RDE構造化処理プログラム開発 手順書

～ **RDEToolKit(invoice**モード**)**を利用したシンプルな**RDE**データ構造化処理プログラムハンズオン ～

リリース**2.7.1 (20250924)**

国立研究開発法人 物質・材料研究機構

目次

1. はじめに 4
    1.1 RDEとは 4
    1.2 目的および対象読者 6
    1.3 開発の流れ 7
    1.4 結局のところ、RDE構造化処理プログラムはなにをすればいいんですか? 10
2. 環境準備 11
    2.1 Linux環境 11
    2.2 Python環境 12
    2.3 Python仮想環境作成 13
    2.4 Python仮想環境の有効化 13
    2.5 RDEToolKitのインストール 13
    2.6 ディレクトリ作成 15
    2.7 初期化 15
    2.8 禁止事項 18
    2.9 本書でのディレクトリ表記 19
    2.10 OS提供コマンドのインストールについて 19
3. サンプルデータの配置 20
    3.1 sampla.data 20
    3.2 invoice.jsonファイルの更新 21
    3.3 invoice.schema.jsonの生成 22
    3.4 metadata-def.jsonの変更 23
    3.5 samples.zipの利用 27
4. 構造化処理の記述開始 28
    4.1 main.py 28
    4.2 datasets_proces.py 29
5. 例外処理(エラー処理)とフォルダ初期化 35
    5.1 例外処理(エラー処理) 35
    5.2 フォルダ初期化 36
6. 入力ファイルのチェック 38
    6.1 想定 38
    6.2 custom_module()にチェックロジックを実装 38
    6.3 確認 39
7. invoice.jsonの読み込みと保存 42
    7.1 想定 42
    7.2 custom_module()に読み込みロジックを実装 43

8. 入力データの読み込み 46
8.1 想定 46
8.2 custom_module()に読み込みロジックを実装 46
9. メタデータの出力 49
9.1 想定 49
9.2 custom_module()にロジックを実装 50
9.3 (参考) metadata.json 51
9.4 おまけ 52
10. 実験データの可読化 54
10.1 想定 54
10.2 custom_module()にロジックを実装 54
10.3 (参考)出力結果の確認 54
11. 実験データの可視化 56
11.1 想定 56
11.2 custom_module()にロジックを実装 56
11.3 (参考)結果の確認 57
12. サムネイル画像の作成 58
12.1 想定 58
12.2 custom_module()にロジックを実装 58
12.3 (参考)画像データをそのままサムネイル画像として使う 59
13. 実験データの永続化 61
13.1 想定 61
13.2 custom_module()にロジックを実装 62
14. 応用編 64
14.1 準備 64
14.2 実装 70
15. 応用編2 87
15.1 準備 87
15.2 実装 89
16. 変更履歴 101

1. はじめに

ようこそ、"RDE構造化プログラム開発"の世界へ!

本書は、RDE構造化プログラムの作成を実際にプログラムを作成しながら習得出来るようにするものです。

> 本書は、RDEToolKit(後述)の "バージョン1.3.4" に対応しています。

1.1 RDEとは

RDEは、物質・材料についての研究データをオンラインで迅速に登録・活用するために国立研究開発法人 物質・材料研究機構(以下NIMS)が開発したシステムです。

装置からの出力ファイル(以下"生データ"または"生データファイル"と呼ぶ)をRDEのサイトにてアップロード、つまり"データ登録"を実行すると、自動的にデータ駆動型のマテリアル研究に適した形に構造化してクラウドに蓄積します。これによりユーザや研究グループ内での再利用や他の研究グループとのデータの共用が容易となり、マテリアル研究開発のDX化を支援します。

多くの場合、RDEへのデータ登録とは「データの永続化」、「データの構造化」、「計算処理」の3つの処理を実現するものです。この「データの永続化」、「データの構造化」、「計算処理」を実行するプログラムを「構造化プログラム」と呼び、上記3 種類の機能の一部または全てを実装します。

例えば、「データ永続化」のみを実施し、「データの構造化」や「計算処理」を全く処理しない「構造化プログラム」を開発することも可能です。

本書では、上記一連の処理を合わせて「RDE構造化処理」と呼ぶことにします。

> 本書の中で「構造化」という言葉は、メタデータ作成、CSVファイルの作成といった狭義の意味で使う場合と、RDE構造化処理全体の一連の処理といった広義の場合の2種類が混在しています。ご注意ください。

1.1.1 データの永続化

実験により得た測定(RAW)ファイルなどを、RDEに登録する事で、削除や上書きされることなく保存しておくことができます。

RDE では登録されたファイルを"永続化ファイル"と呼びます。(決められた期間の後)保存された生データファイルを"非公開"とするか"公開"とするかは「構造化プログラム」の中で決定し処理が実行されます。

- RDEでは、保存するデータファイルとして実験データだけでなく、シミュレーション等の計算結果やパラメータファイルなども扱うこともできます。本書では、ここ以降「実験装置からの出力ファイル」に限定して進めることとします。
- 永続化ファイルは、RDE構造化処理の中で永続化対象フォルダに格納することで永続化ファイルとなります。ディレクトリ構成は後述します。

1.1.2 データの構造化

RDEでは、登録するデータに

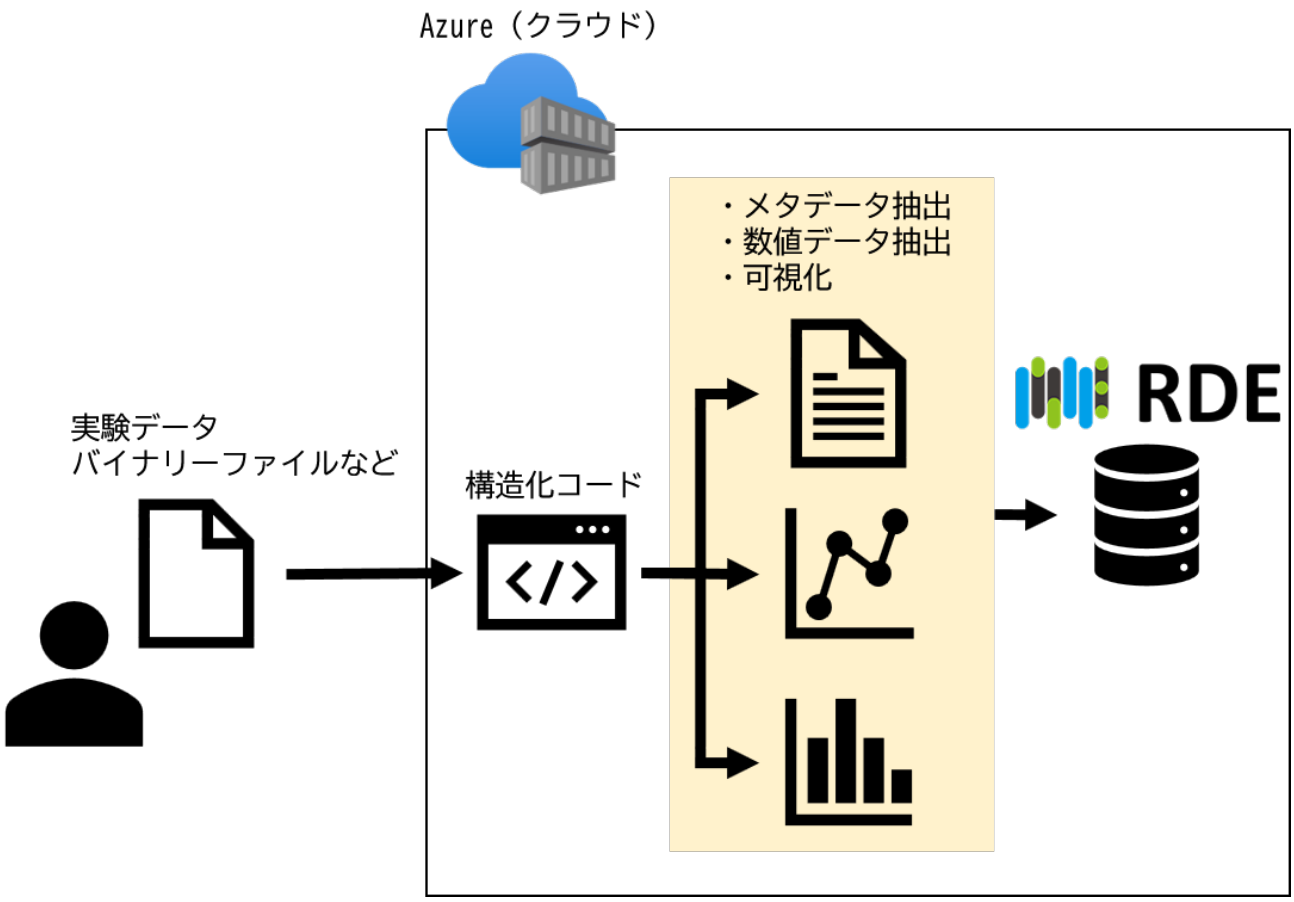
- 実験した日付
- 実験対象の試料種別
- 実験時の温度や圧力などの条件、特徴量

