はじめに

本講座の目標

本講座は、Python初学者が次の3つをマスターできることを目標とした講座です。

1. データの入力：　ファイルの読み取り方
2. データの構造化：データ成形・加工の方法
3. データの出力：　可視化の方法や保存

これら3つのステップは、装置や計測方法が異なっていても、共通して行うデータ処理の流れですよね。Excelによるデータ処理でも、普段から行っている作業です。

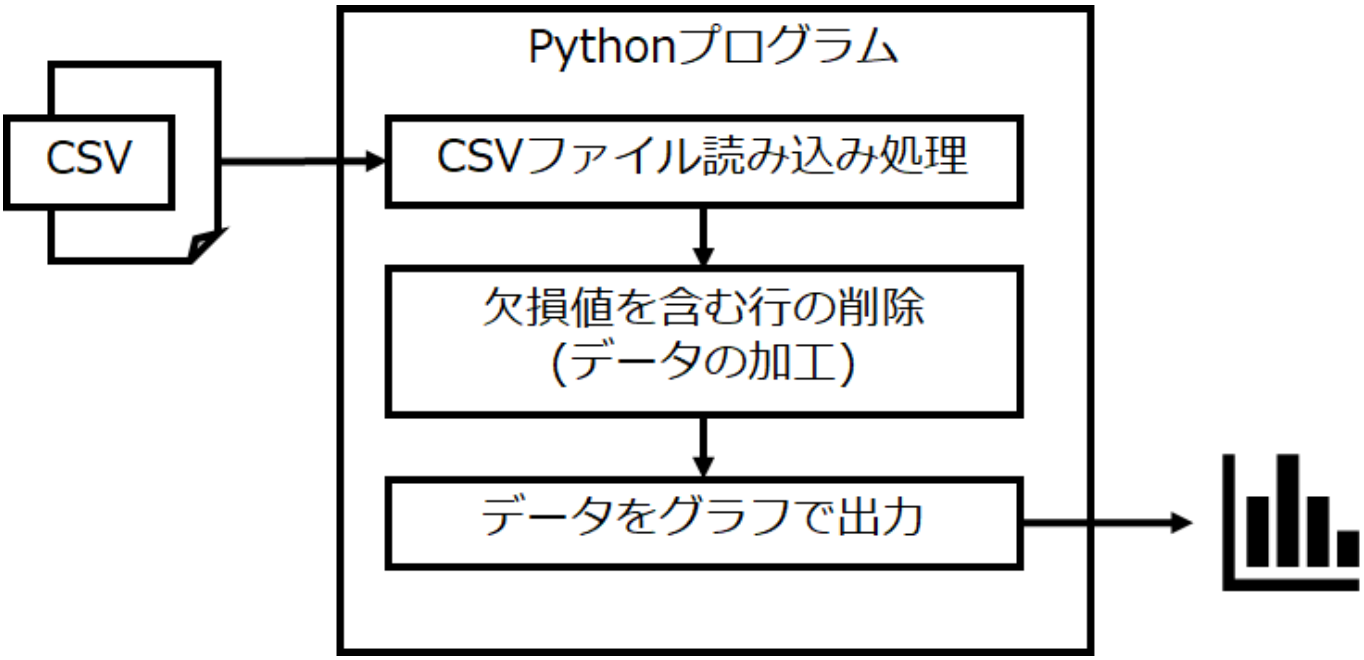
では、Pythonではどのように進めるのでしょうか？本講座は、この作業の流れを習得することを目的したカリキュラムです。

またARIM事業では、各機器ごとにデータ構造化コードを作成していますが、実は、この3つのステップを組み入れたコードを各機器ごとに、その機器での利用形態や今後のデータ利活用にあわせて作っています。

こんなことができます

本講座の内容を習得すると、例として次のようなプログラムを作成・実行することができるようになります。

- CSVファイルを読み込みます。
- CSVファイルから読み込んだデータのうち、欠損値を含む行を取り除きます。
- データをグラフで出力します。



まずは、実際にプログラムを動かしてみましょう。

まだプログラムの具体的な内容を理解する必要はありません。このような短いプログラムを記述するだけで、データの加工やグラフ出力ができる、というイメージをつかんでいただければ十分です。

以下に、プログラムのソースコードが記述されています。ソースコードとは、プログラムで行いたい処理をプログラミング言語(今回はPython)で記述したものです。

プログラムを実行するには、ソースコードの任意の場所をクリックした後、次のどちらかの操作を行います。

- ソースコード左上の実行ボタンを押す
- Ctrl+Enterを押す

In [ ]:

```
# ソースコード
# スペクトルデータのグラフを作成
#-------------------------------------
import pandas as pd
import matplotlib.pyplot as plt

# CSVファイルからデータを読み込み
# Google Colaboratoryではオンラインのファイルを読み込むが
# 通常はローカルドライブのファイルを読み込む
df = pd.read_csv("https://raw.githubusercontent.com/tendo-sms/python_seminar_2022/main/lecture1/samp

# 欠損値を含む行を削除
df = df.dropna()

# 折れ線グラフをプロット
x = df['Wavelength']
y = df['Intensity']

plt.plot(x, y)

# グラフタイトル・軸ラベルの設定
plt.title("Spectrum")
plt.xlabel("Wavelength")
plt.ylabel("Intensity")

# 完成したグラフを画面に描画
plt.show()
```

最初にプログラムを実行するときは、少し時間がかかることがあります。しばらく待つと、プログラムが実行された結果、グラフが表示されます。

Pythonの実行環境・エディタの紹介

Pythonのプログラムは、次の流れで実行します。

1. エディタでソースコードを作成する。(Windowsのメモ帳や、LinuxのVim/Emacs等をイメージしてください)
2. Python実行環境がソースコードに従ってプログラムを実行する。

