第3回 パッケージとモジュール・ファイル操作

第1回、第2回の講義を通して、Pythonの基本的な構文について学びました。

第3回では、データ構造化処理では決して欠かせないファイルの読み書きを学びます。

また、Pythonプログラミングにおいて最も重要な、外部のモジュールやパッケージを利用したプログラムについても見ていきます。

今回学ぶ内容は、次のとおりです。

- モジュール・パッケージの利用
- 基本的なファイルの読み書き
- ファイルパス

はじめに、本講義で使用するファイルを皆さんの環境にダウンロードするため、次のコードを実行してください。

In [ ]:

```
!wget https://github.com/tendo-sms/python_seminar_2022/raw/main/lecture3/files.zip .
!unzip files.zip
!mv files/* .
```

モジュール・パッケージの利用

ここでは、外部のモジュールやパッケージを利用する方法についてご紹介します。

モジュールとパッケージのおさらい

第2回にて、モジュールおよびパッケージという用語をご説明しました。ここでおさらいしておきましょう。

用語	意味
モジュール	ソースコードを記述した拡張子「.py」のファイル
パッケージ	ある機能を提供するために必要な、複数のモジュールをまとめたもの

モジュールやパッケージを利用する意味【超・重要！！】

実は、**「外部のモジュールやパッケージの利用」こそPythonプログラミングの真髄**、と言っても過言ではありません。

モジュールやパッケージを利用するということは、「誰かが作った外部のソースコードを使わせてもらう」というイメージです。

Pythonは他のプログラミング言語と比べて、外部のモジュールやパッケージが非常に充実しています。

皆さんが苦労することは、世界中の誰もが同じように苦労します。そして、苦労した誰かが必ず、便利なソースコードを作っているものなのです。それらのソースコードを簡単に取り込める方法こそが、これから紹介するimportなのです。

苦労して作った数百行のプログラムが、誰かが公開しているソースコードを使ったらたった2～3行で実現できた、なんて例も珍しくありません。「車輪の再発明」に陥らないよう、しっかりと今回の話を理解してくださいね。

次の例を見てみてください。Gatanのdm3ファイルに記録されたスペクトルデータをグラフ化するプログラムです。

あらかじめHyperSpyというパッケージをインストールしてしまえば、ソースコード自体はたったの3行でかけてしまうのです。(外部のモジュールを取り込むimportの行を除けば、メインの処理は実質2行！)

In [ ]:

```
# 事前にHyperSpyパッケージのインストール
!pip install hyperspy
```

In [ ]:

```
# Gatanのdm3ファイルに記録されたスペクトルデータをグラフ化
import hyperspy.api as hs

spectrum = hs.load("spectrum_sample.dm3")
spectrum.plot()
```

ちなみに、スペクトルのグラフ化にはExcelを使う方が多いかもしれません。

しかし、dm3フォーマットを読み解いて、加工して、Excelで読み込んで、マウスでポチポチとグラフを作成するのと比べてどうでしょうか。Pythonの底力を感じませんか！？

プログラミングというと、複雑なアルゴリズムを設計して、たくさんのソースコードを書かなければいけない、というイメージが強いかもしれません。

しかし、Pythonであれば、まるでブロックを組み立てるようにプログラムを作成できてしまいます。皆さんの中でパラダイムシフトを起こして、プログラミングに対するハードルをぐぐっと下げてくださいね。

モジュールのインポート(import)

ここでは、外部の「モジュール」を読み込んで使ってみます。

モジュールの読み込みには、次のとおり「**import文**」を使います。

```
import モジュール名(拡張子.pyはつけない)
```

「モジュール名」には、モジュールのファイル名「モジュール名.py」から拡張子.pyを取り除いた名称を記述します。

モジュールをimportすると、次のようにモジュール内の関数、変数などを呼び出せるようになります。

```
モジュール名.関数名(引数)
モジュール名.変数名
```

Pythonで用意されたモジュールをインポートする

import文の簡単な例として、まずはPythonであらかじめ用意された標準モジュールである、「mathモジュール」をインポートして使ってみましょう。

mathモジュールは、数学的な機能を提供するモジュールです。

まずは、関数を呼び出してみます。

次のソースコードは、平方根を求めるためにmathモジュールのsqrt関数を呼び出しています。「モジュール名.関数名(引数)」の形になっていますね。

In [ ]:

```
import math

print(math.sqrt(2))
```

次に、変数を呼び出してみましょう。

次のソースコードは、円周率の値が格納された変数piを呼び出しています。「モジュール名.変数名」の形になっていますね。

In [ ]:

```
print(math.pi)
```

別名を付けてインポートする

次のようなインポートの仕方もあります。

```
import モジュール名 as 別名
```

このようにインポートすると、モジュール名ではなく、別名で関数や変数などを使用できるようになります。

(逆に、モジュール名は使用できなくなります。)

```
別名.関数名(引数)
別名.変数名
```

ここまで見てきたmathの例で、別名を使ってインポートしてみましょう。「mt」という別名を付けています。

In [ ]:

```
import math as mt

print(mt.sqrt(2))
print(mt.pi)
```

mathはモジュール名が短いのであえてasを使う必要はありませんが、モジュール名が長い場合などはasを使って、記述しやすい別名を付けるのもよいでしょう。

自作のモジュールをインポートする

Pythonであらかじめ用意されたモジュールや、他の誰かが公開している外部モジュールだけではなく、自分でモジュールを作ってインポートすることもできます。

最初に今回の講義で必要なファイルをダウンロードしたので、カレントディレクトリにsample_module.pyというファイルがダウンロードされています。ファイルの中身は、次のような内容です。

```
    def print_function():
        print("関数print_functionが呼び出されました。")

    var1 = "変数1"
    var2 = "変数2"
```