といった追加情報を付与することができます。このような情報を「メタデータ」と呼びます。

RDE における"データの構造化"は、以下のような処理が含まれます。

- 生データからメタデータを抽出し、JSON形式で保存する。
- 生データに含まれる計測値を、人が容易に読解できるような数値データとしたデータファイル(CSV形式のファイルなど)を生成する。
- 生データに含まれる計測値をもとに、png形式やjpg形式のグラフ画像を生成する。

どのような項目をメタデータとして格納するのかは、構造化プログラム開発前に決定する必要があります。それぞれのメタデータに格納する値を、どのように取得するかは開発者が実装します。



1.1.3 計算処理

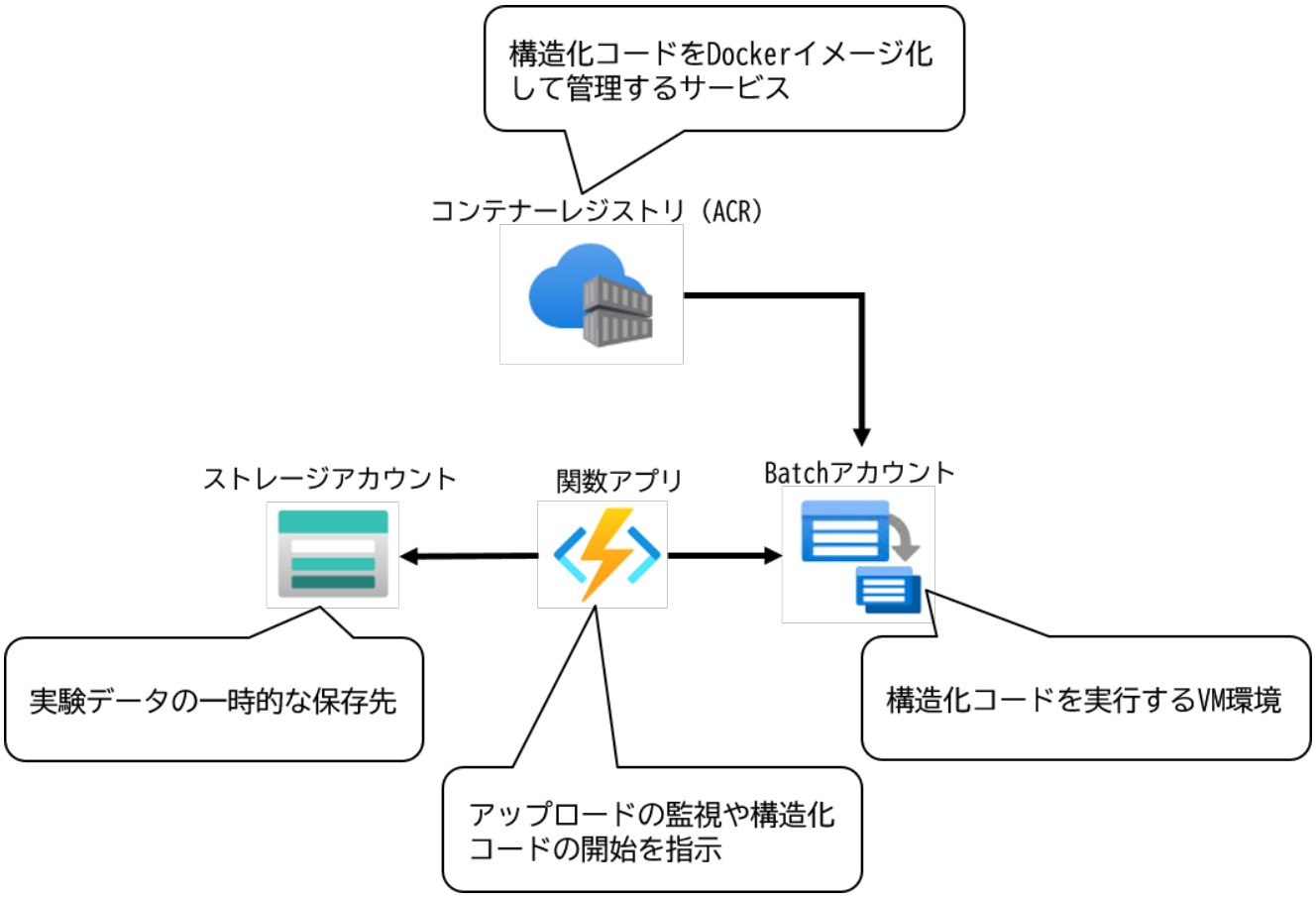
RDEにおける計算処理とは、「実験データなどを入力元として、何らかの計算等でデータを導出する処理」を指します。

例えば以下のようなものです。

- 実験データをGauss 分布等の数値モデルにフィッティングした結果のピーク位置を導出する。
- 実験データを統計的に集計した平均や標準偏差を導出する。

1.1.4 RDEのシステム構成

RDEはクラウド上(MS Azure上)で動作します。データ構造化に関係するAzure のサービス連携図を示します。



構造化プログラムは最終的にDocker イメージ化されて、ACR(Azure Container Registry)に登録して利用します。

1.2 目的および対象読者

本書は、RDEにデータを投入するための構造化処理の開発に携わる方々(以下、"開発者"と呼ぶ)に向けて書かれています。

開発者は、NIMS職員のほか、NIMSまたは他の組織から委託された外部業者の場合もあり得ます。そのため本書では、開発者がいずれの立場であったとしても、問題無く構造化処理についてのプログラム作成が可能となるために必要な知識を提供することを目的とします。

また、本書では、RDE構造化処理を、Python言語で記述します。そのため開発者はPython言語についての一般的な知識を有していることが前提となります。

1.3 開発の流れ

RDEでの構造化処理の開発は、概ね以下の様になります。

1. ローカル開発
2. Dockerイメージ作成&確認
3. 上記Dockerイメージを基に、データセットテンプレート(後述)の作成
4. 作成されたテンプレートを基に、データセット(後述)の作成

本書では、これらのうち「ローカル開発」について記述します。

なお、ローカル開発にDockerコンテナを利用して行う方法もありますが、本書ではDockerコンテナを使用しない方式を想定します。

1.3.1 ローカル開発とは?

RDEで構造化処理が行われるのは、クラウド上のLinux環境上です。ローカル開発とは、自前のLinux環境を用意し、その環境上で、想定されるデータがアップロードされた状態から、RDE構造化処理を実行し、想定されるファイル出力が行われるようにPythonスクリプト(あるいは他の開発言語)を生成することを指します。

構造化プログラムの開発言語は、Docker イメージを作成できれば任意のものに対応可能ですが、推奨している言語はPythonとなります。

本書はPython3.12 を使用して、シンプルな構造化プログラムをローカル環境で実装、動作させます。

1.3.2 データセットテンプレートとは

RDEにおいて、データセットテンプレート(あるいは単にテンプレートと呼ぶ場合もあります)とは、RDE構造化処理に必要な一連のスクリプトを含むDockerイメージのことを指します。

1つのデータセットは、必ず1つのテンプレートが指定され構成されます。

・反対に複数のデータセットが、同じテンプレートを共用する場合もあります。

1.3.3 データセットとは

登録画面の形式(項目や、その項目の形式、それぞれが必須かどうかの設定など)やどういった構造化処理を実施するかは、テンプレートにより確定します。

データセットは、1つのテンプレートを使って登録する一連のデータの集まり、または"器"です。データを登録するRDEユーザは、テンプレートではなく、データセットを選択して登録、つまりファイルをアップロードします。

テンプレートは、データセット開設時に決定し、以後変更されることはありません。つまり、同じデータセットに格納されたデータは、同じ構造化処理を実行したものであることが保証されることになります。

また、あるデータセットに対して想定されるデータとは異なる形式のデータを登録した場合はエラーとなって登録できないことになります。

1.3.4 データタイルとは

前述のように、1つのデータセットには、同じ形式のデータが複数登録されます。

それぞれのデータ(ファイル、または複数のファイル群)を"データタイル"と呼びます。

- 1回のRDE構造化処理にて出来るのが"1つのデータタイル"とは限りません。1回のRDE構造化処理で複数のデータタイルを登録することが(→そのようにテンプレートを構成することが)可能です。

1.3.5 インボイス(Invoice/送り状)とは

RDEでは、データを登録する際に、登録されるデータがどういう条件で実験や計測を行ったか、といった項目を、実験結果のファイルとは別に記述することができます。

この項目を記述したファイルをInvoice(またはInvoiceファイル)と呼びます。さらに別称として"送り状"と呼ぶ場合もあります。

Invoiceファイルは、画面入力をそのままJSON形式のファイルにして利用する場合の他、複数のInvoice情報をまとめて記述できるExcel形式ファイルの場合など複数のやり方があります。