ここでは、代表的なPython実行環境とエディタをご紹介します。

Python実行環境

代表的なPythonの実行環境として、公式版PythonとAnacondaがあります。

公式版Python

- 公式に配布されているPython実行環境です。

- Pythonは、非常に多くのさまざまな機能が「**パッケージ**」という形で公開されています。公式版Pythonは必要最低限の構成で配布されているので、自分で必要なパッケージだけをインストールして使いたい場合は、公式版Pythonを利用します。

Anaconda

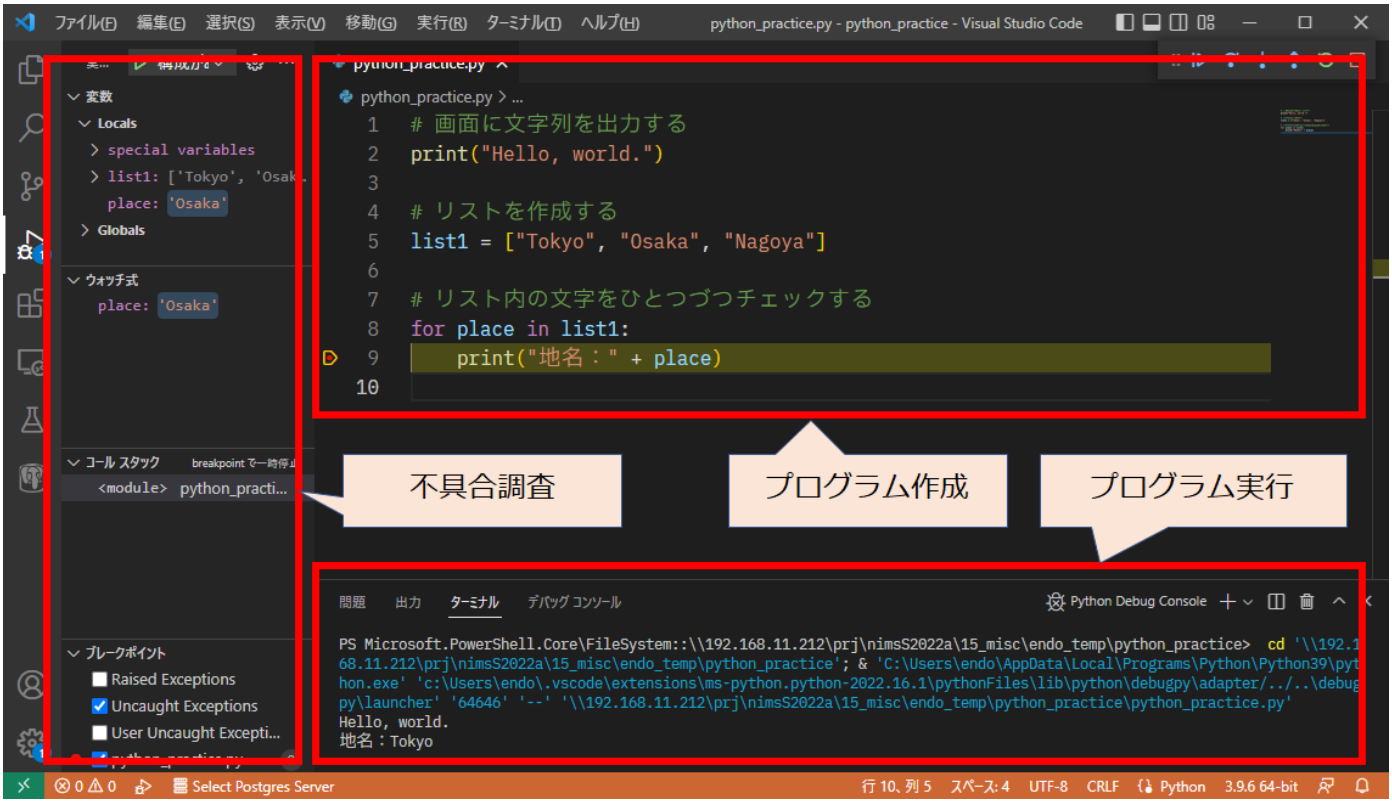
- 公式版Pythonに加え、データサイエンスをはじめとした、さまざまな分野でよく利用されるパッケージを同梱して配布されている、人気の高いPython実行環境です。
- パッケージやツールを自分でインストールする手間を軽減したい場合は、Anacondaを利用します。

