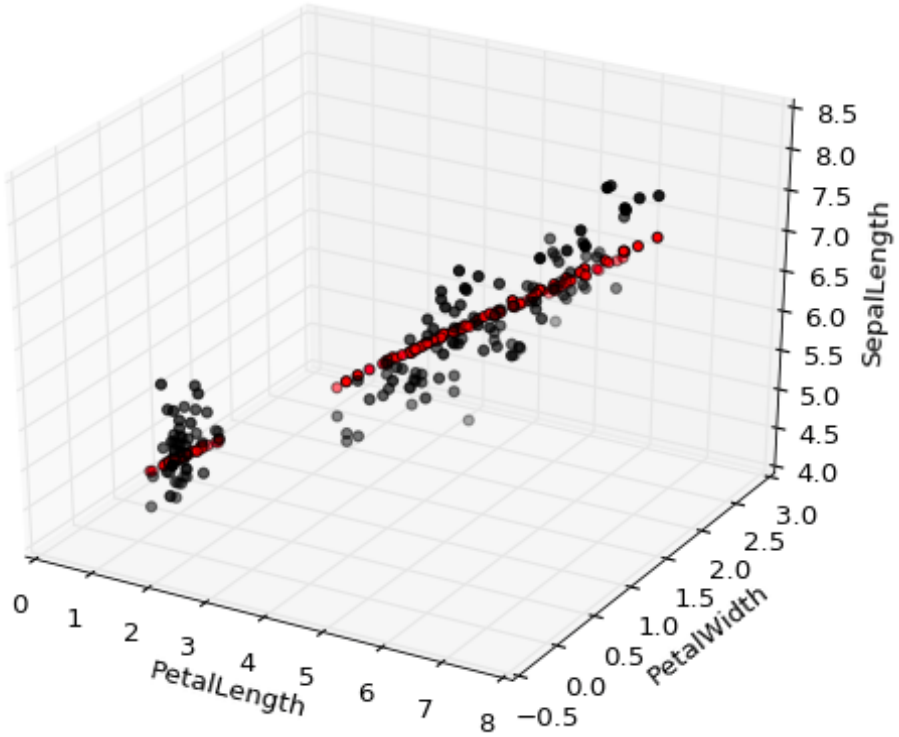
```
X = iris_df[["PetalLength", "PetalWidth"]]
Y = iris_df["SepalLength"]

# 重回帰でも LinearRegression でOK
model = LinearRegression()
model.fit(X, Y)
print("a:", model.coef_, "b:", model.intercept_)
print("R^2:", model.score(X, Y))
```

```
a: [ 0.54099383 -0.31667117] b: 4.18950102385
R^2: 0.766181597904
```

重回帰（3）

- 3次元座標に結果をプロットしてみる

```
# 3次元プロット用のためには mplot3d の Axes3D を利用する
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
plot_3d = plt.figure().gca(projection='3d')
plot_3d.scatter(X["PetalLength"], X["PetalWidth"], Y, c="black")
plot_3d.scatter(X["PetalLength"], X["PetalWidth"], model.predict(X), c="red")
plot_3d.set_xlabel("PetalLength")
plot_3d.set_ylabel("PetalWidth")
plot_3d.set_zlabel("SepalLength")
plt.show()
```

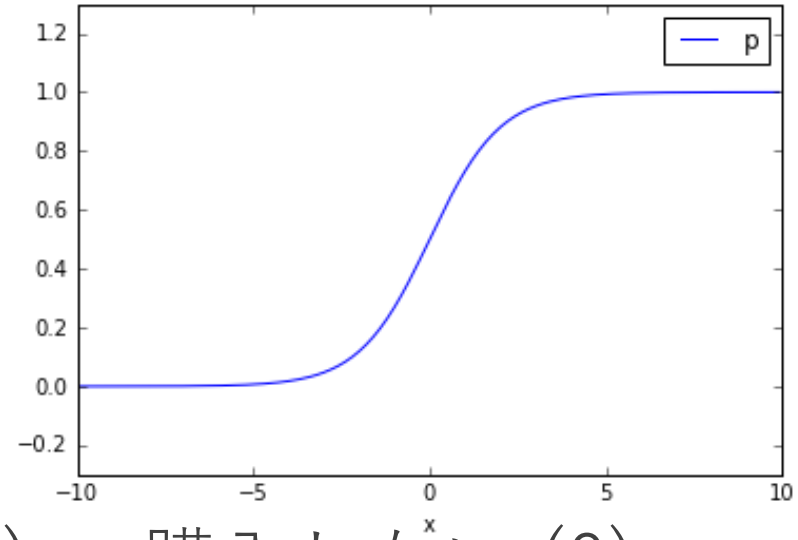


ロジスティック回帰（１）

- ロジスティック回帰は目的変数が 0 or 1 の離散値。
  - 予測結果は 0 ～ 1 の間の連続値として得られる。
  - 回帰だが、後に述べるデータの分類手法として使う。

$$\log\left(\frac{p}{1-p}\right) = a_1x_1 + a_2x_2 + a_3x_3 + \cdots + a_ix_i + b$$

p: 出力（0～1）



- 以下の様な場合に用いられる。
  - 顧客の商品購入確率の算出： 購入する (1) or 購入しない (0)。
  - スパムメールの分類：スパム(1) or スパムではない (0)。

ロジスティック回帰（２）

- Iris データセットを用いて、 PetalLength, PetalWidth がわかっている時、その Iris の品種が setosa である確率を求めてみる。

```
# Iris-setosa なら 1、そうでなければ 0 を返す関数
def is_setosa(string):
    if string == "Iris-setosa":
        return 1
    else:
        return 0

X = iris_df[["PetalLength", "PetalWidth"]]
Y = iris_df["Name"].map(is_setosa)

# ロジスティック回帰モデルを読み込みモデリング
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
model.fit(X, Y)
print("a:", model.coef_, "b:", model.intercept_)
print("R^2:", model.score(X, Y))
```

```
a: [[-1.14714223 -1.7130534 ]] b: [ 3.97259233]
R^2: 1.0
```