モジュールをインポートする方法は、自作のモジュールであっても同じです。次のソースコードでは、モジュールsample_module.pyをインポートして、print_function関数の呼び出し、およびvar1変数とvar2変数の値の参照を行っています。

In [ ]:

```
import sample_module

sample_module.print_function()

print(sample_module.var1)
print(sample_module.var2)
```

【ワンポイント】

モジュールを自作するというのは、どのようなケースでしょうか。

例えば構造化プログラムは、そのほとんどが「データの入力」「データの構造化」「データの出力」という3つの大きな機能からなります。

これらの機能をソースコードの上から下へ、ダラダラと書いていくのはやめましょう。

この3つの機能それぞれを関数にするというやり方も悪くありませんが、これらは一つ一つが大きな機能なので、各機能ごとに自作のモジュールを作る方がよいでしょう

例：

- データの入力：input_data.py
- データの構造化：structure_data.py
- データの出力：output_data.py

そして各機能をさらに細かい機能に分割し、分割した機能ごとに関数を作ると、大きなプログラムでも非常に理解しやすい構造になります。例えば、次のようなイメージです。

- input_data.pyの中に、次のような関数を定義
  - jdxファイルの読み込み関数、CSVファイルの読み込み関数 etc.
- structure_data.pyの中に、次のような関数を定義
  - データのフォーマットを変換する関数、データを整形する関数 etc.
- output_data.pyの中に、次のような関数を定義
  - メタデータをファイルに出力する関数、グラフを出力する関数 etc.

特定の関数や変数だけをインポートする

モジュール全部ではなく、モジュールの中の特定の関数や変数だけをインポートすることもできます。

from ～ import文を使って、次のように定義します。

```
    from モジュール名 import 関数名or変数名 [, 関数名or変数名, ・・・]
```

importする関数や変数が複数あるときは、カンマ区切りで複数指定することができます。

また、この方法でimportした関数や変数は、前に「モジュール名.」を付けずに呼び出します。

```
関数名(引数)
変数名
```

先ほどのsample_module.pyから、関数print_functionと、変数var2だけをインポートして使ってみましょう。

In [ ]:

```
from sample_module import print_function, var2

print_function()

print(var2)
print(var1)
```

関数名や変数名の前に「sample_module.」を付けないことに注意してください。

var1はインポートしていないので、参照しようとするとエラーになっています。

パッケージのインストール

pipコマンドとcondaコマンド

今度は、ある機能を実現するためのモジュールをまとめた、外部の「パッケージ」を使ってみましょう。

パッケージを利用するには、Pythonプログラムを実行する前に、あらかじめ端末(Windowsのコマンドプロンプト、Linuxのログインシェルなど)で次のコマンドを実行してパッケージをインストールしておく必要があります。

- 公式版Pythonでは、「**pipコマンド**」でインストールします。
- Anacondaでは、「**condaコマンド**」でインストールします。

Python実行環境をOSにインストールすると、Windowsのコマンドプロンプトや、Linuxのログインシェルなどから、pipコマンドおよびcondaコマンドを使用できるようになります。

パッケージをインストールするには、次のように実行します。なお「パッケージ名」は大文字小文字を区別しません。

```
pip install パッケージ名
```

または

```
conda install パッケージ名
```

今回の講座では、以降はpipコマンドを例にご説明します。

【注意事項】

pipコマンドでしかインストールできないパッケージや、condaコマンドでしかインストールできないパッケージも存在します。

ご自身の環境で使いたいパッケージがない場合、上記のコマンドを使わずに自力でインストールできることもありますが、非常にハードルが高いです。代替可能な別のパッケージを探すか、Python実行環境の変更も検討しましょう。

なおAnacondaはcondaコマンドだけでなく、pipコマンドも使用できます。そのためAnacondaを使っておけば安心と思いがちです。しかしpipコマンドとcondaコマンドを混在して使用するとやっかいなトラブルが発生することもあるので、あまりお勧めできません。

pipコマンドの使用例

例として、PythonでGUIのプログラムを作成するときによく使用される、PyQt5のパッケージをインストールしてみます。

(本当は、本講座で詳しくご紹介する「pandas」や「Matplotlib」といったパッケージをインストールする例としたかったのですが・・・Google Colaboratoryでは、最初からこれらのパッケージがインストールされていました！)

In [ ]:

```
!pip install pyqt5
```

なお先頭の「!」は、Google Colaboratory独自の構文です。Pythonプログラムを実行するのではなく、Linuxのログインシェルのイメージで、bashのコマンドや、catコマンドやdiffコマンドといったLinuxコマンドを実行できます。

pipコマンドの便利なところは、パッケージ間の依存関係を自動で管理してくれるところです。

パッケージには、「このパッケージを動かすには、前提として別のパッケージが必要」といった依存関係があります。

pipコマンドでパッケージをインストールすると、前提となるパッケージが入っていない場合は自動で一緒にインストールしてくれます。

基本的なファイルの読み書き

第1回の講義で、どんなデータ構造化プログラムでも次のような流れが基本となることをご説明しました。

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

ここでは、データの入力や出力に関わる重要な機能として、ファイルの読み書きの方法をご紹介します。

標準的なファイルの読み書き

まずは外部のモジュールやパッケージを使わずに、Python標準の関数を使ってデータを読み込んでみます。

次のプログラムを例にして説明します。読み込むファイルopen_sample.jdxは、JCAMP-DXのjdxファイルから、説明のために先頭のヘッダ部分のみを抜粋したものです。

In [ ]:

```
# ファイルのオープン
fobj = open("open_sample.jdx")

# ファイルの読み込み
txtdata = fobj.read()
print(txtdata)

# ファイルのクローズ
fobj.close()
```