> RDEではInvoiceの情報項目もメタデータとして利用されます。手入力のメタデータがある場合はInvoiceに定義します。

1.3.6 Pythonのバージョン、外部ライブラリのバージョンの考え方

多くの場合、RDE構造化処理は、Python言語により開発されることが想定されます。

後述するRDEToolKitのバージョンに応じたPythonのバージョンをお使いください。

現時点(2025年09月)では、以下のバージョンをご利用ください。

• Python 3.12.x

> 特に問題がない場合は、Python3.12の中の最新版をお使いください。

RDEToolKitが利用する外部ライブラリは、RDEToolKitに付随するrequirements.txtに記述されていますので、RDEToolKitインストール時に、指定されたライブラリの、指定されたバージョンが同時にインストールされます。

その他必要な外部ライブラリの利用は任意ですが、それぞれのライブラリを `バージョンを指定` しての利用を考えてください。

> これは、例えば"1.1.3以上"と指定するのではなく"1.1.3"と固定で指定することを意味します。

1.3.7 RDEToolKitとは

RDE構造化処理をサポートするツール群や、RDE構造化処理のワークフローを作成するツール群をまとめたもので、Pythonのpipモジュールとして提供されています。

RDEToolKitを使うことで、RDE構造化処理により出力されるファイルを格納するフォルダ構成の自動生成や、metadata-def.jsonファイルを用いたメタデータJSONファイルの簡易生成機能の提供など、開発者が構造化処理を構築する際の作業負荷軽減が可能となります。

RDEToolKitでは、以下の4つの処理モードを想定しています。

#	処理モード	説明	備考
1	Invoiceモード	1つのInvoiceファイルを用い、1つのデータタイルを対象とした処理モード	※本書はこれを想定
2	ExcelInvoiceモード	Invoiceファイルの代わりにExcel形式でInvoice情報を与える処理モード	
3	マルチデータタイルモード	複数のファイルを、Invoiceなしで登録する処理モード	
4	RDEフォーマットモード	構造化処理を予め済ました状態で登録する処理モード	

> これら処理モードの名称は暫定的なものです。近い将来変更される可能性がありますので注意してください。

今後のRDE構造化処理の開発は、RDEToolKitを使用することが強く推奨されます。

1.4 結局のところ、RDE構造化処理プログラムはなにをすればいいんですか?

いままでの説明で、なんとなくはRDE構造化処理プログラムのイメージはつかんでいただけたかと思います。

ただ、こうも思っているかもしれません。

```
で、結局のところ、RDE構造化処理プログラムは何をすればいいのか?
```

具体的な処理については、後続の章に譲りますが、まとめると以下の様になります。

- 起動されたら、以下に示すような処理を実行する(通常は `python main.py` にて起動する)。
- 起動スクリプト(以下、"main.py"として記述します。)と同じ階層にある"data/inputdata"フォルダから入力ファイルを読み込む。zip圧縮されている場合は、展開してから読み込む。
- main.pyと同じ階層にある"data/invoice"フォルダにある"invoice.json"を読み込む。必要なら改変し、永続化フォルダ("data/nonshred_raw"または"data/raw"のいずれか)にコピーする。
- 入力ファイルを、永続化フォルダにコピーする。
- メタデータを収集し、main.pyと同じ階層にある"data/metadata"フォルダに、metadata.jsonとして出力する。
- 可読化など何らかの"解析処理"を実施し、main.pyと同じ階層にある"data/structured"フォルダに出力する。ファイル名、ファイル形式は任意。
- 入力ファイルに画像が含まれる場合はその画像を、main.pyと同じ階層にある"data/main_image"または"data/other_image"フォルダにコピーする。または可視化処理に伴いグラフ画像などの画像データが作成される場合は、main.pyと同じ階層にある"data/main_image"または"data/other_image"フォルダに出力する。ファイル名は任意ですが、現状ファイル形式には制限があります(GIF形式、JPEG形式またはPNG形式)。
    - SVG形式など、その他の画像形式には対応していません。格納することはできますが、プレビューが表示されない、ダウンロード出来なくなるなどの不具合が発生しますので注意してください。

これらの処理の一部、または全部を実行するようにスクリプトを作成すればよいことになります。

2. 環境準備

ローカル開発環境として求めれるのは、以下のようなものです。

項目	内容	備考
開発マシン	実機、仮想マシンいずれも可	
開発OS	Linux	ディストリビューションは問わない
開発言語	Python v3.10以上	仮想環境(venv)が利用できること
RDEToolKit	v1.3.4	開発時点の最新版の利用を推奨
Docker	Docker Engine、Docker CLI	
開発補助ツール	テンプレート生成ツール、構造化処理結果プレビューツール	NIMS提供のツール。テンプレート生成ツールの利用にはMS Excelが必要

なおRDE実行環境では、主に `4vcpu、16GBメモリ、総ディスク容量10GB` の環境を利用して構造化処理プログラムが実行されますので、それを参考に実機、仮想マシンを用意してください。

> 上記を超えるような性能を必要とする場合は、開発前にご相談ください。

またRDEでは、内部でDocker機能を使用していますが、登録するコンテナはNIMSにてbuildするため、ローカル開発ではDocker機能の準備は必須ではありません。ただし、開発者は、Dockerイメージを作成するために使用する `Dockerfile` の作成までを求められますので、そのためにDocker Engine、Docker CLIの準備が必要となります。本書ではDockerを使った開発に関しては扱いません。別途提供している『RDEToolKitを用いたRDE構造化処理用Dockerコンテナ作成手順書』を参照ください。

> 『RDEToolKitを用いたRDE構造化処理用Dockerコンテナ作成手順書』は以下のURLにて公開されています。
>
> https://github.com/nims-mdpf/RDE_Docs_firsttouch

チュートリアルを実施していくにあたって、以下の環境を準備します。

項目	内容	備考
開発OS	Ubuntu 24.04	
開発言語	Python v3.12	仮想環境(venv)を含む
パッケージ	RDEToolKit v1.3.4	RDEToolKitの依存パッケージを含む

2.1 Linux環境

ローカル、つまり手元で利用できるLinux環境を用意します。

Pythonが稼働すればよいので、ディストリビューションに特に制限はありません。

> Pythonを実行するディストリビューションとして、現時点(2025年09月)では `Ubuntu 24.04` の利用を推奨します。以降本書では、Ubuntu 24.04を使った場合の実行例を示します。また、バージョン番号を明記しない `Ubuntu` は、`Ubuntu 24.04` を指すものとします。

また、実マシン上に構築されたOSでもいいですし、VirtualBoxなどを利用したVM(Virtual Machine)でも問題ありません。Windowsを利用している場合は、WSL2の利用を推奨します。

> 以下本書では、Linux環境のロケールは日本語であることを想定しています。英語環境などその他の場合は表示が異なりますので、適宜読み替えるか、または日本語ロケールに変更してください。

以下に、日本語ロケールに変更する場合のコマンド例を示します。

```
$ localectl
System Locale: LANG=en_US.UTF-8
    VC Keymap: (unset)
   X11 Layout: jp
    X11 Model: pc105

$ sudo apt update
$ sudo apt install language-pack-ja

$ sudo localectl set-locale ja_JP.UTF-8

$ localectl
System Locale: LANG=ja_JP.UTF-8
    VC Keymap: (unset)
   X11 Layout: jp
    X11 Model: pc105
```

> 日本語パックを導入し、デフォルトロケールを変更します。

2.2 Python環境

python処理系がインストールされていて、利用可能である必要があります。

```
$ python3 --version
Python 3.12.3
```

また仮想環境も利用しますので、インストールされていることを確認します。Ubuntuの場合、以下の様に確認できます。

```
$ sudo apt list python3-venv
一覧表示... 完了
python3-venv/noble-updates,now 3.12.3-0ubuntu2 amd64 [インストール済み]
N: 追加バージョンが 1 件あります。表示するには '-a' スイッチを付けてください。
```

インストールされていない場合は、インストールします。

```
sudo apt install python3-venv
```

2.3 Python仮想環境作成

> ここでの"Python仮想環境"は、pipモジュールを他の環境とは隔離して利用することを主目的として利用します。そのため、"Python仮想環境"の利用は必須ではありません。

Python仮想環境として、`tenv`という名前で環境を作成します。

```
cd $HOME
python3 -m venv tenv
```

> `-m venv`でモジュールの読み込みを、最後の`tenv`で仮想環境の名前を、それぞれ指定しています。後者は任意に命名して構いません。すでに仮想環境として`tenv`が存在している場合は、適宜変更してください。本書では、Python仮想環境として`tenv`を利用しているものとして記述します。

2.4 Python仮想環境の有効化

```
$ source tenv/bin/activate
(tenv) $
```

上記の例では、"tenv"ディレクトリは、ホームディレクトリに作成されていますので、以下の様にしても同じ結果となります。

```
$ source $HOME/tenv/bin/activate
(tenv) $
```