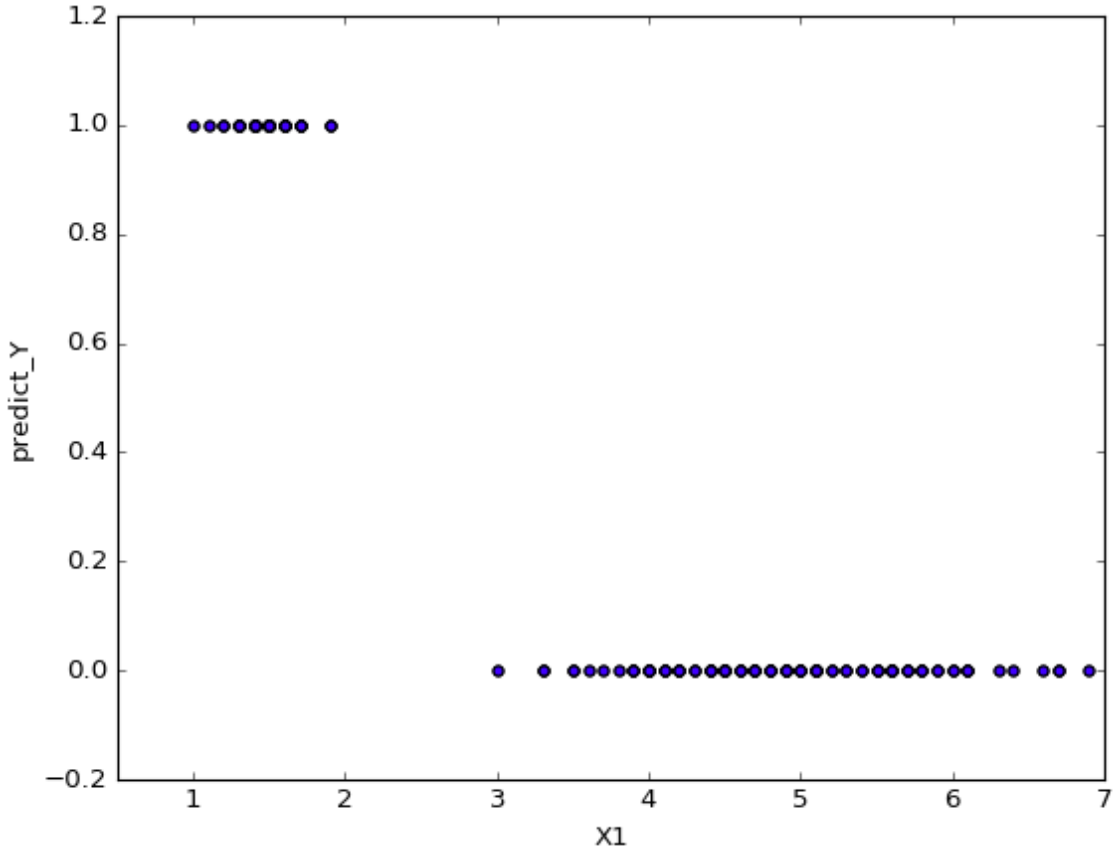
ロジスティック回帰（３）

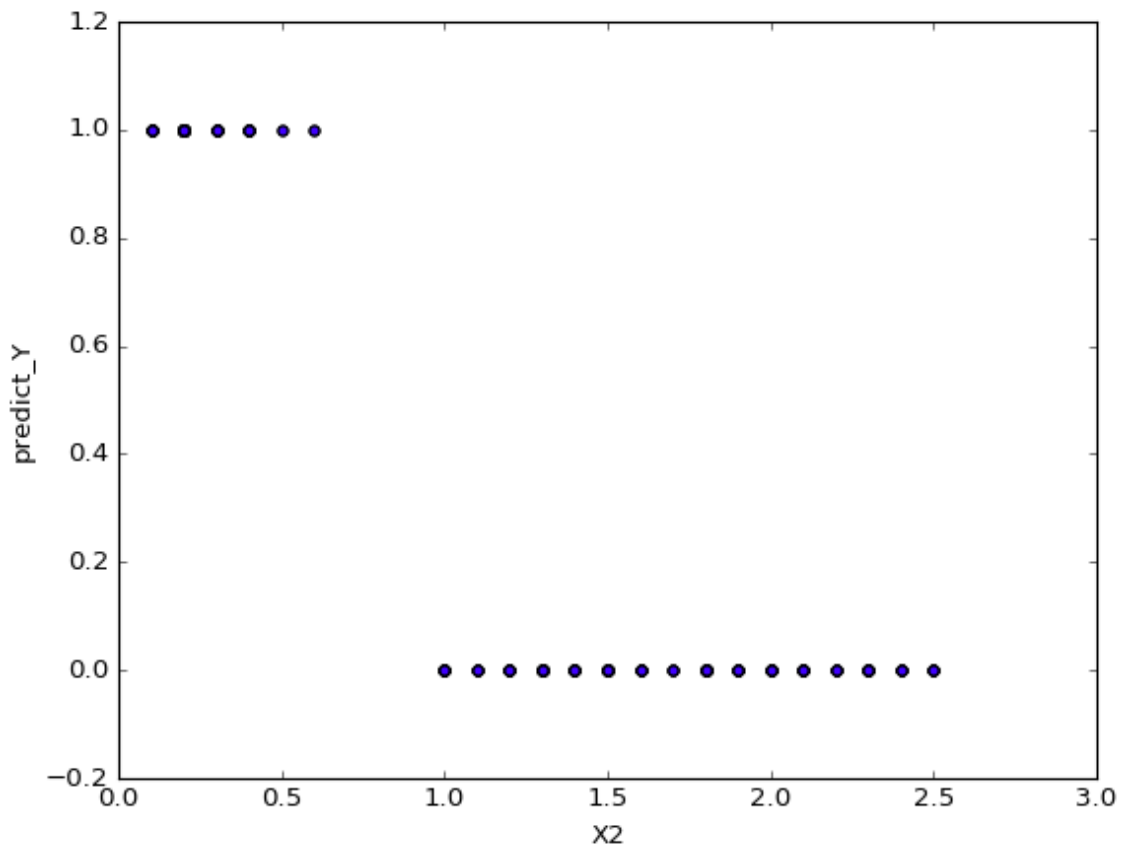
- 分類結果を確認してみる

```
# 与えた X、本当の Y、予測した Y からなる DataFrame 作成
logistic_df = pd.DataFrame(
    {"X1": X["PetalLength"],
     "X2": X["PetalWidth"],
     "Y": Y, "predict_Y": model.predict(X)},
    columns=["X1", "X2", "Y", "predict_Y"]
)

# プロット
ax = logistic_df.plot(kind="scatter", x="X1", y="Y", color="black")
logistic_df.plot(kind="scatter", x="X1", y="predict_Y", color="blue", xlim=(0.5, 7.0), ax=ax)
ax = logistic_df.plot(kind="scatter", x="X2", y="Y", color="black")
logistic_df.plot(kind="scatter", x="X2", y="predict_Y", color="blue", xlim=(0.0, 3.0), ax=ax)
```