このソースコードを構成するそれぞれの機能について、ひとつひとつ解説します。

ファイルのオープン(open関数)

Pythonでファイルの読み書きを行うには、まずファイルのオープンを行う必要があります。

例えば、Windowsでテキストファイルや Excelファイルの読み書きをするとき、まずはエクスプローラでファイルをダブルクリックするなどして、ファイルを開きますよね。そんなイメージです。

標準的なファイルのオープンでは、「**open関数**」を使用します。

```
open(ファイルパス)
```

open関数の戻り値として、「**ファイルオブジェクト**」というものが返ってきます(変数fobjに格納)。ファイルオブジェクトは、ファイルをPythonプログラム上で読み書きできる形にしたデータです。

このファイルオブジェクトを使って、データの読み書きを行います。

ファイルの読み込み(readメソッド)

ファイルからデータを読み込む方法はいくつかありますが、今回の例では最もシンプルな「**readメソッド**」を使用しています。

```
ファイルオブジェクト.read()
```

readメソッドの戻り値として、ファイルの中身が文字列として返ってきます(変数txtdataに格納)。

print関数でtxtdataの中身を画面に出力すると、ファイルの中身が格納されていることが分かります。

ファイルのクローズ(closeメソッド)

最後に、「**closeメソッド**」でファイルを閉じます。

```
ファイルオブジェクト.close()
```

Windowsでテキストファイルや Excelファイルの読み書きを終えたあと、閉じるボタンをクリックするなどしてファイルを閉じますよね。そんなイメージです。

closeメソッドでファイルを閉じた後は、もうreadメソッドで読み込んだりすることはできません。

ちなみにクローズを忘れてしまうと、例えば、長期間動き続けるようなプログラムにおいてファイルをオープンできる数の上限に達してしまい、プログラムがエラーになるなどの問題が発生します。

逆に言うと、上記のようなケース以外では、クローズを忘れても実際のところあまり実害はありません。

とはいえ、ファイルを使い終わったらクローズするのは、Pythonを含めた多くのプログラミング言語において、お作法のようなものです。必ずクローズする癖を付けましょう・・・

・・・と、言いたいところですが、いくら気を付けていても、ファイルのクローズは結構忘れてしまいがちです(ほぼ実害がないだけに)。クローズを入れていても、if制御文やfor制御文のbreakなどでスキップされてしまったなどのミスも起こりえます。

そこでPythonでは、ファイルのクローズを自動的に行うための便利な仕組みを提供しています。それが次でご紹介する「with文」です。

with文

「**with文**」を用いて次のように記述することで、インデントで示されたwith構文の影響範囲を抜けるときに、Pythonが自動でファイルをクローズしてくれます。

```
with open(ファイルパス) as ファイルオブジェクト:
    ファイルオブジェクトを使ってファイルを読み書きする処理
    ファイルオブジェクトを使ってファイルを読み書きする処理
                                :

後続の処理（ここではもうファイルオブジェクトを使えない）
```

前述のプログラムをwith文を使って書き換えると、次のようになります。

In [ ]:

```
with open("open_sample.jdx") as fobj:
    txtdata = fobj.read()
    print(txtdata)
```

これで、最後にcloseメソッドを呼ぶ必要がなくなりました。

with文は非常に便利ですが、稀に、with文を使わずにcloseメソッドを使った方が書きやすいケースもあります。

ですが現在のところは、「**open関数を使うときは必ずwith文を使う**」と覚えておいていただければ問題ありません。

1行ずつファイルを読み込む

前述のreadメソッドでは、ファイルの中身すべてを1つの文字列として読み込みました。

ですが構造化プログラムでは、機器が出力したファイルを1行ずつ読み込みながら処理する、といったことの方が多いです。

次のプログラムを見てください。

In [ ]:

```
with open("open_sample.jdx") as fobj:
    for txtdata in fobj:
        print("読み込んだ行：" + txtdata)
```

read関数を使わず、ファイルオブジェクト(fobj)をfor文の「繰り返し要素」に指定しています。

これで、ファイルの中身を1行ずつ読み込めた！・・・ようにも見えますが、ちょっと違和感がありますね。各行の間に、不自然な空行が入っています。

実はテキストファイルには、実際には以下のような形でデータが書き込まれています。テキストエディタ等では目に見えないだけで、【改行】というデータが入っているのです。

```
##TITLE= deg【改行】
##JCAMP-DX= 5.01【改行】
          :
##TIME= 10:29:30【改行】
##SPECTROMETER/DATA SYSTEM= DELTA2_NMR【改行】
```

注：##の後ろに半角空白が入って見えますが、実際のデータには入っていません。Google Colaboratoryの仕様で、どうしても半角空白が勝手に入って見えてしまうのです・・・

for文にファイルオブジェクトを指定して1行ずつ読み込むと、【改行】も含めた文字列が1行のデータとして読み込まれます(【改行】も含めて変数txtdataに格納される)。

加えて、print関数が画面に出力する際に最後に【改行】を挿入しています。そのため、print関数は次のように画面に文字列を出力しています。

```
##TITLE= deg【元からファイルに入っていた改行】【print関数が挿入する改行】
##JCAMP-DX= 5.01【元からファイルに入っていた改行】【print関数が挿入する改行】
          :
##TIME= 10:29:30【元からファイルに入っていた改行】【print関数が挿入する改行】
##SPECTROMETER/DATA SYSTEM= DELTA2_NMR【元からファイルに入っていた改行】【print関数が挿入
する改行】
```

改行がそれぞれ2つずつあるため、不自然な空行が入ってしまうのです。

これを防ぐには、読み込んだ1行の文字列にrstripメソッドを使いましょう。

【おさらい】

- 文字列.rstrip(削除文字列) と記述すると、文字列の末尾からだけ、連続した削除文字列をすべて削除します。削除文字列を指定しないと、連続した半角空白・タブ・改行を削除します。