エディタ

Pythonプログラムのソースコードを作成するための、代表的なエディタについてご紹介します。

Visual Studio Code

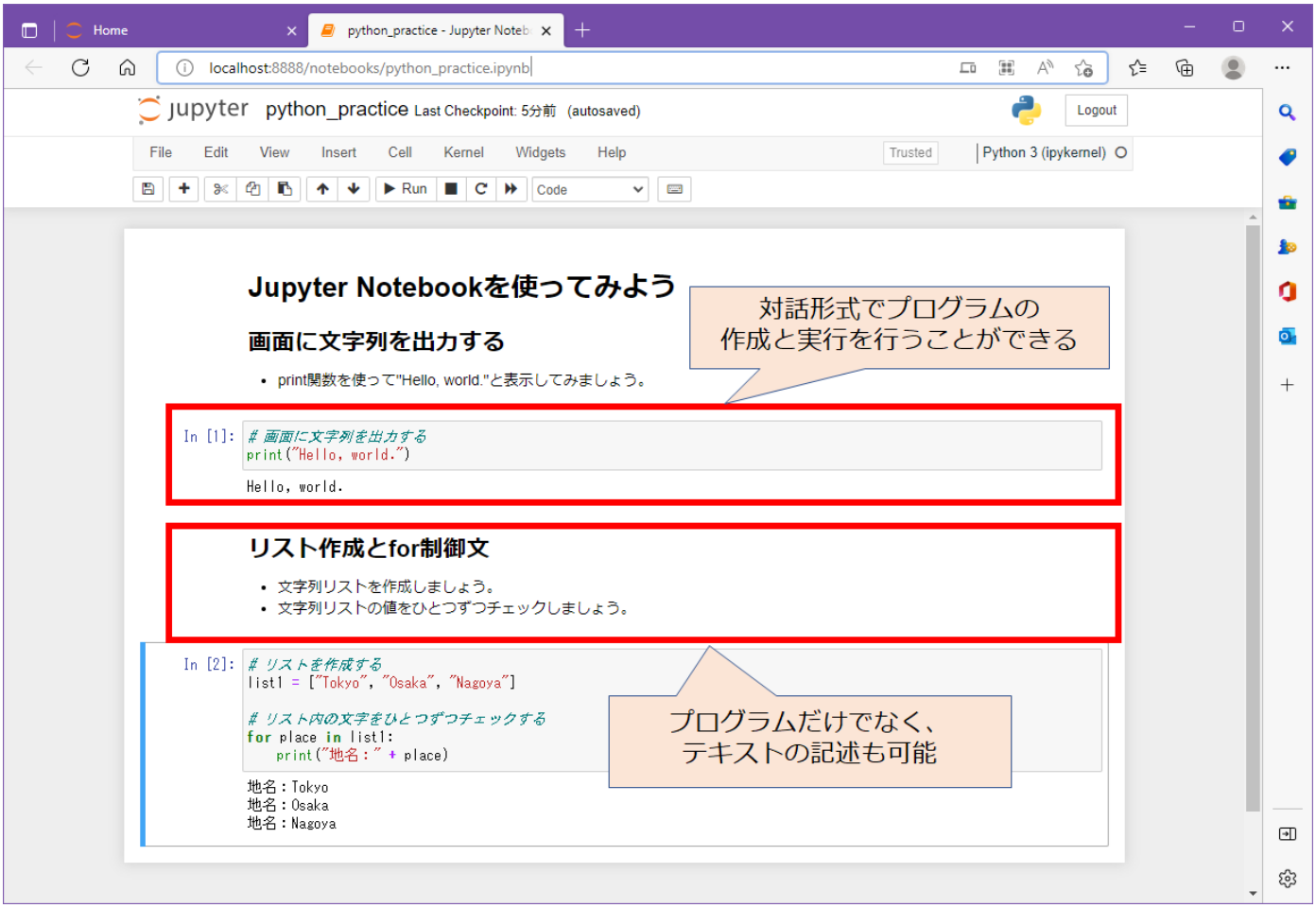
- Visual Studio Code(VSCode)は、Microsoftが開発しているエディタです。
- ソースコードの作成に加え、あらかじめPython実行環境をPCにインストールしておけば、Visual Studio Codeの画面上でプログラムを実行することもできます。
- 複数のソースコードをまとめて管理したり、プログラムの不具合を調査する便利な機能を持っています。そのため、大規模なプログラム開発ではVisual Studio Codeがおすすめです。



Jupyter Notebook

- Jupyter Notebookは、Webブラウザ上で動作するエディタ＋Python実行環境です。Webブラウザ上で動作します。
- 対話型のインタフェースで、Pythonプログラムのソースコード作成し、その場ですぐに実行まで行うことができます。
- ソースコードだけでなく、説明テキストなどを同時に記述できます。
- あらかじめ、Python実行環境をインストールしておく必要があります。
  - 公式版Pythonをインストールした場合、続けてJupyter Notebookのインストールを行います。