ロジスティック回帰（４）

- 1: setosa、0: setosa でない ときっちり別れる。





やってみよう（4 − 1）

1. "03_iris.csv" を読み込み、PetalLength, PetalWidth, Name からなるデータフレームを作成する。
2. 品種別に PetalLength、PetalWidth の相関係数を算出する。

やってみよう（4 − 2）

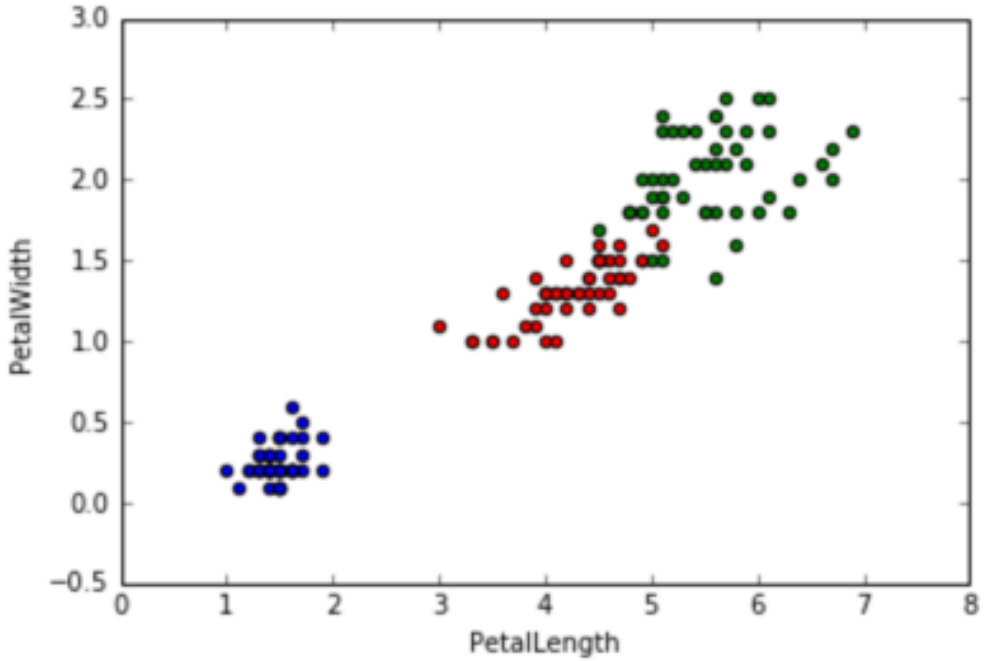
1. "03_iris.csv" を DataFrame として読み込み、品種名が "Iris-setosa" のもののみ取り出す。
2. 品種名が "Iris-setosa" の SepalLength からなる DataFrame 、 またSepalWidth からなる Series をそれぞれ作成する。
3. "Iris-setosa" の SepalLength から SepalWidth を予測するモデルを構築する。
4. 元データを点で、予測結果を直線でプロットする。

解答例（４－１）

- 品種別で見ると versicolor 以外は関係性が薄い
  => 全体と個の傾向が異なる

```
# PetalLength、 PetalWidth と Name からなるデータフレーム作成
petal_df = iris_df[["PetalLength", "PetalWidth", "Name"]]

# 品種別の相関係数の算出
petal_df.groupby("Name").corr()
```

		PetalLength	PetalWidth
Name			
Iris-setosa	**PetalLength**	1.000000	0.306308
	PetalWidth	0.306308	1.000000
Iris-versicolor	**PetalLength**	1.000000	0.786668
	PetalWidth	0.786668	1.000000
Iris-virginica	**PetalLength**	1.000000	0.322108
	PetalWidth	0.322108	1.000000