In [ ]:

```
with open("open_sample.jdx") as fobj:
    for txtdata in fobj:
        print("読み込んだ行：" + txtdata.rstrip())
```

rstripメソッドにより、txtdataに格納された文字列の右端にあった【元からファイルに入っていた改行】が削除されました。

それにより、print関数は次のように画面に文字列を出力しました。

```
##TITLE= deg【print関数が入れた改行】
##JCAMP-DX= 5.01【print関数が入れた改行】
          :
##TIME= 10:29:30【print関数が入れた改行】
##SPECTROMETER/DATA SYSTEM= DELTA2_NMR【print関数が入れた改行】
```

これで、期待どおりの出力になりましたね。

【ワンポイント】

ちなみに、この【改行】として記録されているデータは、OSによって異なるデータが採用されています。Windowsは「CRLF」、LinuxやmacOSなどでは「LF」というデータになっています。(ちなみに、古いmacOSではまた別の「CR」というデータでした。)

ただしPythonでファイルを読み込むとき、特別に指定をしなければ、これらの違いを意識する必要はありません。どちらの改行コードも正しく読み取ることができます。

ファイルをオープンするときのモード

open関数には、モードを2番目の引数として渡すことができます。

```
open(ファイルパス, モード)
```

モードには次の種類があり、"r"などといった形で文字列として指定します。

モード	意味
"r"	読み込み用で開く(デフォルト値)
"w"	書き込み用で開く ファイルが存在する場合は上書き
"x"	書き込み用で開く ファイルが存在する場合はエラー
"a"	追加書き込み用で開く
"r+"	読み込みと書き込みの両方ができる　書き込みは追加書きとなる　ファイルが存在しない場合はエラー
"w+"	読み込みと書き込みの両方ができる　ファイルが存在する場合は上書き
"b"	"rb"や"wb"という形で指定してバイナリモードで読み書き

これまでのソースコードの例ではオプションを指定していなかったため、デフォルトの"r"となっていました。

ファイルの書き込み(writeメソッド)

今度は、ファイルを書き込み用(w)で開いて書き込みを行ってみましょう。

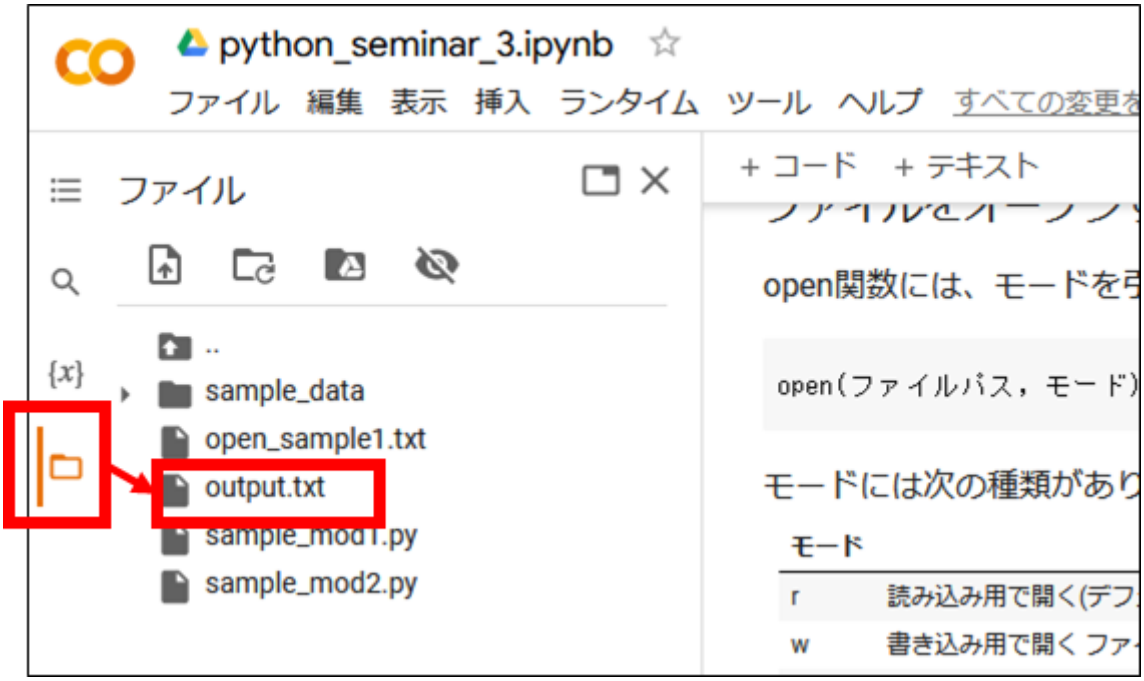
ここでは、「**writeメソッド**」を使って書き込みを行います。次のソースコードを見てみましょう。

In [ ]:

```
with open("output.txt", "w") as fobj:
    write_str ="データを書き込みます"
    fobj.write(write_str)
```

output.txtは、プログラム実行開始前には存在していないファイルです。"w"モードでファイルをオープンすることは、書き込み用という意味だけでなく、ファイルの新規作成を行うことも兼ねています。

writeメソッドの引数に文字列を指定すると、オープンしたファイルにその文字列が書き込まれます。Google Colaboratory画面左端のフォルダマークから、ファイルの中身を確認してみましょう。



データが書き込まれていることが確認できました。

なお、複数行書き込みたい場合の例を次に示します。

In [ ]:

```
with open("output.txt", "w") as fobj:
    write_str = "データを書き込みます１"
    fobj.write(write_str + "¥n")
    write_str = "データを書き込みます２"
    fobj.write(write_str + "¥n")
```