> sourceコマンド実行の結果、プロンプトが変わります(→ 元のプロンプトの前に、Python仮想環境名が括弧付きで表示されます)ので、当該仮想環境が有効になったことが分かります。

python仮想環境を使った場合、`python3`コマンドだけでなく、`python`コマンドも利用可能になります。

```
(tenv) $ python3 --version
Python 3.12.3

(tenv) $ python --version
Python 3.12.3
```

> どちらも同じものです。本書では`python3`ではなく、`python`コマンドを用いることにします。

2.5 RDEToolKitのインストール

この時点では、pipだけがインストールされた状態になっているはずです。

```
(tenv) $ pip list
Package Version
------- -------
pip     24.0
```

> インストールしたPythonバージョンによっては、"setuptools"が表示されることがあります。特に問題はありませんのでそのまま先に進みます。

pip自身を最新にします。

```
pip install pip -U
```

> Proxyの設定やSSLエラー対応などの設定がすでに済んでいるものとします。

RDEToolKitをインストールします。

```
pip install rdetoolkit
```

> 上記コマンドにて、必要な依存パッケージも同時にインストールされます。

インストールが正常に終わった場合、pipモジュールは以下の様になります。

```
Package                   Version
------------------------- -----------------
annotated-types           0.7.0
attrs                     25.3.0
build                     1.3.0
chardet                   5.2.0
charset-normalizer        3.4.3
click                     8.2.1
et_xmlfile                2.0.0
eval_type_backport        0.2.2
jsonschema                4.25.1
jsonschema-specifications 2025.4.1
Markdown                  3.8.2
numpy                     2.3.2
openpyxl                  3.1.5
packaging                 25.0
pandas                    2.3.2
pip                       25.2
polars                    1.32.3
pyarrow                   21.0.0
pydantic                  2.11.7
pydantic_core             2.33.2
pyproject_hooks           1.2.0
python-dateutil           2.9.0.post0
pytz                      2025.2
PyYAML                    6.0.2
rdetoolkit                1.3.4
referencing               0.36.2
rpds-py                   0.27.0
six                       1.17.0
toml                      0.10.2
tomlkit                   0.13.3
types-pytz                2025.2.0.20250809
typing_extensions         4.15.0
typing-inspection         0.4.1
tzdata                    2025.2
```

> 他のバージョンのRDEToolKitの場合は、依存パッケージのバージョンも変わっている可能性があります。
> RDEToolKitでは、バージョン毎に必要なpipモジュールのバージョンも固定していますので注意してください。また、インストール時期により依存パッケージバージョンが変わるものもありますので、同じバージョンのRDEToolKitを利用していた場合でも、必ずしも上記と同じバージョンになるとは限りません。
>
> 他のプロダクトとの関係で、特定のpipモジュールバージョンだけ変更する必要がある場合もあり得ます。その場合はそちらに合わせてみて、構造化処理プログラム実行に問題がないかを確認する必要があります。

2.6 ディレクトリ作成

次に `作業用のディレクトリ` を作成します。

> 開発の後半フェーズでは、Dockerイメージを作りますが、そのDockerイメージにはこのディレクトリ下にあるフォルダ/ファイルだけを格納します。従って、このディレクトリ名は任意です。

ユーザのホームディレクトリに `handson` というディレクトリを作成し、さらにその下に `tutorial` というディレクトリを作成します。

```
mkdir $HOME/handson
mkdir $HOME/handson/tutorial
```

`tutorial` ディレクトリに移動します。

```
cd $HOME/handson/tutorial
```

2.7 初期化

RDEToolKitを用いて、RDE用のディレクトリ構成を作成します。

最初に、現在のディレクトリが、ホームディレクトリ(ここでは/home/devel)の下の `handson/tutorial` であることを確認します。

```
(tenv) $ pwd
/home/devel/handson/tutorial
```

想定通りであることが確認できたので、"初期化"処理を実施します。"初期化"処理をすることで、構造化処理に必要な最小限のディレクトリ、ファイルが作成されます。

```
(tenv) $ python -m rdetoolkit init
Ready to develop a structured program for RDE.
Created: /home/devel/handson/tutorial/container/requirements.txt
Created: /home/devel/handson/tutorial/container/Dockerfile
Created: /home/devel/handson/tutorial/container/data/invoice/invoice.json
Created: /home/devel/handson/tutorial/container/data/tasksupport/invoice.schema.json
Created: /home/devel/handson/tutorial/container/data/tasksupport/metadata-def.json
Created: /home/devel/handson/tutorial/templates/tasksupport/invoice.schema.json
Created: /home/devel/handson/tutorial/templates/tasksupport/metadata-def.json
Created: /home/devel/handson/tutorial/input/invoice/invoice.json

Check the folder: /home/devel/handson/tutorial
Done!
```

このバージョンのRDEToolKitでは、以下のディレクトリ/ファイルが作成されます。

```
(tenv) $ tree -F
./
├── container/
│   ├── Dockerfile
│   ├── data/
│   │   ├── inputdata/
│   │   ├── invoice/
```

```
│   │   │   └── invoice.json
│   │   └── tasksupport/
│   │       ├── invoice.schema.json
│   │       └── metadata-def.json
│   ├── main.py
│   ├── modules/
│   └── requirements.txt
├── input/
│   ├── inputdata/
│   └── invoice/
│       └── invoice.json
└── templates/
    └── tasksupport/
        ├── invoice.schema.json
        └── metadata-def.json

12 directories, 9 files
```

> ダミーデータを使ったファイルが設定され、この時点で、最小限実行可能な状態となっています。後述するようにこれらのファイルを置き換えていく必要があります。
>
> また、input/ ディレクトリにあるファイルや templates/ にあるファイルは、実際の構造化処理においては利用されません。構造化処理で利用されるのは、container/ ディレクトリ下にある data/ ディレクトリ以下のファイルとなります。

初期化直後の、各ファイルは以下の様になっています。

```
(tenv) $ cat container/data/invoice/invoice.json
{
    "datasetId": "3f976089-7b0b-4c66-a035-f48773b018e6",
    "basic": {
        "dateSubmitted": "",
        "dataOwnerId": "7e4792d1a8440bcfa08925d35e9d92b234a963449f03df441234569e",
        "dataName": "toy dataset",
        "instrumentId": null,
        "experimentId": null,
        "description": null
    },
    "custom": {
        "toy_data1": "2023-01-01",
        "toy_data2": 1.0
    },
    "sample": {
        "sampleId": "",
        "names": [
            "<Please enter a sample name>"
        ],
        "composition": null,
        "referenceUrl": null,
        "description": null,
        "generalAttributes": [],
        "specificAttributes": [],
        "ownerId": "1234567e4792d1a8440bcfa08925d35e9d92b234a963449f03df449e"
    }
}(tenv) $
```

```
(tenv) $ cat container/data/tasksupport/invoice.schema.json
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://rde.nims.go.jp/rde/dataset-templates/dataset_template_custom_sample/invoice.schema.json",
    "description": "RDEデータセットテンプレートテスト用ファイル",
    "type": "object",
    "required": [
        "custom",
        "sample"
```

```
    ],
    "properties": {
        "custom": {
            "type": "object",
            "label": {
                "ja": "固有情報",
                "en": "Custom Information"
            },
            "required": [
                "toy_data1"
            ],
            "properties": {
                "toy_data1": {
                    "label": {
                        "ja": "トイデータ１",
                        "en": "toy_data1"
                    },
                    "type": "string"
                },
                "sample2": {
                    "label": {
                        "ja": "トイデータ２",
                        "en": "toy_data2"
                    },
                    "type": "number"
                }
            }
        },
        "sample": {
            "type": "object",
            "label": {
                "ja": "試料情報",
                "en": "Sample Information"
            },
            "properties": {}
        }
    }
}(tenv) $
```

> RDEToolKit v1.3.2より前のバージョンでは、ローカル開発時は問題ないものの、そのままRDEシステム上で利用すると問題が発生するものが生成されていました。最初の4行が"\$schema"、"\$id"、"description" および "type"となっていることを確認してください。

```
(tenv) $ cat container/data/tasksupport/metadata-def.json
{}(tenv) $
```

> RDEToolKitのバージョンにより内容が変更になる可能性があることに注意してください。

この時点で、RDEToolKitが導入されたことを確認します。

```
(tenv) $ cd container
(tenv) $ ls
Dockerfile  data  main.py  modules  requirements.txt
(tenv) $ python -m rdetoolkit version
1.3.4
```

・上は本書執筆時点でのバージョンとなります。随時アップデートされます。

次に、なにも変更せずに、"構造化処理"を実行してみます。

```
(tenv) $ python main.py
(tenv) $
```

RDEToolKitの過去のバージョンでは、構造化処理プログラム実行時に プログレスバー(進捗表示) が表示されるものがありましたが、現在はプログレスバーは表示されません。

また、RDEToolKitのバージョンによっては、上記実行時に"FileNotFoundError: The schema and path do not exist: metadata.json"といったエラーが表示されることがあります。