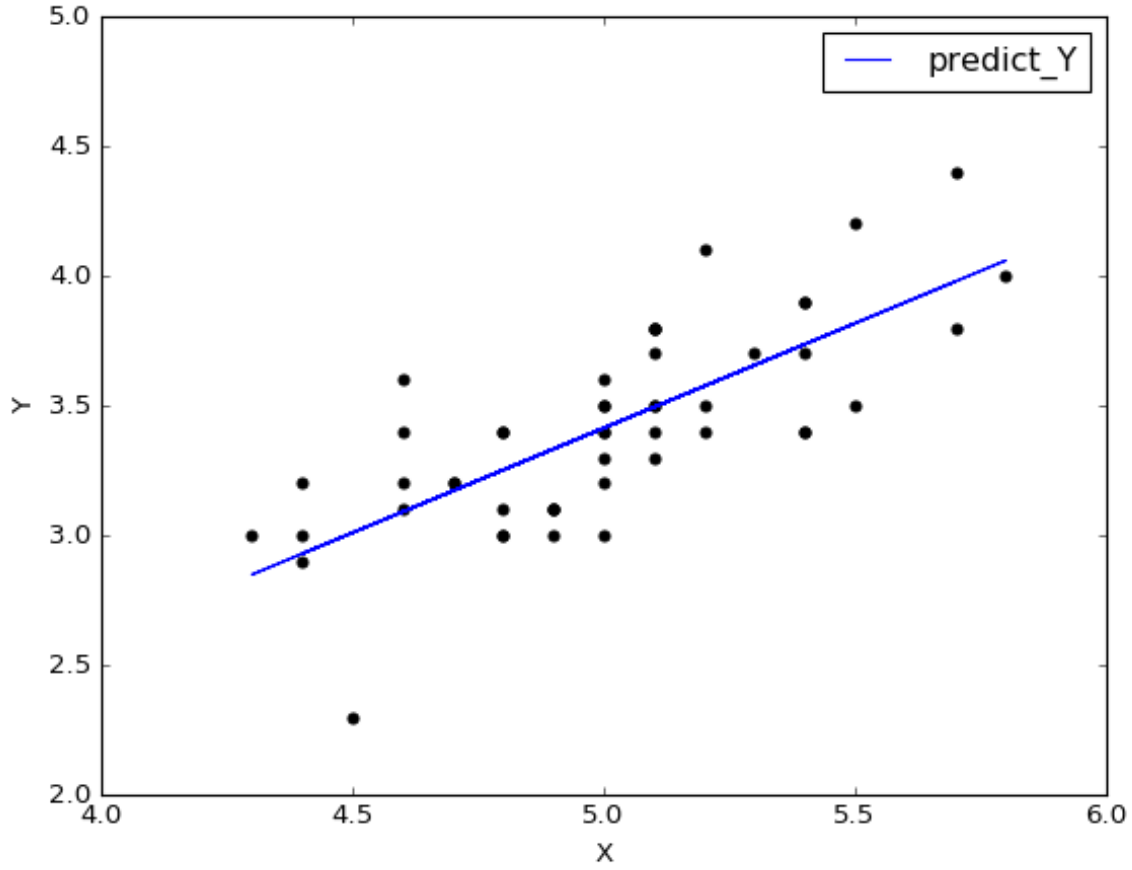


解答例（４－２）

```
# 元データ作成
setosa_df = iris_df[iris_df["Name"] == "Iris-setosa"]
X = setosa_df[["SepalLength"]]
Y = setosa_df["SepalWidth"]

# 線形回帰モデルを構築
model = LinearRegression()
model.fit(X, Y)
print("a:", model.coef_, "b:", model.intercept_, "R^2:", model.score(X, Y))

# プロット
linear_df = pd.DataFrame(
    {"X": X["SepalLength"], "Y": Y, "predict_Y": model.predict(X)},
    columns=["X", "Y", "predict_Y"]
)
ax = linear_df.plot(kind="scatter", x="X", y="Y", color="black")
linear_df.plot(kind="line", x="X", y="predict_Y", color="blue", xlim=(4, 6), ax=ax)
```

解答例（4 － 2）

```
a: [ 0.80723367] b: -0.623011727604 R^2: 0.557680925892
```



データの分類

データの分類

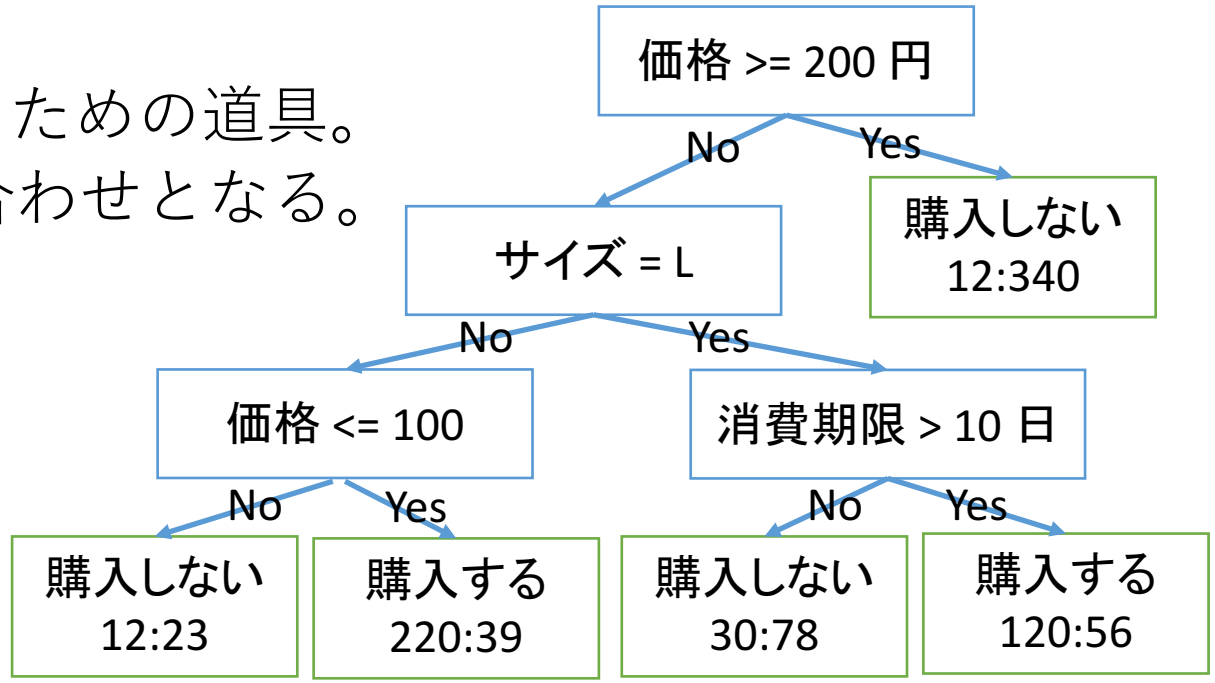
- 分類
  - 分類とは？
  - 決定木分析
- クラスタリング
  - クラスタリングとは？
  - k-means 法
- モデル構築用データとテスト検証用のデータ