注意していただきたいのは、改行を自分で指定する必要がある点です。改行は、文字列中に「**\n**」と記述することで挿入できます。

(Google Colaboratoryですと、通常の説明文では「バックスラッシュ」、コード例では「円記号」として表示されますが、実態は同じ文字です。)

文字エンコーディングの指定

カレントディレクトリに、テキストファイル「encoding_sample.jdx」というファイルを置いてあります。このファイルの中身は、次のような内容です。

これまで使用した入力ファイル「open_sample.jdx」の内容に加え、最後の行にユーザー定義ラベルでオペレーターの名前が記録されています。

```
##TITLE= deg
##JCAMP-DX= 5.01
##DATA TYPE= NMR SPECTRUM
##DATA CLASS= NTUPLES
##NUM DIM= 1
##ORIGIN= DELTA2_NMR
##OWNER= delta
##DATE= 2021/10/02
##TIME= 10:29:30
##SPECTROMETER/DATA SYSTEM= DELTA2_NMR
##$OPERATOR= 髙橋太郎
```

この「encoding_sample.jdx」ファイルを読み込んでみます。

In [ ]:

```
with open("encoding_sample.jdx") as fobj:
    txtdata = fobj.read()
    print(txtdata)
```

プログラムがエラーになってしまいました。

実はencoding_sample.txtの内容は、文字エンコーディング(文字コード、文字エンコードなどと呼ぶこともあります。)が「CP932」です。

Google Colaboratoryは、Linux上でPythonプログラムが動いています。LinuxのPythonでは、デフォルトでは文字エンコーディングが「UTF-8」だとみなしてファイルを読み込もうとします。その結果、UTF-8でない文字が出てきたためにエラーとなりました。

LinuxでUTF-8以外の文字エンコーディングで記述されたテキストファイルを読み込む場合、open関数の引数に「**encodingオプション**」を指定します。次のソースコードを見てみましょう。

In [ ]:

```
with open("encoding_sample.jdx", encoding="cp932") as fobj:
    txtdata = fobj.read()
    print(txtdata)
```

今度は、正しくファイルを読み込むことができました。

文字エンコーディングは、初級者がつまづきやすい代表的なポイントです。次のようなことでハマりがちです。

- Pythonでファイルを読み込もうとしたらエンコーディングエラーになった
- Pythonで出力したCSVファイルをExcelで読み込んだら文字化けした

1点目は「エンコーディングエラー」を示すメッセージが出てエラーとなることから、比較的、原因が分かりやすいです。

しかし2点目は、とても分かりにくいです。Excelは、基本的に文字エンコーディングがCP932であるとみなしてファイルを読み込みます。

UTF-8のファイルも「BOM付きUTF-8」という形式なら正しく読み込めるのですが、Pythonのopen関数のデフォルト値であるUTF-8とは別の形式であるため、明示的に「BOM付きUTF-8」でオープンする必要があります。

文字エンコーディングの詳細については膨大な説明となってしまいますので割愛しますが、「**ファイルを読み書きするときは文字エンコーディングが何であるかを確認し、open関数のencodingオプションを適切に指定する**」という点だけは、必ず覚えておいてください。

【ワンポイント】

ちなみに「CP932」というのは、Microsoft社がShift-JISをベースに独自拡張した文字エンコーディングです。今回の例では、「髙 (はしご高)」がCP932の独自文字であり、標準のShift-JISには含まれない文字です。他にもよく出てくる独自文字には、「﨑 (立ち崎)」などもあります。

Windowsの世界で「Shift-JIS」という言葉を使っていても、実は「CP932」を指していることがほとんどです。

encodingオプションには"shift_jis"という指定もありますが、これは標準のShift-JISを示します。「髙 (はしご高)」を扱えない文字エンコーディングなので、前述のプログラムはエラーとなります。

「Windowsで作られたShift-JISのファイル」は、encoding="cp932"とすべきケースがほとんどでしょう。

練習問題

「基本的なファイルの読み書き」の最後に、練習問題にチャレンジしてみましょう。

カレントディレクトリに、「practice1.txt」というファイルがあります。内容は次のとおりです。

```
TITLE= MgO SPECTRUM
DATA TYPE= INFRARED SPECTRUM
LONG DATE= 2021/10/02 10:29:30
NPOINTS= ???
$OPERATOR= 山﨑花子
```

「NPOINTS」の値は???としていますが、実際には値が入っています。この???部分だけを、「practice1_out.txt」というファイルに出力するプログラムを作成してみましょう。("NPOINTS= "という部分や、他の行は出力しない)

ヒント：

本日学んだ内容のほか、第2回で学んだ以下の内容も活用してください。

- 条件により処理を変更する(if文)
- 文字列の分割(split)
- 文字列の先頭および末尾の文字を削除(strip、lstrip、rstrip)
- 文字列の判定(in、not in、startswith、endswith)

第2回講義資料のURL：

https://colab.research.google.com/drive/1hW1zR-T81I3HO_u0NyalmtAJIJ129zoc?usp=sharing
(https://colab.research.google.com/drive/1hW1zR-T81I3HO_u0NyalmtAJIJ129zoc?usp=sharing)

In [ ]:

```
# プログラムを作成してみましょう
```

パッケージによるCSVファイルの読み書き

ここでは、外部パッケージによるCSVファイルの読み書きについてご紹介します。

取り上げるのは、「**NumPy**」「**pandas**」という2つのパッケージです。それぞれの特徴を、次にまとめます。

- NumPy
  - 「行列計算(行列和・行列積など)」を得意とするパッケージです。
- pandas
  - 「配列データの整形・加工」を得意とするパッケージです。