その場合は、下記のようにダミーのmetadata.jsonを作成してください。

> 最新のRDEToolKitではこのエラーは発生しないため、この処理は不要です。

```
(tenv) $ echo -e '{\n  "constant": {},\n  "variable": []\n}' > data/meta/metadata.json
```

`vi` などのエディタを利用して `metadata.json` を作成しても構いません。

```
{
  "constant": {},
  "variable": []
}
```

実行し、エラーが表示されないことを確認します。

```
(tenv) $ python main.py
(tenv) $
```

2.8 禁止事項

以下のようなものは、RDE環境で利用することが出来ませんので、ローカル開発でも使用しないようにしてください。

- 公開利用が禁止されているもの
- 有償の開発環境、パッケージ、コマンド

上記の例として、以下のようなものがあります。

- 事例1 装置メーカー提供のデータ変換ツール(配布が禁止されているもの)
- 事例2 Anaconda パブリック リポジトリからパッケージのインストール

> 本制限は、RDE上で実行される構造化処理プログラム についての禁止事項となります。RDEにデータを登録する 前処理 として装置メーカー提供のデータ変換ツールやAnacondaパブリックリポジトリのパッケージを利用することは問題ありません(使用許諾やライセンス的に問題がない場合)。

2.9 本書でのディレクトリ表記

以下、本書でのディレクトリまたはディレクトリを含むファイルの表記をする場合、init処理で作成された `container/` ディレクトリを起点として表記することにします。

例: data/inputdata/sample.dat (本書の場合$HOME/handson/tutorial/container/data/inputdata/sample.datを意味します。

> `container/` 配下以外の場所のファイルについて記述する場合は、フルパスで記述するか、$HOMEに続いて記述するなどして、それと分かるように記述します。

例1: "/etc/passwd" (絶対パスの場合)

例2: $HOME/.bashrc (ユーザのホームディレクトリ直下のファイルの場合)

また、本書では"ディレクトリ"と"フォルダ"を同じ意味で使用します。

2.10 OS提供コマンドのインストールについて

本書では、`tree` や `vi` など多くのLinuxディストリビューションにて標準提供されているコマンドを利用しています。それらのコマンドはインストール直後には利用できない場合があります。

必要に応じて、インストールして利用してください。

Ubuntuにて、treeコマンドをインストールする場合の例:

```
sudo apt install tree
```

3. サンプルデータの配置

実装に先立ち、チュートリアルで利用するデータを配置します。

> 本書で利用する入力データはテキストであり、かつ、大きなものではないので、本書から"コピー&ペースト"で準備することとします。

3.1 sampla.data

入力データとして、`data/inputdata/sample.data` を以下の内容で作成します。

> コピーの際に、本書のフッタやヘッダが含まれないように注意してください。

data/inputdata/sample.data

```
[METADATA]
data_title=data_title 2
measurement_date=2023/1/25 23:08
x_label=x(unit)
y_label=y(unit)
series_number=3
[DATA]
series1
16
1,101
2,102
3,103
4,104
5,105
6.2,106
7
8,108
9.5,109
10,101
11,103
12,105
13,104
14,100
15,98
16,82
[DATA]
series2
11
1,101
2,102
3,103
4,104
5,105
6,106
9,109
10,90
12,60
13,52
14,41
[DATA]
series3
12
1,101
1.5,101.2
```

```
2,102
3,103
4,104
5,105
6,106
9,109
10,90
12,60
13,52
14,41
```

3.2 invoice.jsonファイルの更新

> 本来"invoice.json"ファイルは、RDEデータ登録画面からのデータ入力で自動的に作成された状態で構造化処理に渡ってきます。ローカル開発においては適切なinvoice.jsonを作成する手段がありませんので、本書では内容が入力されたinvoice.jsonを以下の様に手動作成することにします。
>
> 指定書式のExcelファイルに適切に設定することで、invoice.jsonを生成するツールを別途提供しています。以下のURLを参照ください。
>
> https://github.com/nims-mdpf/RDE_datasettemplate-schemafile-make-tool

RDEToolKitのinit処理にて、invoice.jsonが作成されていますが、以下の内容で、`data/invoice/invoice.json`ファイルを上書きします。

```
{
    "datasetId": "8fdec13d-bca2-4808-8753-2bb7e4e9c927",
    "basic": {
        "dateSubmitted": "2023-01-26",
        "dataOwnerId": "119cae3d3612df5f6cf7085fb8deaa2d1b85ce963536323462353734",
        "dataName": "NIMS_TRIAL_20230126b",
        "instrumentId": "413e53fb-aec9-41f8-ae55-3f88f6cd8d41",
        "experimentId": null,
        "description": ""
    },
    "custom": {
        "measurement_date": "2023-01-25",
        "invoice_string1": "送状文字入力値1",
        "invoice_string2": null,
        "is_devided": "devided",
        "is_private_raw": "private"
    },
    "sample": {
        "sampleId": "4d6de819-4b42-4dfe-9619-a0a0588653bc",
        "names": [
            "試料"
        ]
```

```
    }
}
```

> "invoice_string2"に指定しているnullは、データ登録時に入力が無かったことを意味します。RDEToolKitの古いバージョンでは、スキーマチェックで異常として検知されてしまうことがありました。その場合は、RDEToolKitのバージョンを新しいものに入れ替えてテストしてください。
>
> 本書の後半で、タイトル("basic"中の"dataName")に変更を加える処理を実施します。"dataName"の末尾に"/(2024)"のような文言が付加されている場合は、当該処理が正常に稼働したかの確認が難しくなるので、この時点でその部分を削除しておいてください。

3.3 invoice.schema.jsonの生成

> invoice.schema.jsonは、RDEデータ登録画面での入力画面を生成するのに用いられます。すなわち、invoice.jsonは、invoice.schema.jsonから、(少なくとも骨組みについては)生成されることになります。invoice.schema.jsonの作成は、開発者が実行する"開発"業務に含まれますが、それについては別途記述します。ここでは、以下の内容でinvoce.schema.jsonを作成(コピー&ペースト)してください。

`data/tasksupport/invoice.schema.json`を以下の内容で上書きします。

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "TEST_NIMS_Sample_Template_v0.0",
    "description": "None",
    "type": "object",
    "required": [
        "custom",
        "sample"
    ],
    "properties": {
        "custom": {
            "type": "object",
            "label": {
                "ja": "固有情報",
                "en": "Custom Information"
            },
            "required": [],
            "properties": {
                "measurement_date": {
                    "label": {
                        "ja": "計測年月日",
                        "en": "measurement_date"
                    },
                    "type": "string",
                    "format": "date"
                },
                "invoice_string1": {
                    "label": {
                        "ja": "文字列1",
                        "en": "invoice_string1"
                    },
                    "type": "string"
                },
                "invoice_string2": {
                    "label": {
                        "ja": "文字列2",
                        "en": "invoice_string2"
                    },
                    "type": "string"
```

```
                },
                "is_devided": {
                    "label": {
                        "ja": "複数ファイル区分",
                        "en": "is_devided"
                    },
                    "type": "string"
                },
                "is_private_raw": {
                    "label": {
                        "ja": "公開区分",
                        "en": "is_private_raw"
                    },
                    "type": "string"
                }
            }
        },
        "sample" : {
            "type": "object",
            "label": {
                "ja": "試料情報",
                "en": "Sample Information"
            },
            "required": [
                "names"
            ],
            "properties": {
                "generalAttributes": {
                    "type": "array",
                    "items": [
                    ]
                }
            }
        }
    }
}
```

> RDEToolKit v1.3.2より前のバージョンでは、data/tasksupport/invoice.schema.json のルートノード直下の"required"句に"sample"の指定が無く、かつ、data/invoice/invoice.json にsample句が記述されている場合でもエラーにはなりませんでしたが、RDEToolKit v1.3.2以降のバージョンではバリデーション機能の強化に伴いエラーとなります。試料情報を入力する場合は、適切にinvoice.schema.jsonを記述する必要があります。

3.4 metadata-def.jsonの変更

どういった項目を"メタデータ"とするかについてはmetadata-def.jsonファイルに定義しておく必要があります。

> metadata.jsonに値を設定して書き出す処理は、構造化処理の中で実施する必要があります。

`data/tasksupport/metadata-def.json` を以下の内容で上書きします。

```
{
    "data_title": {
        "name": {
            "ja": "データ名",
            "en": "data title"
        },
        "schema": {
            "type": "string"
        },
        "order": 1,
        "mode": "Derived from data"
```