- Anacondaをインストールした場合、Jupyter Notebookが同梱されているため、すぐに利用開始できます。

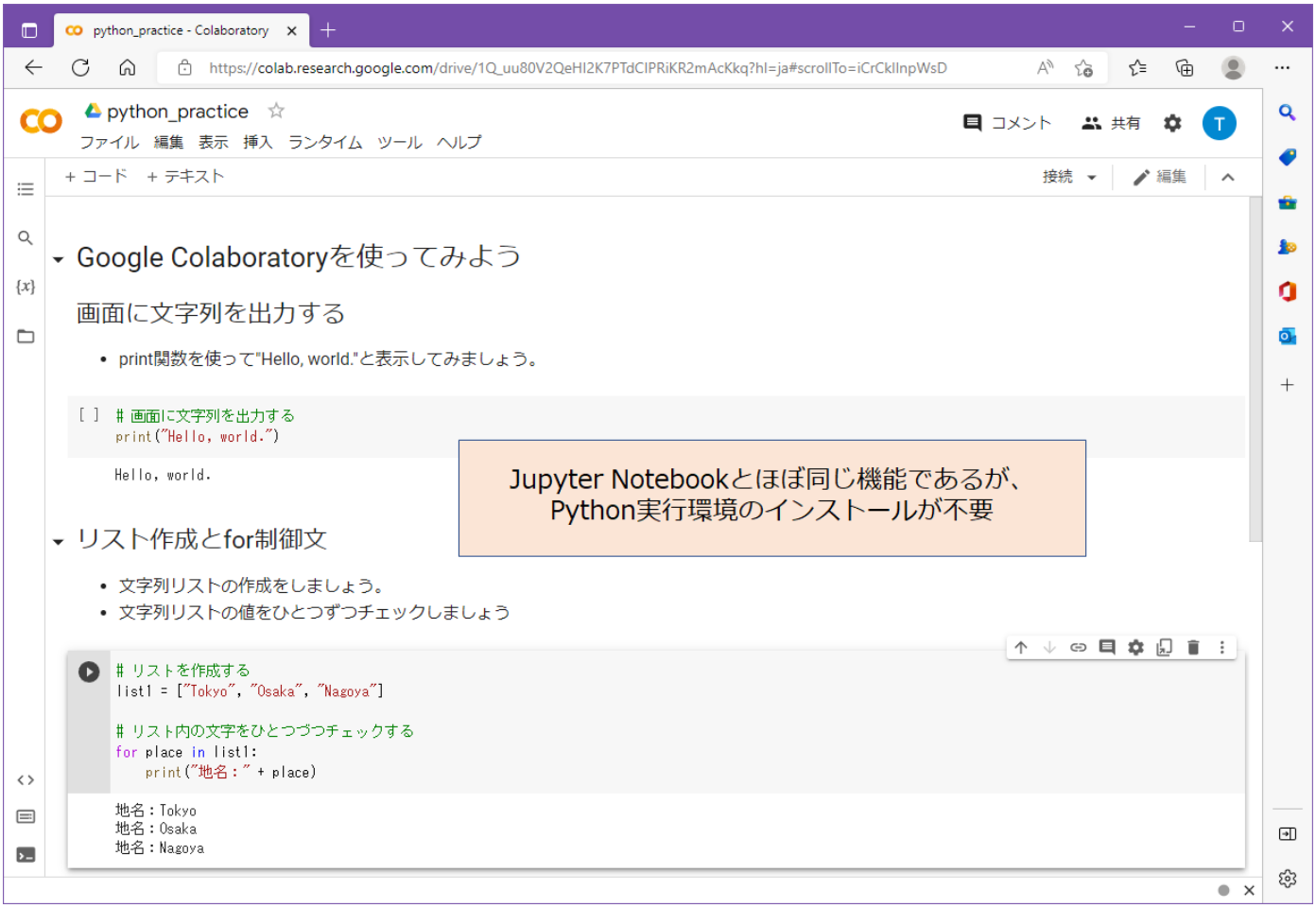


- 後継のエディタとして、JupyterLabがあります。まだ利用者やWebでの情報はJupyter Notebookの方が多い印象ですが、今後はJupyterLabが主流になっていく可能性があります。

【小ネタ】Jupyterの「py」は、Pythonの頭文字です。そのため、Jupyterの正しい読み方は「ジュパイター」だという意見もあります。しかし現実には、ほとんどの人が「ジュピター」と呼んでいるのが実態です。

Google Colaboratory

- Jupyter Notebookをベースに、Googleが開発したエディタ＋実行環境です。
- ソースコードの作成および実行、説明テキストの記述など、基本的な機能はJupyter Notebookとほぼ同じです。
- Python実行環境・エディタをPCにインストールする必要がありません。Googleアカウントと Webブラウザさえあれば、Pythonを実行できます。



本講座では、Google Colaboratoryを使用しています。

プログラムを動かしてみよう

ここから、いよいよPythonの基礎について学んでいきましょう。

実際にソースコードを見ながら、プログラムがどのように動いているのかを見ていきます。

文字列を出力する

以下のソースコードは、画面に"Hello, world."という文字列と、整数値の10を出力するプログラムです。

- ソースコードに「**print関数**」という命令が記述されています。print関数はPythonにあらかじめ用意された命令であり、()内に指定された文字列や数値などを画面に出力します。
- Pythonでは、文字列を"(ダブルクォート)または'(シングルクォート)で囲むルールになっています。整数値などの場合は、囲む必要はありません。

それでは実際に、プログラムを動かしてみましょう。

In [ ]:

```
print("Hello, world.")
print(10)
```

- ここでご紹介したprint関数のように、Pythonではよく使用する便利な機能を「**関数**」という形で提供しています。関数については、第2回でさらに詳しくご説明します。

【小ネタ】Pythonに限らず様々なプログラミング言語の入門書において、一番最初に学ぶプログラムが「画面に"Hello, world."と表示する。」であることが多いです。ある意味、最も有名なソースコードと言えるかもしれません。

変数を使用する

次のソースコードで、AAAやaaaは「**変数**」と呼ばれるものです。変数は、文字列や数値などの値を格納する入れ物のようなものです。

- 「変数名=値」と記述すると、変数に値が格納(代入)されます。=の前後に半角空白を入れても構いません。
- print関数は、()に文字列や数値を直接指定するだけでなく、変数を指定することもできます。変数を指定すると、変数を格納された値が画面に出力されます。

それでは実際に、プログラムを動かしてみましょう。

In [ ]:

```
AAA = "Hello, world."
aaa = 10

print(AAA)
print(aaa)
```

- 変数名は、大文字と小文字が区別されることに注意が必要です。変数名「AAA」と「aaa」は、別の変数として扱われます。
- 変数名には、英大文字、英小文字、数字、_(アンダーバー)を使用することができます。ただし、数字で始まる変数名は使用できません。様々な変数名の例を、次に示します。

In [ ]:

```
Hello_World = "Hello, world."
number10 = 10
_valueX = 123

print(Hello_World)
print(number10)
print(_valueX)
```

文字列の長さを求める

次のソースコードは、"Hello world."という文字列の長さを出力するプログラムです。

- ソースコードに「**len関数**」という命令が記述されています。len関数はPythonにあらかじめ用意された関数であり、()内に指定されたデータの長さを取得します。
- len関数で求めた文字列の長さを、str_lenという変数に格納しています。
- 変数str_lenに格納された値を、print関数で画面に出力しています。

それでは、実際にプログラムを動かしてみましょう。

In [ ]:

```
str_len  = len("Hello, world.")
print(str_len)
```

コメントを記述する

プログラムには、コメントを記述することができます。コメントはプログラムを読みやすくするための記述であり、プログラムの動作には何の影響も与えません。

#によるコメント

次のソースコードは、「変数を使用する」でご紹介したプログラムにコメントを追加したものです。

- #で開始する行は、行末までがコメントとなります。複数行のコメントを記述したい場合は、それぞれの行の先頭に#を記述する必要があります。

In [ ]:

```
# 変数に文字列を格納する
AAA = "Hello, world."

# 変数に数値を格納する
aaa = 10

# 変数の値を画面に出力する
# 変数名の大文字と小文字は区別されることに注意する
print(AAA)
print(aaa)
```