データ構造化プログラムでは、機器が出力したデータなど、CSVファイルの入出力を行うことがとても多いです。

これらのパッケージを使って読み書きをすると、CSVファイルの内容を自動的に多次元のリスト(リストのリスト、入れ子のリスト)として扱うことができるなど、ここまでにご紹介したPython標準関数と比べて非常に便利です。

ちなみに、NumPyとpandasどちらも配列データを扱うパッケージということから、どのように使い分ければよいのか、初級者は悩みがちです。

前述の説明のとおり「行列計算をしたいのか」「データの整形・加工を行いたいのか」という観点で、適切なパッケージを選択しましょう。

特にデータ構造化プログラムでは、機器が出力したCSVデータを整形・加工してメタデータとして登録するなどの目的で、pandasをよく利用します。

NumPyでのCSVファイル読み書き

NumPyでCSVファイルの読み書きをやってみましょう。

モジュールのインポート

NumPyを利用するには、今回の講義でご紹介したimport文を使って、NumPyのモジュールをインポートします。モジュール名は小文字の「numpy」なので、間違えないようにしてください。

なおNumPyをインポートするときは、asを使って「np」という別名を付けるのが慣例になっています。Pythonプログラマの共通認識のようなものですので、慣例に倣うようにしてください。

```
import numpy as np
```

【注意事項】

Google ColaboratoryやAnacondaでは、NumPyが最初からインストールされているのでimport文だけで利用できます。公式版Pythonでは最初はNumPyがインストールされていませんので、プログラムを作成する前に、あらかじめpipコマンドを使ってインストールしてください。

```
pip install numpy
```

CSVファイルの読み込み

NumPyでCSVファイルを読み込むには、loadtxt関数を使います。

今回は、次のような内容のCSVファイルを読み込みます。

1 2 3

4 5 6

7 8 9

ちなみに、Python標準関数でファイルを読み込むときは、先にopen関数でファイルオープンした後にread関数で読み込みを行いました。

しかしloadtxt関数でCSVファイルを読み込むときには、ファイルオープンも一緒に行ってくれますので、事前にopen関数やwith文を使う必要はありません。同様にファイルのクローズも内部で自動的に行われますので、意識する必要はありません。

In [ ]:

```
import numpy as np

csv0bj1 = np.loadtxt("numpy_sample.csv", delimiter=",", dtype=int)
print(type(csv0bj1))
print("----------------------------------------")
print(csv0bj1)
```

- delimiterオプションは、入力ファイルがCSVファイルであることから、要素を,(カンマ)で区切ることを示しています。
- dtypeオプションは、読み込んだ内容を整数型として扱うことを示しています。

loadtxtでは、CSVファイルから読み込んだ結果をndarrayというデータ型の配列に格納します。このndarrayはNumPy独自のデータ型で、行列計算を高速に行うためのデータ型です。

行列計算の実行

ここでは配列の各要素を10倍した新たな配列を作成します。ndarray型のデータに対して四則演算の記述をするだけで、配列の全ての要素に対して演算を行うことができます。

In [ ]:

```
csv0bj2 = csv0bj1 * 10
print(csv0bj2)
```

CSVファイルの書き出し

新たな配列csvObj2を、CSVファイルに書き出してみます。NumPyでのファイルへの書き込みには、savetxt関数を使います。

In [ ]:

```
np.savetxt("numpy_output.csv", csv0bj2, fmt="%d")
```

- fmtオプションは、整数型として値を書き込むことを示しています。例えば、ここを"%f"とすると浮動小数点型となります。

Google Colaboratory画面左端のフォルダマークから、書き出された「numpy_output.csv」の中身を確認してみましょう。

データ構造化プログラムでは、NumPyを使うことは少ないかもしれません。

とはいえ例えば、メタデータとして出力したい値を、機器が出力した配列データから計算する必要があるケースなどがあれば、NumPyを使うことを検討してみてください。

pandasでのCSVファイル読み書き

pandasでCSVファイルの読み書きをやってみましょう。

モジュールのインポート

pandasを利用するには、今回の講義でご紹介したimport文を使って、pandasのモジュールをインポートします。

pandasをimportするときは、asを使って「pd」という別名を付けるのが慣例になっています。これもNumPyの「np」と同様、慣例に倣うようにしましょう。

```
import pandas as pd
```

【注意事項】

NumPyと同様に、Google ColaboratoryやAnacondaでは、pandasが最初からインストールされているのでimport文だけで利用できます。公式版Pythonでは最初はpandasがインストールされていませんので、プログラムを作成する前に、あらかじめpipコマンドを使ってインストールしてください。

```
pip install pandas
```

CSVファイルの読み込み

pandasでCSVファイルを読み込むには、「**read_csv関数**」を使います。

NumPyのloadtxt関数と同様に、pandasのread_csv関数でもファイルオープンのオープン・クローズを内部で自動的に行いますので、意識する必要はありません。

今回の例で読み込むのは、次のような内容のCSVファイルです。

measureID	date	operator	temperature	measureValue	measureUnit
MEA001	2022/11/1	鈴木	25	1000	sec
	2022/11/2	山田	20	999	sec
MEA002	2022/11/4	髙橋	R.T	98	min
MEA004	3-Nov-22	Adam	18	9	hour
MEA005			15	8	hour
MEA003	11-05-2022	山田		(欠測)	

In [ ]:

```
import pandas as pd

df = pd.read_csv("pandas_sample.csv", encoding="cp932")
print(type(df))
print("-------------------------------------------")
print(df)
```

read_csv関数にも、open関数のようにencodingオプションがあります。pandas_sample.csvはCP932のファイルなので、encoding="cp932"としています。

pandasでは、CSVファイルから読み込んだ結果をデータフレームという特別な配列に格納します。

元のCSVファイルとの違いの一つとして、データフレームの左端に通し番号0～6がついていますね。これは、pandasが自動的に付与したデータであり、「インデックス」と呼ばれます。