```
    },
    "measurement_date": {
        "name": {
            "ja": "測定日時",
            "en": "measurement date"
        },
        "schema": {
            "type": "string",
            "format": "date-time"
        },
        "order": 2,
        "mode": "Derived from data"
    },
    "x_label": {
        "name": {
            "ja": "独立変数(X軸ラベル)",
            "en": "x-label"
        },
        "schema": {
            "type": "string"
        },
        "order": 3,
        "mode": "Derived from data"
    },
    "y_label": {
        "name": {
            "ja": "従属変数(Y軸ラベル)",
            "en": "y-label"
        },
        "schema": {
            "type": "string"
        },
        "order": 4,
        "mode": "Derived from data"
    },
    "series_number": {
        "name": {
            "ja": "系列数",
            "en": "series number"
        },
        "schema": {
            "type": "integer"
        },
        "order": 5,
        "unit": "PCS",
        "mode": "Derived from data"
    },
    "series_name": {
        "name": {
            "ja": "系列名",
            "en": "series name"
        },
        "schema": {
            "type": "string"
        },
        "order": 6,
        "mode": "Analysis value",
        "variable": 1
    },
    "series_data_count": {
        "name": {
            "ja": "系列ごとデータ数",
            "en": "data count by series"
        },
        "schema": {
            "type": "integer"
        },
        "order": 7,
        "unit": "PCS",
        "mode": "Analysis value",
```

```
        "variable": 1
    },
    "series_data_mean": {
        "name": {
            "ja": "系列ごと平均値",
            "en": "data mean by series"
        },
        "schema": {
            "type": "number"
        },
        "order": 8,
        "mode": "Analysis value",
        "variable": 1
    },
    "series_data_median": {
        "name": {
            "ja": "系列ごと中央値",
            "en": "data median by series"
        },
        "schema": {
            "type": "number"
        },
        "order": 9,
        "mode": "Analysis value",
        "variable": 1
    },
    "series_data_max": {
        "name": {
            "ja": "系列ごと最大値",
            "en": "data max by series"
        },
        "schema": {
            "type": "number"
        },
        "order": 10,
        "mode": "Analysis value",
        "variable": 1
    },
    "series_data_min": {
        "name": {
            "ja": "系列ごと最小値",
            "en": "data min by series"
        },
        "schema": {
            "type": "number"
        },
        "order": 11,
        "mode": "Analysis value",
        "variable": 1
    },
    "series_data_stdev": {
        "name": {
            "ja": "系列ごと標準偏差",
            "en": "data stdev by series"
        },
        "schema": {
            "type": "number"
        },
        "order": 12,
        "mode": "Analysis value",
        "variable": 1
    },
    "measurement date": {
        "name": {
            "ja": "送状測定日時",
            "en": "measurement date from invoice"
        },
        "schema": {
            "type": "string",
            "format": "date-time"
```

```
        },
        "order": 13,
        "mode": "Invoice"
    },
    "invoice_number1": {
        "name": {
            "ja": "送状数値入力値1",
            "en": "invoice_number1"
        },
        "schema": {
            "type": "number"
        },
        "order": 14,
        "mode": "Invoice"
    },
    "invoice_number2": {
        "name": {
            "ja": "送状数値入力値2",
            "en": "invoice_number2"
        },
        "schema": {
            "type": "number"
        },
        "order": 15,
        "mode": "Invoice"
    },
    "invoice_string1": {
        "name": {
            "ja": "送状文字入力値1",
            "en": "invoice_string1"
        },
        "schema": {
            "type": "string"
        },
        "order": 16,
        "mode": "Invoice"
    },
    "invoice_string2": {
        "name": {
            "ja": "送状文字入力値2",
            "en": "inboice_string2"
        },
        "schema": {
            "type": "string"
        },
        "order": 17,
        "mode": "Invoice"
    },
    "invoice_list1": {
        "name": {
            "ja": "送状選択値1",
            "en": "invoice_list1"
        },
        "schema": {
            "type": "string"
        },
        "order": 18,
        "mode": "Invoice"
    }
}
```

> 実際のところ、上のファイルにはスペルミスが含まれています。本書中で修正しますので、ここでは上記内容のままコピー&ペーストにてファイルを上書き保存してください。

3.5 samples.zipの利用

本書同時に配布されている `samples.zip` に、上で用意したファイルと同じものが用意されています。そちらを展開し、適切なフォルダにコピーすることでも上記と同様のことが可能です。

> samples.zipに含まれる内容は、上記の内容と同じものです。必要に応じて利用してください。
>
> 同時配布がない場合は、本書の取得元等、関係者におたずねください。

unzipコマンドがない場合はインストールします。

```
sudo apt update
sudo apt install unzip
```

以下ホームディレクトリに `samples.zip` が存在しているものとして、例を示します。

```
cd $HOME
unzip samples.zip
```

これにより `samples/` フォルダ下にファイルが作成されます。

```
(tenv) $ unzip samples.zip
Archive:  samples.zip
   creating: samples/
  inflating: samples/metadata-def.json
  inflating: samples/invoice.schema.json
  inflating: samples/sample.data
  inflating: samples/invoice.json
```

所定の場所にコピーします。

```
cp samples/sample.data  tutorial/container/data/inputdata/
cp samples/invoice.json  tutorial/container/data/invoice/
cp samples/invoice.schema.json  tutorial/container/data/tasksupport/
cp samples/metadata-def.json  tutorial/container/data/tasksupport/
```

> 同様に、Pythonコードなどが含まれた `handson_codes.zip` も同時配布されます。こちらはこれ以降で作成されるPythonスクリプトが含まれています。こちらも必要に応じて利用してください。

4. 構造化処理の記述開始

4.1 main.py

以降、必要な処理を実装していきますが、それらは"どこ"に記述していけばよいでしょうか?

処理が実行される順に確認していきます。

RDE構造化処理では、最初に以下が実行されます。

```
(tenv) $ python main.py
```

> テンプレート内で、"container/"フォルダにあるmain.pyを実行するように設定するためです。そのためこの設定を変更すれば別のファイル名を実行するように変更することもできます。プログラムのメンテナンスの観点から変更しないことを推奨します。

init処理(python -m rdetoolkit init)を実行しただけのmain.pyでは、空行、コメント行を除けば、以下の2行からなります。

```
import rdetoolkit
rdetoolkit.workflows.run()
```

独自のRDE構造化処理を実行するには、この"rdetoolkit.workflows.run()"の名前付き引数"custom_dataset_function="に、実行される関数を指定する必要があります。

```
:
rdetoolkit.workflows.run(custom_dataset_function=datasets_process.dataset)
:
```

独自処理を記述したPythonスクリプトは、(main.pyと同じ階層にある)"modules/"フォルダに配置することになっています。

従って、上記のように記述した場合、"modules/datasets_process.py"にある、"dataset()"関数が実行されます。

また、スクリプトの上部で、当該スクリプトを"import"する必要があります。

以上をまとめると、"main.py"は以下の様になります。

```
import rdetoolkit
from modules import datasets_process
```

```
rdetoolkit.workflows.run(custom_dataset_function=datasets_process.dataset)
```

> 本書ではコメントについての記述は、一部を除いて行いません。実際のスクリプトにおいては、適切なコメントを付与することが推奨されます。
>
> スクリプトファイル名も、"rdetoolkit.workflows.run()"にて実行される関数名も、任意に決定することができますが、プログラムのメンテナンス性確保のため、スクリプトファイル名として"modules/datasets_process.py"を、実行される関数名として"dataset()"の利用を推奨します。同様に格納するフォルダ名も"modules/"である必要はありませんが、こちらもメンテナンスの観点から変更しないことをお勧めします。

なお、main.pyは、よくあるpythonスクリプトの様に、以下の様に記述しても構いません。

```
import rdetoolkit
from modules import datasets_process


def main():
    rdetoolkit.workflows.run(
        custom_dataset_function=datasets_process.dataset
    )

if __name__ == '__main__':
    main()
```

まだ"modules/datasets_process.py"を作成していませんので、この時点でmain.pyを実行すると、以下の様なエラーメッセージが表示されます。

```
(tenv) $ python main.py
:
ImportError: cannot import name 'datasets_process' from 'modules' (unknown location)
```

続いて、このdatasets_process.pyを記述していきます。

4.2 datasets_proces.py

最初の一歩として、"Hello RDE!"と表示されるような処理を実装してみます。

"modules/datasets_process.py"を、以下の内容で作成します。

```
from rdetoolkit.errors import catch_exception_with_message
from rdetoolkit.exceptions import StructuredError
from rdetoolkit.models.rde2types import RdeInputDirPaths, RdeOutputResourcePath


@catch_exception_with_message(error_message="ERROR: failed in data processing", error_code=50)
```

```
def dataset(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    print("Hello RDE!")
```

- "catch_exception_with_message"および"StructuredError"は、なんらかのエラーが発生した場合に利用されますので、"import"する必要があります。
- main.pyの"custom_dataset_function=〜"で指定される関数は、2つの引数を取る必要があります。1つ目はRdeInputDirPathsオブジェクトのインスタンスで、主に入力に関係するフォルダ/ファイルのpathlibオブジェクトが格納されています。2つ目はRdeOutputResourcePathオブジェクトのインスタンスで、主に出力に関係するフォルダ/ファイルのpathlibオブジェクトが格納されています。これらはRDEToolKitが自動でセットしてくれるため、多くのRDE構造化処理プログラムで共通して利用することができます。

実行すれば以下の様に出力されます。