プログラムを見た人が内容を理解しやすいよう、積極的にコメントを書くことを心がけましょう。

docstringによるコメント

コメントには、#で開始するコメントのほかに、docstring(ドックストリング)というコメントも存在します。以下のソースコードは、docstringによるコメントを記述した例です。

- docstringは、後述する自作関数・クラスなどの、仕様や利用方法を記述するのに使用します。
- 関数やクラスの定義で、一番最初に記述した文字列は、docstringとなります。
- docstringでは多くの情報を見やすく記載するため、、"""(ダブルクォート3つ)、または'''(シングルクォート3つ)で囲んだ文字列を記述することが多いです。"Hello, world."のように1つのクォートで囲む場合、改行を含むには特殊な記述が必要です。3つのクォートで囲むと、改行をそのまま記述することができます。

In [ ]:

```
def remainder(v1, v2):
    """割り算の余りを求める関数です

    Args:
        v1 (int, float): 数値
        v2 (int, float): 数値

    Returns:
        rem (int, float): 余り
    """
    rem = v1 % v2
    return rem
```

【小ネタ】docstringは単にソースコードを読みやすくするだけではありません。最終回では皆さんに関数を作成していただきますが、ここでdocstringを書いておくと、help機能で自作関数を確認することができるのです！

In [ ]:

```
help(remainder)
```

代表的なデータ型

Pythonで扱うデータには、データの種類を示す「**型(type)**」という概念があります。

【ワンポイント】「型」の判別は、データ構造化の作業では大変に重要な作業となります。例えば、「2」という数字があった場合を考えます。これはプログラムで認識させるときに「文字」としての"2"であるのか、整数としての"2"であるのか、あるいは小数としての"2.0"として処理をするのがよいか。それぞれによって、プログラムの内容も変わります。それを、Pythonに限らず各種のプログラミング言語においては「型」で判別をします。

基本的なデータ型

まずは基本的なデータ型として、次の表に示すデータ型を紹介します。

データ型	Pythonプログラムでの表現
文字列	str
整数	int
浮動小数点数	float
論理値	bool

文字列(str)

"(ダブルクォート)または'(シングルクォート)で囲まれた値は、文字列となります。

次のプログラムを実行して、データの型を確認しましょう。「**type関数**」は、指定したデータの型が何であるかを取得します。

In [ ]:

```
# 文字列型の値を変数に格納する
str_value = "Hello, world."

# データの型を取得する
type(str_value)
```

「docstringによるコメント」でご紹介したように、"""(ダブルクォート3つ)、または'''(シングルクォート3つ)で囲むと、文字列に改行などの特殊な文字をそのまま記述できます。

In [ ]:

```
# 文字列型の値を変数に格納する
str_value = """Hello, world.
Hello, python"""

# 改行を含む文字列を出力する
print(str_value)

# データの型を取得する
type(str_value)
```

【注意事項 (とても重要！)】

上記のプログラムを実行すると、type関数の結果として「str」が画面に出力されました。実は、このようにtype関数の結果が画面に出力されるのは、Jupyter NotebookやGoogle Colaboratoryの独自機能によるものです。本来、type関数には画面に結果を出力する機能はありません。

type関数だけでなく、Jupyter NotebookやGoogle Colaboratoryでは、print関数を使わなくても計算結果などを画面に表示してくれることがあります。しかし、これはあくまでエディタの補助的な機能によるもので、本来は**データを画面に出力するときはprint関数を使用する**ことを必ず覚えておいてください。本来、type関数の結果を画面に出力するには、次のように記述します。

```
print(type(str_value))
```

以降では、print関数をきちんと使って、ソースコードを記述しています。

整数(int)

- 整数を値として使用できます。
- 次のプログラムを実行して、データの型を確認しましょう。

In [ ]:

```
# 整数型の値を変数に格納する
int_value = 10

# データの型を取得する
print(type(int_value))
```

- 上記は整数を10進数で記述しました。それ以外にも、2進数、8進数、16進数で記述することもできます。
  - 先頭に「0b」または「0B」を付けると、2進数で記述できます。
  - 先頭に「0o」または「0O」を付けると、8進数で記述できます。
  - 先頭に「0x」または「0X」を付けると、16進数で記述できます。

- それぞれの記述例を、次に示します。

In [ ]:

```
# 2進数
print(0b1011)

# 8進数
print(0o777)

# 16進数
print(0x1A)
```

【ワンポイント】では、ここで次のようしたらどうなるでしょうか？

In [ ]:

```
int_value = "10"
print(type(int_value))
```

同じ半角の10であっても、"10"とダブルクォートで囲むことで、文字列となりましたね。

浮動小数点数(float)

- 浮動小数点数を値として使用できます。
- 次のプログラムを実行して、データの型を確認しましょう。

In [ ]:

```
# 浮動小数点数型の値を変数に格納する
float_value = 3.14

# データの型を取得する
print(type(float_value))
```

論理値(bool)

- 論理値としてTrueまたはFalseのいずれかを値として使用できます。先頭のT、Fのみ大文字とする点に注意してください。
- 次のプログラムを実行して、データの型を確認しましょう。

In [ ]:

```
bool_value = True

# データの型を取得する
print(type(bool_value))
```

型の変換

Pythonでは、データ型を変換する関数が用意されています。

次のソースコードでは、「**str関数**」を使って整数型の値を文字列型に変換しています。

In [ ]:

```
# 整数型の値を定義
int_value = 100
print(type(int_value))

# str関数を使って整数型の100を文字列型"100"に変換する
int_to_str = str(int_value)
print(type(int_to_str))
```