分類とは？

- 因果関係がある変数の組み合わせを利用して、ある変数（1 ～ n 個）からその要素が属するラベルを予測する。
  - 正解となるデータが必須。
  - 例：
    - メールのスパム判定（スパムである or スパムでない）
    - 猫画像判定（ある画像に猫が含まれれている or 含まれていない）
    - ユーザーのカテゴリ判定（学生、専業主婦、会社員、etc. を出力）
  - 回帰では出力が連続値だったが、分類は離散値。
    - 前述のロジスティック回帰は分類のためによく使われる。

決定木分析（１）

- 決定木（Decision Tree）
  - 樹木状の構造で分類ルールを抽出するための道具。
  - 大量の分岐条件（IF-ELSE） の組み合わせとなる。



例：たまご（10個入り）の購買モデル

- ジニ係数
  - 不平等さを表す指標。
  - 0 ～ 1 の値をとり、 0 で最も平等。
    => 0 なら同じ種類のデータのみ。
  - ジニ係数が小さくなるようなルールを策定し、次々とデータを分割していく。

決定木分析（２）

- 「タイタニック号乗客の生存要因」について、決定木分析を行ってみる。

- 使用するデータ (04_titanic.tsv)
  - 以下のカラムからなる集計値
    - インデックス (idx)
    - 船室等級 (Class)
    - 性別 (Sex)
    - 年齢 (Age)
    - 生還の有無 (Survived)
    - 人数 (Freq)

	Class	Sex	Age	Survived	Freq
1	1st	Male	Child	No	0
2	2nd	Male	Child	No	0
3	3rd	Male	Child	No	35
4	Crew	Male	Child	No	0
5	1st	Female	Child	No	0

決定木分析（３）

- 量的変数と質的変数
  - 量的変数： 大小関係が存在する。数値。
    - 例：
      - 1、9.0、-134 など
  - 質的変数： 大小関係が存在せず比較できないもの。文字列など。
    - 例：
      - "男性" と "女性"、アヤメの品種名 など
    - scikit-learn では質的変数をそのまま説明変数にできないので、量的変数に変換してやる必要がある。

	Class	Sex	Age
1	1st	Male	Child
2	2nd	Male	Child

	1st	2nd	3rd	Crew	Male	Female	Child	Adult
0	1	0	0	0	1	0	1	0
1	0	1	0	0	1	0	1	0

決定木分析（３）

- モデルを構築する

```
# 必要なライブラリ読み込み
from sklearn import tree
from sklearn.feature_extraction
import DictVectorizer

# データ読み込み (TSV: タブ区切り形式なので sep 引数で指定、index カラムは除外して読み込む)
titanic_df = pd.read_csv("04_titanic.tsv", sep="¥t", index_col=0)

# 説明変数の DataFrame (DictVectorizerで質的変数を変換する)、目的変数の Series 作成
dict_vec = DictVectorizer()
X = dict_vec.fit_transform(
    titanic_df[["Class", "Sex", "Age"]].to_dict('records')
    ).toarray()
Y = titanic_df["Survived"]  # Series として取り出す
W = titanic_df["Freq"]  # モデルを構築
model = tree.DecisionTreeClassifier(max_depth=3)  # 最大の深さを3に
model.fit(X, Y, sample_weight=W)  # 集計済みデータの場合 Weight に頻度を指定
```

決定木分析（４）

- ツリーグラフを描くのに、Graphviz というツールのインストールが必要になります。
- 今回は各環境に対応する時間的余裕がなかったため、実行結果の紹介のみとし、インストール方法は割愛します。
- 各自調べて、後日チャレンジしてみてください。