また、データフレームではデフォルトの動作として、1行目を自動的にヘッダとみなします。これにより、「measureValue列の値をすべて取り出す」といった指定での操作も可能になります。

データの加工

データ加工の簡単な例として、温度(temperature)が「R.T」となっているところを、25に置き換えましょう。

要素を置き換える方法は様々ありますが、ここでは「**loc**」を使用します。

次のように指定すると、データフレームの「"カラム名"列、"インデックス"行」の要素をポイントできますので、置き換えたい値を代入しています。

データフレーム.loc[インデックス, カラム名]

In [ ]:

```
df.loc[3, "temperature"] = 25
print(df)
```

"temperature"列の3行目はR.Tでしたが、25に置き換わりました。

CSVファイルの書き出し

それでは、書き換えた結果をファイルに書き込んでみます。

pandasによるCSVファイルへの書き込みには、to_csvメソッドを使います。

In [ ]:

```
df.to_csv("pandas_output.csv", index=False)
```

Google Colaboratory画面左端のフォルダマークから、書き出された「numpy_output.csv」の中身を確認してみましょう。

index=Falseを付けないと、データ左端にある0～6の番号もCSVファイルに出力されます。

これは元の入力ファイルにはなかったデータなので、今回はindex=Falseとして出力しませんでした。

【ワンポイント】

データ構造化プログラムにおいては、機器が出力したデータに欠測値があったり、「R.T」の例のように数値列の中に文字列が混じっていたりするなど、そのままでは構造化しづらいデータが頻繁に登場します。

そのようなデータをきれいに構造化するために、pandasは大活躍します。

pandasについては、第4回でもさらに詳しくご説明します。

ファイルパス

ここでは、ファイルパスについての基礎知識をご紹介します。

Windowsにおけるファイルパスの注意点

LinuxやmacOSとは異なり、Windowsではファイルパスの区切り文字が「\」になっています。

(Google Colaboratoryですと、通常の説明文では「バックスラッシュ」、コード例では「円記号」として表示されますが、実態は同じ文字です。)

OS	ファイルパスの例
Linux, macOS等	/data/tempfile.txt
Windows	C:\data\tempfile.txt

Windowsにおいて、例えばopen関数で次のようにファイルを開こうとすると、エラーとなります。

(Google ColaboratoryはLinux上でPythonが動作しているので、講義の場で実際に動かせないのが残念ですが・・・)

```
oepn("C:¥data¥tempfile.txt")
```

これは、Pythonにおいて、**ダブルクォートやシングルクォートで囲まれた文字列中の「\」記号が特別な意味を持つ**ためです。

上記では、2つ目の「\」記号の次に「t」が来ています(C:\data**\t**empfile.txt)。Pythonでは文字列中の「\t」は「タブ」の意味となります。

また、「ファイルの書き込み(writeメソッド)」でご紹介したように、ダブルクォートやシングルクォートで囲まれた文字列中に「\n」と記述すると、改行の意味になります。

それ以外にも、「\」記号が様々な意味を持つパターンがたくさんあります。

なお、これはあくまで**ダブルクォートやシングルクォートで囲まれた**文字列中の「\」記号の話です。次のように変数でファイルパスを指定していたとき、変数に格納された文字列に「\」記号があっても特別な意味にはなりません。

```
oepn(filepath)
  → filepathの中身が文字列「C:¥data¥tempfile.txt」であってもエラーにはならない
```

以上のことから、PythonでWindowsのファイルパスを文字列として表すには、工夫が必要です。いくつかの方法をご紹介します。

「\」記号をエスケープする

文字列中に「\\」と記述すると、特別な意味を持たない単なる「\」という文字になります。

このように、特別な意味を持つ記号の前に「\」を付け足して、特別な意味を持たなくすることを、プログラミングの世界では「エスケープする」と言います。

したがって、次のとおり記述することで、前述のプログラムはエラーとはならなくなります。

```
oepn("C:¥¥data¥¥tempfile.txt")
```

ちなみに「\d」は特別な意味を持たないパターンなので、最初の「\」はエスケープしなくても問題ありません。とはいえ、どれが特別な意味を持つのかな？とあれこれ考えるよりも、すべての区切り文字をエスケープした方が安全でしょう。

少し手間はかかりますが、確実な方法です。

raw文字列を使う

文字列の開始となる"(ダブルクォート)や'(シングルクォート)の前に「r」と記述すると、raw文字列というものになり、文字列中の「\」が**ほぼ**特別な意味を持たなくなります。次のとおり記述することで、前述のプログラムはエラーとはならなくなります。

```
oepn(r"C:¥data¥tempfile.txt")
```

前述の「\」記号をエスケープする方法よりも手間がかかりませんが、次の点に注意が必要です。

- 特別な意味を持つ「\」も同時に使いたい場合は、この方法が使えません。
- 文字列の最後が「\」で終わる場合、その「\」だけは「\\」と記述してエスケープしなければいけません。raw文字列であっても「\"」「\'」だけは「\」が特別な意味を持ってしまうからです。(文字列の終わりを示す「"」「'」ではなく、単なる「"」「'」という文字になってしまう。)

```
open(r"C:¥data¥tempfile¥")  # 文字列終了を示す「"」がないとみなされてエラー
open(r"C:¥data¥tempfile¥¥") # このようにする必要がある
```

「\」の代わりに「/」を使う

実は、open関数を含むPythonの標準関数や、広く使われるほとんどの外部モジュール・パッケージでは、OSがWindowsであっても、LinuxやmacOSと同じようにファイルパスの区切り文字を「/」にしても正しく動作します。

```
oepn("C:/data/tempfile.txt")
```