次のソースコードでは、「**int関数**」を使って文字列型の値を整数型に変換しています。

In [ ]:

```
# 文字列型の値を定義
str_value = "999"
print(type(str_value))

# int関数を使って文字列型の"999"を整数型999に変換する
str_to_int = int(str_value)
print(type(str_to_int))
```

str関数およびint関数以外にも、それぞれのデータ型に対応した変換用の関数が用意されています。

複数の値からなるデータ型

次に、複数の値からなるデータ型として、次の表に示すデータ型を紹介します。

データ型	Pythonプログラムでの表現
リスト	list
タプル	tuple
辞書	dict
集合	set

このようなデータ型は、「**コンテナ型**」とも呼ばれます。

Pythonの入門書や、Python以前からあるプログラミング言語などでは、「配列」と呼ぶこともある内容で、データの格納方法を示す概念です。

この中でも**リスト**と**辞書**はデータ構造化の中でも最も頻繁に使います。しっかりと理解をしましょう。

リスト(list)

リストは、複数の値を順序付けられた一つのデータとして扱うことができます。

- リストは、[値1,値2,値3,・・・]という形式で表現します。,(カンマ)の前後に空白を入れても構いません。
- 値の重複を許可します。

次のプログラムを実行して、データ内容と型を確認しましょう。

In [ ]:

```
# リストを作成する 値が重複していてもよい
num_list = [1, 3, 5, 5, 7]

# リストの内容を画面に出力する
print(num_list)

# データの型を画面に出力する
print(type(num_list))
```

リストの値は、上記の整数型だけでなく、任意のデータ型とすることができます。次のソースコードは、様々なデータ型の値を持つリストの例です。

- リストを値として持つリスト(入れ子のリスト)とすることもできます。
- 1つのリスト内に複数のデータ型を混在させることもできます。よく理解して使えば便利な反面、きちんとしたドキュメントなどでの説明がないと、後に修復できないほどのエラーを生じさせる要因ともなります。気を付けましょう。

In [ ]:

```
# 文字列型のリスト
str_list = ["Tokyo", "Osaka", "Nagoya"]
print(str_list)

# リストのリスト(入れ子のリスト)
list_list = [[1, 3], [5, 5, 7]]
print(list_list)

# 複数のデータ型が混在したリスト
multi_list = [10, "Tokyo", True, [1, 3]]
print(multi_list)
```

リストについては、第4回でも詳しく説明します。

タプル(tuple)

タプルはリストと似ていますが、次の点が異なります。それ以外は、リストと同様に使用することができます。

- タプルは、(値1,値2,値3,・・・)という形式で表現します。,(カンマ)の前後に空白を入れても構いません。
- タプルを作成後、タプルに含まれる値を変更することができません。そのため、プログラムの不具合などによる意図しない変更を防ぐことができます。
  - 【ご参考】このように、作成後に値を変更できない性質を「イミュータブル」といいます。逆に、リストのように作成後に値を変更することができる性質を「ミュータブル」と言います。
- 同じ処理をリストとタプルで記述したケースを比較すると、基本的にタプルの方がプログラムの実行速度が速くなります。
- このあとご紹介する「辞書」のキーとしてリストを使用することはできませんが、タプルは使用することができます。

次のプログラムを実行して、データ内容と型を確認しましょう。

In [ ]:

```
# タプルを作成する。値の重複を許可する
num_tuple = (0, 1, 2, 3, 4, 5, 6, 7)

# タプルの内容を画面に出力する
print(num_tuple)

# データ型を確認する
print(type(num_tuple))
```

辞書(dict)

辞書はキー(key)と値(value)のペアを複数集めたデータです。

- 辞書は、{キー1:値1,キー2:値2,キー3:値3, ・・・}という形式で表現します。,(カンマ)や:(コロン)の前後に空白を入れても構いません。
- キーには文字列以外にも何種類かの型を使用できますが、一般的には文字列を使用します。
  - キーは、作成後に値を変更できない「イミュータブル」な性質を持ちます。
- 値には、任意の型を使用できます。
- 辞書に含まれるキーと値のペアには、基本的に順序(何番目のペア、などの概念)はありません。
  - 厳密には、順序の概念があるOrderdDictという機能があったり、比較的新しいPythonでは順序を意識したプログラムを書くこともできますが、今回は詳しい説明はいたしません。
- 値の重複を許可しますが、キーの重複は許可しません。

次のプログラムを実行して、データ内容と型を確認しましょう。

In [ ]:

```
# 辞書を作成する
mydict = {"key1": "value1", "key2": "value2", "key3": "value3"}

# 辞書の内容を画面に出力する
print(mydict)

# データ型を確認する
print(type(mydict))

# キーや値には文字列以外のデータ型も使用できる
multi_mydict = {"key1": "value1", 2: 10, "key3": [1, 2, 3]}
print(multi_mydict)
```

【小ネタ】計測にかかるメタデータの構造化では、この辞書型を必須として使っています。例えばSEMのメタデータではkey='倍率'、value=10000といったように格納します。様式2で選定したメタデータは 、ここで活用されます。

辞書ついては、第4回でも詳しく説明します。

次のソースコードでは、タプルのところでご紹介したように、辞書のキーとしてタプルを使用しています。一方、辞書のキーとしてリストは使用できません。

In [ ]:

```
# 辞書を作成する
mydict = {("key1-1", "key1-2"): "value1", ("key2-1", "key2-2"): "value2"}

# タプルの内容を画面に出力する
print(mydict)
```

集合(set)

集合は、値の集合を示すデータです。

- 集合は、{値1,値2,値3,・・・}という形式で表現します。,(カンマ)の前後に空白を入れることもできます。
- 集合に含まれる値には、順序(何番目の値、など)はありません。
- 値の重複を許可しません。