```
(tenv) $ python main.py
Hello RDE!
```

この時点で、以下のようなエラーがでる場合があります。

```
(tenv) $ python main.py
Hello RDE!
Traceback (most recent call last):
  File "/home/devel/handson/tenv/lib/python3.12/site-packages/rdetoolkit/workflows.py", line 310, in run
    status, error_info, mode = _process_mode(
                               ^^^^^^^^^^^^^^
  File "/home/devel/handson/tenv/lib/python3.12/site-packages/rdetoolkit/workflows.py", line 221, in _process_mode
    raise StructuredError(emsg, status.error_code or 999)
rdetoolkit.exceptions.StructuredError: Processing failed in Invoice mode: Error in validating invoice.json:
1. Field: sample
   Type: required_fields_only
   Context: Field 'sample' is not allowed. Only required fields ['basic', 'custom', 'datasetId'] are permitted in invoice.json
:
```

これは、RDEToolKit v1.3.3からバリデーション処理が変更になり、`invoice.schema.json` に定義のない項目が `invoice.json` に含まれている場合に異常として処理するように変更になったことが原因です。

- 実際のRDEシステムで稼働する際は、invoice.schema.jsonに定義のない情報が含まれるinvoice.jsonが作成されることはないため、上記のようなエラーは発生しません。

このエラーの対処は、以下の2つが考えられます。

- invoice.jsonからsample部分を削除する。
- invoice.schema.jsonに、sampleに関する定義を追加し、requiredに `sample` も加える。

古いバージョンのsamples.zipに含まれるinvoice.jsonとinvoice.schema.jsonの組み合わせで上記エラーが発生する場合があります。上記エラーが発生した際には、3章で示しているinvoice.schema.jsonと同じものが `data/tasksupport/invoice.schema.json` にセットされているかを確認し、違っている場合は、3章で示したものと同じになるように修正してください。

なお、Python コードのスタイルガイド(→ https://pep8-ja.readthedocs.io/ja/latest/)には以下の様にあります。

```
import文 は次の順番でグループ化すべきです:
1. 標準ライブラリ
```

```
2. サードパーティに関連するもの
3. ローカルな アプリケーション/ライブラリ に特有のもの
```

本書も、基本的には上記に従います。

また、import文の書き方も重要となります。

例1)

```
import rdetoolkit
from modules import datasets_process


rdetoolkit.workflows.run(custom_dataset_function=datasets_process.dataset)
```

例2)

```
import rdetoolkit
from modules.datasets_process import dataset


rdetoolkit.workflows.run(custom_dataset_function=dataset)
```

例2の方が、ロジック部分の記述がすっきりするため、本書では例2の方の書き方を中心にします。

> 本質的には同じものとなります。詳しくはPython関連の情報を参照してください。

4.2.1 datasets() → custom_module()

このままdatasets()関数内で処理を記述してもよいのですが、現在取り扱っている構造化プログラムでは、多くの場合datasets()内からcustom_module()関数を呼び出し、そこに必要な構造化処理を書いていくスタイルを取っています。

そのため本書でも、そのような書き方で処理を書いていくようにします。

スクリプト内のdatasets関数の上部に、custom_module()関数を定義し、サンプルで追加していたprint文をそちらに移動します。datasets()関数では、新たに定義したcustom_module()を実行するように変更します。

結果、"modules/datasets_process.py"は以下の様になります。

```
from rdetoolkit.errors import catch_exception_with_message
from rdetoolkit.exceptions import StructuredError
from rdetoolkit.models.rde2types import RdeInputDirPaths, RdeOutputResourcePath


def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    print("Hello RDE!")

@catch_exception_with_message(error_message="ERROR: failed in data processing", error_code=50)
def dataset(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    custom_module(srcpaths, resource_paths)
```

> custom_module()関数は、datasets()関数と同じ引数を取るようにします。custom_module()関数を呼ぶ際に、dataset()が受け取った引数をそのまま渡しています。

実行結果は、同じものとなります。

```
(tenv) $ python main.py
Hello RDE!
```

4.2.2 dataset()の引数

前述の様に、dataset()関数は、以下の2つの引数を取ります。

- srcpaths
- resource_paths

前者は入力ファイルのパス情報が、後者は出力ファイルのパス情報がRDEToolKitにより自動的にセットされます。いずれも状況に応じて適切なパス情報がセットされますので、開発者はこれらを使ってパス情報を組み立てる必要があります。

pythonにて標準提供されている`pprint`を使って内容を表示してみます。

```python
from pprint import pprint

from rdetoolkit.errors import catch_exception_with_message
from rdetoolkit.exceptions import StructuredError
from rdetoolkit.models.rde2types import RdeInputDirPaths, RdeOutputResourcePath


def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    print("Hello RDE!")
    pprint(srcpaths)
    pprint(resource_paths)

@catch_exception_with_message(error_message="ERROR: failed in data processing", error_code=50)
def dataset(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    custom_module(srcpaths, resource_paths)
```

上記の様にして実行すると以下の様になります。

```
(tenv) $ python main.py
Hello RDE!
RdeInputDirPaths(inputdata=PosixPath('data/inputdata'),
                 invoice=PosixPath('data/invoice'),
                 tasksupport=PosixPath('data/tasksupport'),
                 config=Config(system=SystemSettings(extended_mode=None, save_raw=False, save_nonshared_raw=True,
save_thumbnail_image=False, magic_variable=False), multidata_tile=MultiDataTileSettings(ignore_errors=False), smarttable=None))
RdeOutputResourcePath(raw=PosixPath('data/raw'),
                      nonshared_raw=PosixPath('data/nonshared_raw'),
                      rawfiles=(PosixPath('data/inputdata/sample.data'),),
                      struct=PosixPath('data/structured'),
                      main_image=PosixPath('data/main_image'),
                      other_image=PosixPath('data/other_image'),
                      meta=PosixPath('data/meta'),
                      thumbnail=PosixPath('data/thumbnail'),
                      logs=PosixPath('data/logs'),
                      invoice=PosixPath('data/invoice'),
                      invoice_schema_json=PosixPath('data/tasksupport/invoice.schema.json'),
                      invoice_org=PosixPath('data/invoice/invoice.json'),
                      temp=PosixPath('data/temp'),
```

```
                    invoice_patch=PosixPath('data/invoice_patch'),
                    attachment=PosixPath('data/attachment'))
```

> 上記の出力フォルダはRDEToolKitにより自動的に作成されます。RDEの想定しているフォルダ名と異なるフォルダ名とならないように、構造化処理プログラム中ではフォルダの生成は行わずに、上記フォルダを使用するようにしてください。一時的な作業用フォルダなど、RDEに格納することを目的としないフォルダ名はこの限りではありません。

以上をまとめると、以下の様になります。

srcpath(入力フォルダ、設定値)

#	変数名	意味	備考
1	srcpaths.inputdata	入力ファイルのフォルダ	
2	srcpaths.invoice	invoice.jsonが格納されているフォルダ	
3	srcpaths.tasksupport	tasksupportファイル群の格納されているフォルダ	
4	srcpaths.config	設定したコンフィグ内容	

> 最後のsrcpaths.config以外は、いずれもフォルダを意味するPathオブジェクトです。ファイルを指定するために利用する際は、joinpath()を使うか、"/ (ファイル名)"を使います。

例1)

```
invoice_file = srcpaths.invoice.joinpath("invoice.json")
```

例2)

```
invoice_file = srcpaths.invoice / "invoice.json"
```

> 上記例1と例2は、invoice.jsonファイルを示すPathオブジェクトです。どちらを指定しても同じ意味です。本書では主に後者を用いることにします。

ただし、`srcpaths.inputdata` と `srcpaths.invoice` の利用には注意が必要です。Excelインボイス方式など複数のファイルを一度に処理するモードや、複数のインボイスをExcel形式などで一度に登録して利用する際は、入力ファイルとして後述する `resource_paths.rawfiles` を、invoiceファイルが格納されているフォルダとして `resource_paths.invoice` を使う必要があります。

> インボイスモードだけを利用する場合は、srcpathsを利用しても問題ありませんが、その場合Excelインボイス利用時にソース改変が必要となります。通常のインボイスモード利用時とExcelインボイス利用時でソース改変が不要になるよう、本書ではresource_pathsを利用して必要な情報を取得しています。

resource_paths(出力フォルダ、ファイル)

#	変数名	意味	備考
1	resource_paths.raw	raw出力フォルダ	共有 生データの格納用フォルダ
2	resource_paths.rawfiles	入力されたファイルを示す"配列"	
3	resource_paths.struct	structured出力フォルダ	
4	resource_paths.main_image	main_image出力フォルダ	
5	resource_paths.other_image	other_image出力フォルダ	
6	resource_paths.meta	metadata.json出力フォルダ	
7	resource_paths.thumbnail	thumbnail出力フォルダ	
8	resource_paths.logs	ログ出力フォルダ	
9	resource_paths.invoice	invoice.json出力フォルダ	入力フォルダと同じ
10	resource_paths.invoice_schema_json	invoice.schema.jsonファイル	
11	resource_paths.invoice_org	invoice.json入力ファイル	
12	resource_paths.temp	一時ファイル出力フォルダ	
13	resource_paths.invoice_patch	invoice_patchフォルダ	
14	resource_paths.attachement	attachement出力フォルダ	データ登録時に指定されなかった場合はNone
15	resource_paths.nonshared_raw	nonshared_raw出力フォルダ	非共有 生データ格納用のフォルダ

> resource_pathsには、主に出力に関するフォルダ、ファイルを示すPathオブジェクトが格納されていますが、一部は入力関連のPathオブジェクトも格納されていますので注意してください。

例1)