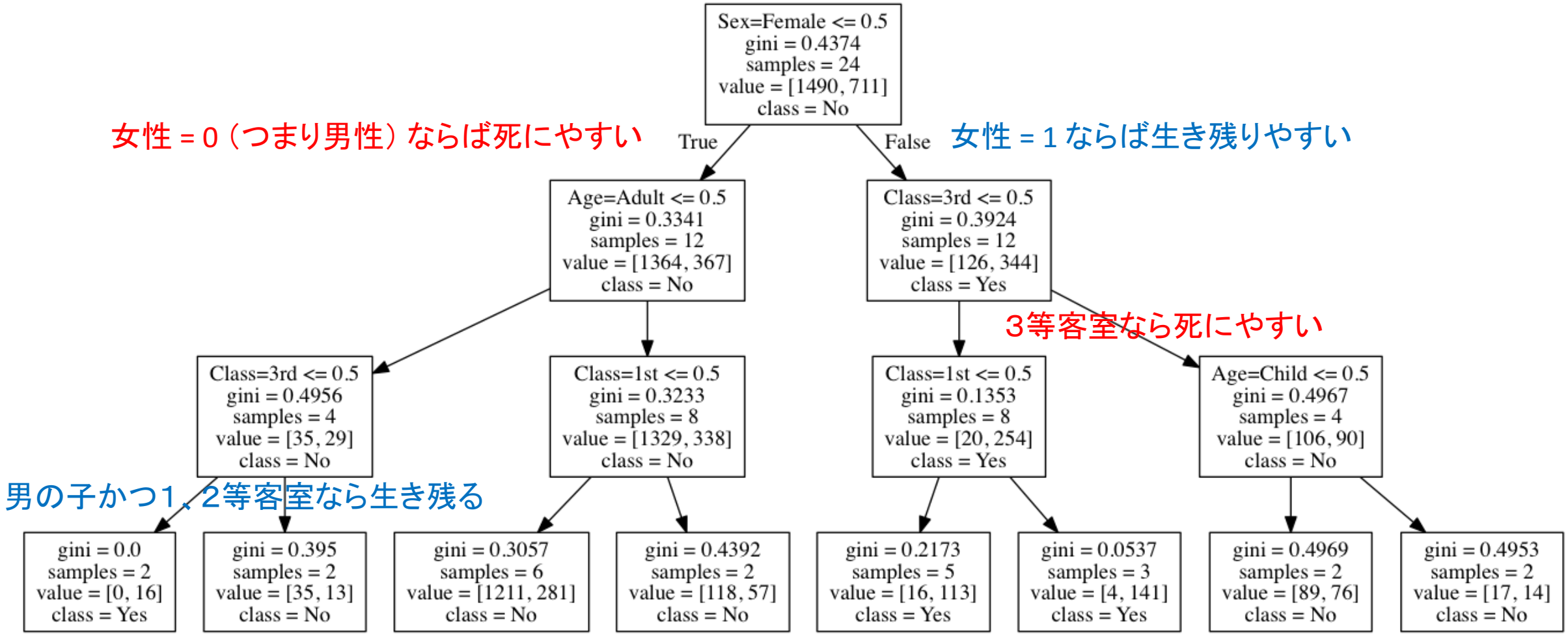
決定木分析（５）

- 図の作成

```
# 図の作成# ライブラリ読み込み
from sklearn.externals.six import StringIO
from IPython.display import Image
!pip install pydotplus  # pydotplus のインストール
import pydotplus
# 画像データ作成
dot_data = StringIO()
tree.export_graphviz(
    model,
    out_file=dot_data,
    feature_names=dict_vec.get_feature_names(),
    class_names=["No", "Yes"]
)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
# 画像として出力
Image(graph.create_png())
```

決定木分析（6）

- このような図が出力される。



クラスタリングとは？

- 因果関係がある変数の組み合わせを利用して、ある変数（1 ～ n 個）から「似ている」要素同士をまとめて、ラベル付けを行う。
  - 一度ラベル付けを行うモデルを作成すれば、未知のものに対してもラベル付けを行える。
  - 正解となるデータはなくてよい（分類と違う点）。
  - つまり「似ている」の定義は誰かの主観に基づいて決定される。

    => 客観的な証拠として使うには不適切。

k-means 法（１）

- k-means 法（k平均法）
  - クラスタの平均を用い、事前に与えられたクラスタ数に分割する。
  - 「似ている」＝「距離が近い」。
    - 使用する変数の最大、最小値が大きく異なる場合、
      ０～１となるように割合に直す（正規化する）必要がある。
    - 距離計算のために質的変数は量的変数に直す必要がある。
  - 振られるラベル（クラスタ番号）はランダムなので注意。
    - 「小さい数値に小さい番号が割り当たる」という規則はない。

k-means 法（２）

- ある EC サイトを利用しているユーザーの1ヶ月間支払金額を用いて、ユーザーをヘビー、ミドル、ライトユーザーに自動分類してみる。

```
# データの読み込み
#    "02_10000values.csv" を利用して仮想的なデータを作ってみる
s = pd.read_csv("02_10000values.csv", header=None, squeeze=True).map(int)
payment_df = pd.DataFrame(s, columns=["payment"])
payment_df.head()
```

	payment
0	2929
1	1000
2	2279
3	869
4	148

k-means 法（３）

- ヘビー、ミドル、ライトユーザー => クラスタ数： 3

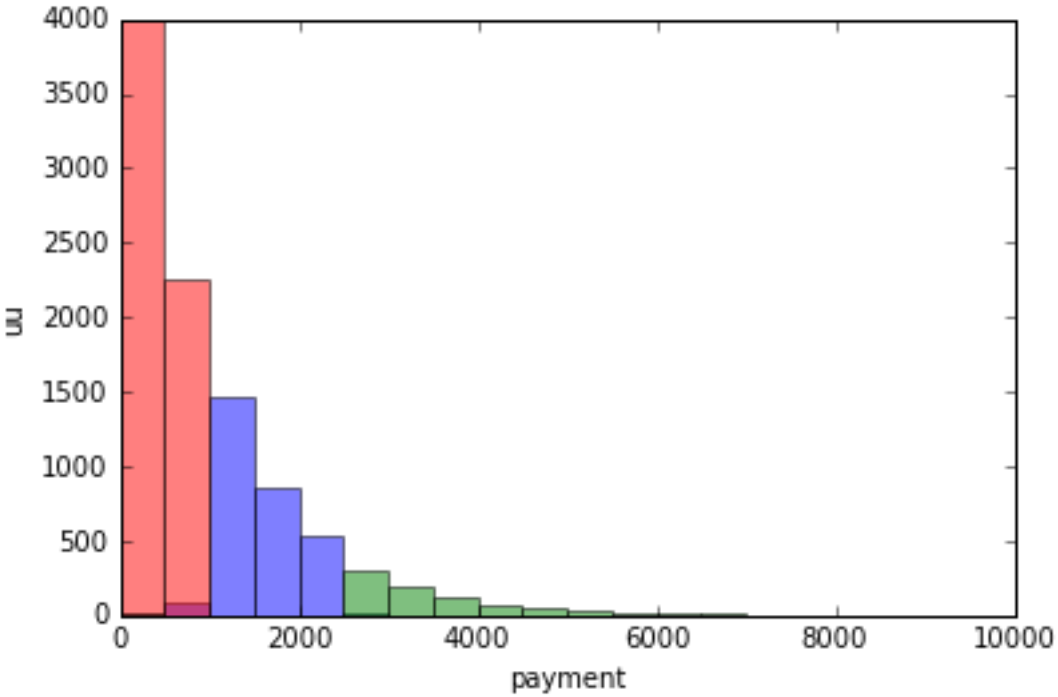
```
# ライブラリ読み込み
from sklearn.cluster import KMeans

# モデル作成 (クラスタ数 = 3)
model = KMeans(n_clusters=3)
model.fit(payment_df)

# ラベルの取得
payment_df["label"] = model.labels_
payment_df.head()
```