次のプログラムを実行して、データ内容と型を確認しましょう。

In [ ]:

```
# 集合を作成する
num_set = {1, 3, 5, 7}

# 集合の内容を画面に出力する
print(num_set)

# データ型を確認する
print(type(num_set))
```

次のソースコードは、3つの集合の積を求めています。

- 集合1&集合2&集合3・・・と記述すると、集合の積を求めることができます。&(アンド)の前後に半角空白を入れることもできます。

In [ ]:

```
# 3つの集合を作成する
setA = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
setB = {2, 4, 6, 8, 10}
setC = {1, 2, 4, 8}

# 積集合を求める
setP = setA & setB & setC

# 求めた積集合を画面に出力する
print(setP)
```

要素の数の数え方や選び方

要素の数の数え方

次のソースコードのように、len関数に複数の値からなるデータ型のデータを指定すると、データに含まれる値の数を取得することができます。

In [ ]:

```
# 値の数が5個のリストの長さを取得
str_list = ["Tokyo", "Osaka", "Nagoya", "Fukuoka", "Hokkaido"]
len_str_list = len(str_list)
print(len_str_list)

# 値の数が4個の辞書の長さを取得
int_dict = {"key1": 1, "key2": 2, "key3": 3, "key4": 4}
len_int_dict = len(int_dict)
print(len_int_dict)

# 値の数が3個のタプルの長さを取得
int_tuple = (10, 20, 30)
len_int_tuple = len(int_tuple)
print(len_int_tuple)

# 値の数が2個の集合の長さを取得
int_set = {100, 200}
len_int_set = len(int_set)
print(len_int_set)
```

要素の選び方 (任意の一つの値を取得)

次のソースコードでは、リストに含まれる値のうち、任意の一つの値を取得しています。

- リストから特定の1つの要素を取得する場合、リストに続く[]の中に、何番目の要素を取得するかを示す整数を記述します。
- 何番目の要素かを示す整数は、0から始まる点に注意が必要です。
- マイナスの整数を指定すると、「後ろから何番目」という意味になります。

In [ ]:

```
# リストを作成
# 0番目が"Tokyo"、1番目が"Osaka"、2番目が"Nagoya"となる点に注意
str_list = ["Tokyo", "Osaka", "Nagoya"]

# 2番目の要素を取得する
print(str_list[2])

# 後ろから2番目の要素を取得する
# -3番目が"Tokyo"、-2番目が"Osaka"、-1番目が"Nagoya"となる
print(str_list[-2])
```

タプルについても、リストと同じように値を取得できます。

In [ ]:

```
# タプルを作成
str_tuple = ("Tokyo", "Osaka", "Nagoya")

# 2番目の要素を取得する
print(str_tuple[2])

# 後ろから2番目の要素を取得する
print(str_tuple[-2])
```

辞書には順序の考えがないため、このように順序指定により値を取得することはできません。

辞書については、次のいずれかの方法で任意の一つの値を取得できます。

- 辞書名に続く[]の中に、値を取り出したいキーを指定します。この場合、存在しないキーを指定するとプログラムがエラーとなります。
- 辞書名.get(キー)と指定します。この場合、存在しないキーを指定すると、値がないことを示すNoneというものが取得されます。プログラムはエラーになりません。
- 辞書名.get(キー, デフォルト値)と指定します。存在しないキーを指定すると、デフォルト値が取得できます。

In [ ]:

```
# 辞書を作成する
str_dict = {"key1": "value1", "key2": "value2", "key3": "value3"}

# キー"key2"に対応する値を取得する
# 存在しないキーを指定するとプログラムがエラーとなる
print(str_dict["key2"])

# get(キー)では存在しないキーを指定するとNoneとなる
print(str_dict.get("key2"))
print(str_dict.get("keyX"))

# get(キー, デフォルト値)では存在しないキーを指定するとデフォルト値となる
print(str_dict.get("key2", "default_value"))
print(str_dict.get("keyX", "default_value"))
```

要素の選び方 (スライス)

次のソースコードは、リストのうち一部の要素を切り出した別のリストを取得しています。このような操作を「**スライス**」と呼びます。

- スライスは、リストに続く[]の中に開始点や終了点などを整数で指定します。次のような指定ができます。
  - リスト[X:Y]と指定すると、リストのX番目から(Y-1)番目までの要素を取得できます。
  - リスト[X:]と指定すると、リストのX番目から最後までの要素を取得できます。
  - リスト[:Y]と指定すると、リストの最初から(Y-1)番目までの要素を取得できます。
  - リスト[X:Y:D]と指定すると、リストのX番目から(Y-1)番目までの要素をD個おきに取得できます。

In [ ]:

```
# リストを作成
num_list = [0, 1, 2, 3, 4, 5, 6, 7]

# リストの3番目から5番目までの値を取得
# 5番目までの値を取得する場合は、終了点を6とする必要がある点に注意
print(num_list[3:6])

# リストの4番目から最後までの値を取得
print(num_list[4:])

# リストの最初から6番目までの値を取得
# 6番目までの値を取得する場合は、終了点を7とする必要がある点に注意
print(num_list[:7])

# リストの1番目から5番目までの値を2個おきに取得
# 5番目までの値を取得する場合は、終了点を6とする必要がある点に注意
print(num_list[1:6:2])
```

タプルについても、リストと同じようにスライスを使用できます。

In [ ]:

```
# タプルを作成
num_tuple = (0, 1, 2, 3, 4, 5, 6, 7)

# タプルの3番目から5番目までの値を取得
print(num_tuple[3:6])

# タプルの4番目から最後までの値を取得
print(num_tuple[4:])

# タプルの最初から6番目までの値を取得
print(num_tuple[:7])

# タプルの1番目から5番目までの値を2個おきに取得
print(num_tuple[1:6:2])
```