お手軽でよい方法ですが、やはり注意すべきポイントがあります。

例えば次のソースコードでは、exmoduleモジュールのfunc関数に、引数としてファイルパスを渡すとします。

このとき、func関数の中で区切り文字が「\」であることを前提とした処理があると、正しく動きません。

```
import exmodule
exmodule.func("C:/data/tempfile.txt")
```

pathlibモジュールと組み合わせる

少し高度なやり方として、「**pathlibモジュール**」と組み合わせる方法があります。

pathlibモジュールは、ファイルパスに関わる様々な機能を提供する、Pythonの標準モジュールです。

pathlibモジュールを厳密に理解するには「クラス」の知識が必要となりますが、ここではイメージをつかんでいただく程度で簡単にご紹介します。

```
from pathlib import Path

filepath = Path("C:/data/tempfile.txt")
```

pathlibモジュールが提供するPathクラスというものを使って、ファイルパスオブジェクトを作成します。上記の例では、変数filepathにファイルパスオブジェクトが格納されます。

ファイルパスオブジェクトは文字列型ではありませんが、str関数を使うと、ファイルパスの文字列になります。

```
print(str(filepath))
```

ファイルパスオブジェクトは、作成時に区切り文字として「/」を使っていても、OSがWindowsであれば区切り文字を自動的に「\」に変換して保持してくれます。

そのため、上記のprint関数の出力結果は、区切り文字を「\」として「C:\data\tempfile.txt」となります。

前述のexmodule.func関数の例は、次の通り記述すれば、関数に「C:\data\tempfile.txt」が渡されるため、問題が起こりません。

```
from pathlib import Path
import exmodule

filepath = Path("C:/data/tempfile.txt")
exmodule.func(str(filepath)) # 区切り文字「¥」で関数に渡される
```

結論として、もっとも確実な方法は「\」記号をエスケープするか、区切り文字を「/」としてpathlibと組み合わせることです。

ただし、注意点として挙げた内容が問題なければ、他の方法を使ってもよいでしょう。

ファイルパスの操作

データ構造化プログラムでは、例えば入力ファイル名を加工して、新たな拡張子を付けて出力ファイル名にするなど、ファイルパス文字列に対して様々な操作を行います。

ここでは、ファイルパスを加工するいくつかの機能をご紹介します。

ファイルパスからファイル名を取り出す

Pythonが提供するos.pathモジュールの「**basename関数**」を使用すると、ファイルパスからファイル名を取り出すことができます。

os.pathモジュールを利用する際は、osモジュールをインポートして、ソースコード中で「os.path.関数名」「os.path.変数名」といった形で参照します。

In [ ]:

```
import os

print(os.path.basename("/root/data/file.txt"))
```

ファイルパスからディレクトリ部分を取り出す

os.pathモジュールの「**dirname関数**」を使用すると、ファイルパスからディレクトリ部分を取り出すことができます。

In [ ]:

```
import os

print(os.path.dirname("/root/data/file.txt"))
```

ディレクトリ名やファイル名を連結してファイルパスを生成する

os.pathモジュールの「**join関数**」を使用すると、ディレクトリ名やファイル名を連結してファイルパスを生成することができます。

このとき、連結部分の区切り文字は、OSがWindowsであれば「\」に、LinuxやmacOSであれば「/」に、自動で設定してくれます。

In [ ]:

```
import os

print(os.path.join("root", "data", "file.txt"))
```

上記をWindows上で実行したときは、結果は「root\data\file.txt」となります。

練習問題

basename関数、join関数の2つを使って、「input/spectrum_001.jdx」という入力ファイルのパスから、「output/spectrum_001_graph.png」という出力ファイル名を生成してみましょう。

(実際のところはいくらでも方法はあるのですが、今回は上記2つの関数を使ってください！)

ヒント：splitメソッドも使用します。

In [ ]:

```
import os

input_path = "input/spectrum_001.jdx"

# input_pathを加工して「output/spectrum_001_graph.png」という文字列を作成し、変数output_pathに格納し

 print(output_path)
```

pathlibの利用

-
「Windowsにおけるファイルパスの注意点」でご紹介したpathlibモジュールを使って、ここまでご紹介したようなファイルパス操作のいくつかを行うこともできます。

また、ファイルやディレクトリの作成、移動、削除などといった操作を行うこともできます。(標準的なやり方では、これらの操作にはshutilモジュールなどを使用します。)

ご参考として、pathlibモジュールが提供するファイルパス操作のための関数を、いくつかピックアップして一覧でご紹介します。

関数	機能
name	ファイルパスからファイル名を取り出す
stem	nameに加えて拡張子を取り除く(pathlib固有の機能)
parent	ファイルパスからディレクトリ部分を取り出す
parents	ファイルパスから、ディレクトリ部分を分解した結果を取り出す(pathlib固有の機能)
/演算子、joinpath	ディレクトリ名やファイル名を連結してファイルパスを生成する
mkdir	ディレクトリを作成する(再帰的な作成も可能)
rename	移動する、名前を変更する
unlink	削除する

用途ごとにosモジュール、shutilモジュールと別々のモジュールを使い分けるよりも、すべてpathlibにまとめたほうがプログラムを作成しやすいですし、pathlibを使うことをチーム内でルール化すれば、他の人が作ったソースコードを読みやすくなるという利点もあります。

pathlibは非常に多機能であること、「クラス」についての知識が必要であることから、今回は詳しくはご説明しませんが、Pythonの知識がついてきたらぜひ調べて使ってみてください。

ARIM事業のデータ構造化プログラムでも、pathlibを使ってファイルパス操作を行っています。

第3回の講義は、ここまでとなります。

第4回では、今回ご紹介したpandasについてのさらなる詳細など、データ構造化プログラムにおけるの代表的なデータ操作方法についてご紹介します。