	payment	label
0	2929	1
1	1000	2
2	2279	2
3	869	0
4	148	0

k-means 法（４）

- ラベル別にプロットしてみる

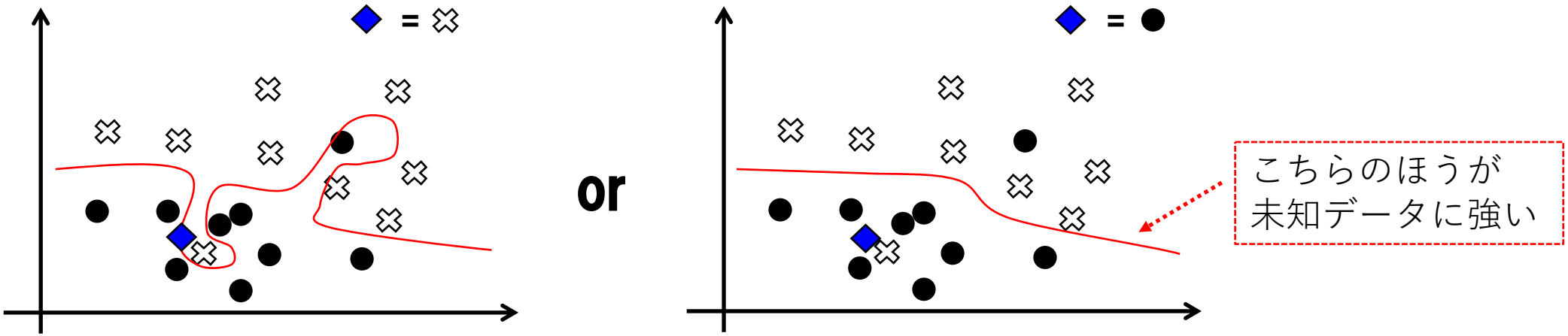
```
# ライブラリ読み込み
import matplotlib.pyplot as plt
%matplotlib inline

# データ分割
label_0 = payment_df[payment_df.label == 0]["payment"]
label_1 = payment_df[payment_df.label == 1]["payment"]
label_2 = payment_df[payment_df.label == 2]["payment"]

# matplotlib を直接利用してグラフ描画
plt.hist(label_0, bins=20, range=(0, 10000), alpha=0.5)
plt.hist(label_1, bins=20, range=(0, 10000), alpha=0.5)
plt.hist(label_2, bins=20, range=(0, 10000), alpha=0.5)
plt.xlabel("payment")
plt.ylabel("uu")
plt.show()
```



無事ヘビー、ミドル、
ライトユーザーに別れた

モデル構築用データとテスト検証用のデータ（１）

- 過剰適合（Overfitting）
  - 訓練データは高精度だが、未知データに対しては精度が出ない現象。
    - そもそもデータ数が少ない
    - 説明変数が訓練データ量に対して多すぎる
    - モデルの自由度が高すぎる