なお、集合については要素を選ぶ機能はありません。

代表的な演算子

Pythonでは、数値の四則演算をはじめとして、様々な演算子が用意されています。ここでは、よく使用する代表的な演算子を紹介します。

算術演算子

算術演算子として、次の演算子を使用できます。

演算子	機能
x + y	加算
x - y	減算

演算子	機能
x * y	乗算
x / y	除算
x % y	xをyで割った余り
x ** y	xのy乗
x // y	切り捨て除算

なお、()による計算順序の指定もできます。

次のソースコードは、算術演算子を使用した計算プログラムの例です。

- 算術演算子は2項演算子であり、「値1 演算子 値2」という形式で記述します。演算子の左右には半角空白を入れることができます。

In [ ]:

```
int_num = 10

# 5を加算
print(int_num + 5)

# 3で割った余り
print(int_num % 3)

# 2乗
print(int_num ** 2)

# ()を用いた計算順序の指定
print((int_num + 5) * 10)
```

整数型だけでなく、浮動小数点数型の計算にも使用できます。

In [ ]:

```
float_num = 3.14

# 浮動小数点数の乗算
diameter = 1.5
print(diameter * float_num)
```

文字列演算子

文字列演算子として、次の演算子を使用できます。

演算子	機能
s + t	文字列sと文字列tを連結
s * t	文字列sをt回繰り返す

次のソースコードは、文字列演算子を使用した計算プログラムの例です。

- 文字列演算子は2項演算子であり、「値1 演算子 値2」という形式で記述します。演算子の左右には半角空白を入れることができます。

In [ ]:

```
# 文字列の連結
print("abc" + "def")

# 文字列の繰り返し
print("abc" * 3)
```

【小ネタ】文字の足し算は、ファイルの出力のときに多用しています。例： "sample"という名前のファイル名を".csv"を付けてsaveするときは、("sample"+".csv")とします。

代入演算子

代入演算子として、次の演算子を使用できます。

演算子	機能
x = n	xにnを代入
x += n	x = x + nと同じ
x -= n	x = x - nと同じ
x *= n	x = x * nと同じ
x /= n	x = x / nと同じ
x %= n	x = x % nと同じ
x **= n	x = x ** nと同じ

次のソースコードは、代入演算子を使用した計算プログラムの例です。

- 代入演算子は、「変数名 演算子 代入する値」という形式で記述します。演算子の左右には半角空白を入れることができます。

In [ ]:

```
# =で代入
x = 5
print(x)

# +=で代入
# x = x + 10と同じ動作
x += 10
print(x)
```

比較演算子

比較演算子として、次の演算子を使用できます。

演算子	機能

演算子	機能
x == y	xがyと等しい(=が2つであることに注意)
x != y	xがyと異なる
x < y	xがyよりも小さい
x > y	xがyよりも大きい
x <= y	xがy以下である
x >= y	xがy以上である

比較演算子は、比較結果が成り立つときに論理値のTrue、成り立たないときに論理値のFalseとなります。

次のソースコードは、比較演算子による比較結果がTrueとなるかFalseとなるかを確認しています。

In [ ]:

```
num_value = 10

# 比較結果が成り立つ例
print(num_value == 10)

# 比較結果が成り立たない例
print(num_value != 10)
```

なお、==および!=と似ている比較演算子として、次の比較演算子もあります。

演算子	機能
x is y	xがyと等しい
x is not y	xがyと異なる

これらの演算子は、==および!=と厳密には同じではありません。次のソースコードを実行してみましょう。

In [ ]:

```
# 内容が同じ2つのリストを作成
list1 = [1, 2, 3]
list2 = [1, 2, 3]

# ==とisそれぞれを用いて比較
print(list1 == list2)
print(list1 is list2)
```

- ここでは詳しくは説明しませんが、別々に作成した2つのリストは、中に含まれる値が同じであってもPythonの中では別のデータとして管理されています。
- ==はリストに含まれる値しか比較しませんが、isはPythonの中で同じデータとして扱われているかを比較します。
- 今のところは、isおよびis notではなく、==および!=を使って等しい・等しくないの判定を行うようにしましょう。

比較演算子として、次のような演算子も使用できます。

演算子	機能

演算子	機能
x in y	xがyに含まれる
x not in y	xがyに含まれない

使用例として、次のソースコードのように、リスト中にある値が含まれるかどうかを判定できます。

In [ ]:

```
list1 = [1, 2, 3]

# list1の値に2が含まれるか
print(2 in list1)

# list1の値に5が含まれるか
print(5 in list1)

# list1の値に2が含まれないか
print(2 not in list1)

# list1の値に5が含まれないか
print(5 not in list1)
```

inおよびnot inは、文字列の操作にも使用でき来ます。詳細は、第2回の「文字列の操作」で紹介します。

また、比較演算子のより具体的な使用例は、第2回の「代表的な制御構文」でご説明します。

ブール演算子

ブール演算子として、次の演算子を使用できます。

演算子	機能
x and y	xもyもTrueであればTrue
x or y	xまたはyがTrueであればTrue
not x	xがFalseであればTrue

ブール演算子は、比較演算子と組み合わせて複雑な条件比較を行うことなどに使用します。次のソースコードは、比較演算子とブール演算子を組み合わせて、結果がTrueとなるかFalseとなるかを確認しています。

In [ ]:

```
num_value = 10
str_value = "abc"

# 比較結果が成り立つ例
print(num_value == 10 and str_value == "abc")
print(not num_value == 20)

# 比較結果が成り立たない例
print(num_value != 10 and str_value == "abc")
print(num_value == 20 or str_value == "def")
```

第1回の講義は、ここまでとなります。次回は、次の内容をご紹介します。

- 代表的な制御構文
- 文字列の操作
- 予約語
- 関数とメソッド
- 例外
- 注意すべき用語