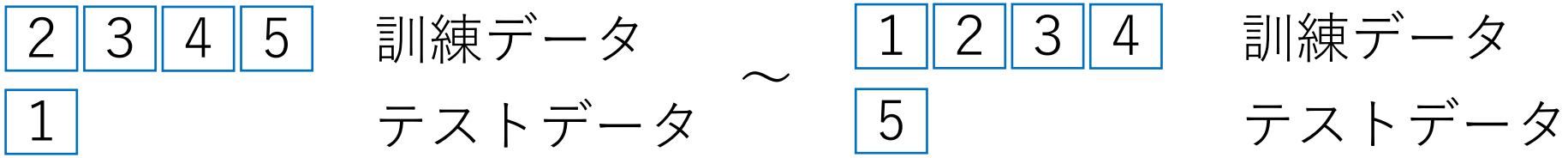


  - 「適度ないい加減さ」も回帰・分類で安定した精度を追求する上では大切。

モデル構築用データとテスト検証用のデータ（２）

- クロスバリデーション（交差検定）
  - 訓練データとテストデータの分割方法

全データを k 個に分割



k 回試行してその平均を利用

テストデータは常に未知のデータ
⇒ 過剰適合になりづらい

やってみよう：宿題

- Iris の品種を識別するための特徴（PetalWidth、PetalLength、SepalWidth、SepalLength）を調べる。

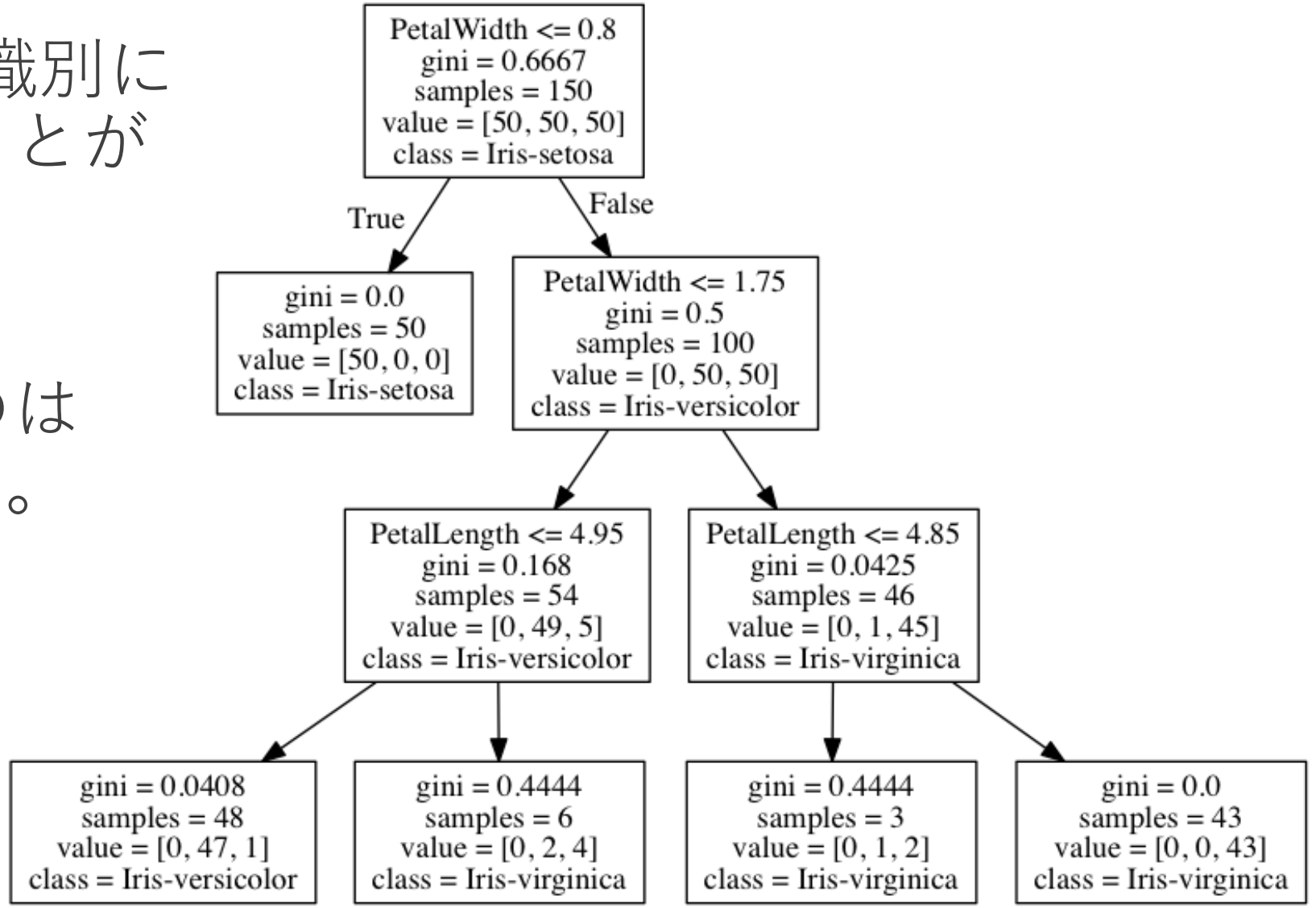
1. Iris データセットを読み込む。
2. PetalWidth、PetalLength、SepalWidth、SepalLength からなる DataFrame を作成する。
3. Name からなる Series を作成する。
4. 決定木によるモデルを構築する。
5. 図にする。

解答例（宿題）

```
iris_df = pd.read_csv("03_iris.csv")

features= ["PetalWidth", "PetalLength", "SepalWidth", "SepalLength"]
X = iris_df[features]
Y = iris_df["Name"]

model = tree.DecisionTreeClassifier(max_depth=3)
model.fit(X, Y)

dot_data = StringIO()
tree.export_graphviz(
    model,
    out_file=dot_data,
    feature_names=features,
    class_names=["Iris-setosa", "Iris-versicolor", "Iris-virginica"]
)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

解答例（宿題）

- PetalWidth が最も識別に有効な変数であることが確認できた。
- そして次に有効なのは PetalLength である。



参考資料

参考資料一覧

- 今回著者の方の許可をいただき多くの内容を下記から引用しています。
  - データサイエンティスト養成読本 R活用編,
    酒巻 隆治 , 里 洋平, 市川 太祐, 福島 真太朗, 安部 晃生, 和田 計也, 久本 空海,
    西薗 良太, 技術評論社 (2014/12/12)
  - ビジネス活用事例で学ぶ データサイエンス入門,
    酒巻 隆治, 里 洋平, SBクリエイティブ (2014/6/25)
- Iris
  - https://raw.githubusercontent.com/pydata/pandas/master/pandas/tests/data/iris.csv
- タイタニック号乗客の生存
  - http://www.is.titech.ac.jp/~mase/mase/html.jp/temp/Titanic.jp.html

付録

Kaggle

- 時間に余裕をもって終わってしまった方はチャレンジしてみましょう。スキルの向上のためには実戦あるのみ。

- データサイエンティストのためのコンペ
  - https://www.kaggle.com

- 挑戦者の例（この人は R 言語でチャレンジ）
  - http://shinya131-note.hatenablog.jp/entry/2015/07/12/153538