```
thumbnail_path =  resource_paths.thumbnail
```

例2)

```
input_files = resource_paths.rawfiles
for input_file in input_files:
    print(input_file)
```

> resource_paths.rawfilesは、入力ファイルが1個しかない場合でも、"配列"としてセットされます。

5. 例外処理(エラー処理)とフォルダ初期化

5.1 例外処理(エラー処理)

RDE構造化処理プログラム中で、以降の処理を停止するような何らかのエラー処理をする場合は、"StructuredError([エラーメッセージ[,エラー番号]])"をraiseします。

> 第一引数のエラーメッセージも省略可能ですが、エラー内容を正しく伝えるためにエラーメッセージは必ずセットしてraiseするようにしてください。

例:

```
:
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    print("Hello RDE!")
    raise StructuredError('*** RDE ERROR ***', 200)
:
```

> 前章からの継続の場合、`modules/datasets_process.py'の、 custom_module()`関数の末尾に加えてください。またpprint()は不要ですので削除してください。

これを実行すると、以下の様になります。

```
(tenv) $ python main.py
Hello RDE!

Traceback (simplified message):
Call Path:
   File: /home/devel/handson/tenv/lib/python3.12/site-packages/rdetoolkit/errors.py, Line: 43 in wrapper()
     └── File: /home/devel/handson/tutorial/container/modules/datasets_process.py, Line: 14 in dataset()
         └── File: /home/devel/handson/tutorial/container/modules/datasets_process.py, Line: 10 in custom_module()
             └──> L10: raise StructuredError('*** RDE ERROR ***', 200)

Exception Type: StructuredError
Error: *** RDE ERROR ***
```

同時に、"data/job.failed"ファイルが作成され、上記で設定した内容が書き込まれます。

```
(tenv) $ cat data/job.failed
ErrorCode=200
ErrorMessage=Error: *** RDE ERROR ***
```

RDEのジョブ実行結果はWebブラウザで確認することができますが、ジョブが失敗した場合、そのステータス情報は、この"data/job.failed"の内容をもとに表示されます。

データ登録するユーザが、「なぜ登録に失敗したのか?」を確認できるのはこの画面だけになります。そのため、"data/job.failed"にセットされる内容、つまり"StructuredError()"に与える引数は非常に重要です。

5.1.1 終了コード

構造化処理プログラムが正常に終了した場合は、シェルの終了コードは "0" で終了します。なんらかの異常が発生して終了した場合は、上述の様に"data/job.failed"に内容をセットすると同時に、終了コード "0以外"で終了します。

> "raise StructuredError()"を実行した場合、特に指定しなくても終了コード "1"で終了しますので、通常は終了コードについて考慮する必要はありません。

```
(tenv) $ python main.py > /dev/null 2>&1 ; echo $?
1
```

5.2 フォルダ初期化

前述の様に、ジョブ実行時にエラーが発生した場合は"data/job.failed"ファイルが自動で生成されます。

RDEの本番環境では、ジョブ実行の都度フォルダが新しく生成されますので問題ありませんが、ローカル開発においては、前の処理で作られた"data/job.failed"あるいはその他生成されたフォルダ/ファイルを削除しないと、修正がうまくいっているのかがわかりにくい場合があります。

そのため、必須ではありませんが、以下の様のスクリプトを利用することで、実行の都度生成されるフォルダ・ファイルをクリアすることも考慮してください。

例：reinit.sh

```
#!/bin/bash


# ./data/inputdata
#  -> 何もしない

#./data/job.failed
if [ -f ./data/job.failed ];then
    echo "./data/job.failed was removed"
    rm -f ./data/job.failed
fi

# ./data/invoice
#  -> 何もしない

#./data/tasksupport
#  -> 何もしない

#./modules
#  -> 何もしない

#./requirements.txt
#  -> 何もしない

D="
./data/logs
./data/meta
./data/main_image
./data/other_image
./data/raw
./data/nonshared_raw
./data/structured
./data/temp
```

```
./data/thumbnail
./data/attachment
./data/invoice_patch
"
for d in ${D};do
    echo "${d} was removed"
    rm -rf ${d}
done

# Python cache
rm -rf modules/__pycache__
```

実行権限を付与します。

```
chmod a+x ./reinit.sh
```

ローカル環境でのRDE構造化処理実行時、つまり"python main.py"の実行前に実行します。

```
(tenv) $ ./reinit.sh
./data/job.failed was removed
./data/logs was removed
./data/meta was removed
./data/main_image was removed
./data/other_image was removed
./data/raw was removed
./data/nonshared_raw was removed
./data/structured was removed
./data/temp was removed
./data/thumbnail was removed
./data/attachment was removed
./data/invoice_patch was removed
```

> 表示が不要の場合は、echo文をコメントアウトするなどして調整してください。
>
> 結果表示の1行目(`./data/job.failed was removed`)は、`data/job.failed` ファイルが存在する場合のみ表示されます。

なお、繰り返しになりますが、実際のRDEの環境では、RDE構造化処理が実行される都度、あたらしいフォルダ構成が生成されます。そのため、明示的に"前の処理で作成されたフォルダやファイルを削除する"処理は必要ありません。

> 本書では扱いませんが、RDEToolKitには"エクセルインボイスモード"や"マルチデータタイルモード"といったモードも存在し、それを使うと"data/divided"フォルダが作成されます。そういった場合には"rm -rf data/divided"の処理を追加する必要があります。

6. 入力ファイルのチェック

最初に、入力ファイル、つまり登録されたデータのチェックを行います。

6.1 想定

入力ファイルは様々なパターンが考えられますが、本書では、以下を想定します。

- "inputdata"ディレクトリ内に、ファイルが1個だけ存在している。
- 入力ファイルの拡張子は".data"である。

> どんな確認処理が必要かについては、それぞれのプロジェクト/データセットテンプレート毎に吟味する必要があります。

これらを順に確認し、条件に合致しない場合は、エラーとして処理します。

modules/datasets_process.pyは、以下の状態になっているものとして、ここからコードを修正していきます。

```
from rdetoolkit.errors import catch_exception_with_message
from rdetoolkit.exceptions import StructuredError
from rdetoolkit.models.rde2types import RdeInputDirPaths, RdeOutputResourcePath


def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    print("Hello RDE!")

@catch_exception_with_message(error_message="ERROR: failed in data processing", error_code=50)
def dataset(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    custom_module(srcpaths, resource_paths)
```

6.2 custom_module()にチェックロジックを実装

custom_module()内に、入力ファイルのチェックロジックを追加していきます。

```
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    # Check input file
    input_files = resource_paths.rawfiles
    if len(input_files) == 0:
        raise StructuredError("ERROR: input data not found")

    if len(input_files) > 1:
        raise StructuredError("ERROR: input data should be one file")

    raw_file_path = input_files[0]
    if raw_file_path.suffix.lower() != ".data":
        raise StructuredError(
```

```
            f"ERROR: input file is not '*.data' : {raw_file_path.name}"
        )
```

> 前の処理で追加していた `print("Hello RDE!")` は不要ですので、削除します。本書の以前の版では `input_dir = srcpaths.inputdata` 、`input_files = list(input_dir.glob("*"))` として入力ファイルを求めていました。その方法ではExcelインボイスでの登録時に問題が発生(入力ファイルを正しく認識できない)します。いずれのモードでも正しく処理が進むように、上で示した方法を利用してください。

1. 1つ目のif文で、入力ファイルの個数が0かどうかをチェックし、0だったらエラーとして処理します。
2. 次のif文で、同ファイル個数が1より多いかどうかをチェックし、多かったらエラーとして処理します。
3. その後、ファイルが1個だけ存在することが確定していますので、そのファイルの拡張子が".data"かどうかをチェックし、そうでない場合はエラーとして処理します。

6.3 確認

ダミーファイルを置き、data/inpudataフォルダにファイルが"2個"存在する状態にし、エラーとなるかを確認します。

```
(tenv) $ touch data/inputdata/dummy.txt

(tenv) $ ls data/inputdata
dummy.txt  sample.data
```

この状態でmain.pyを実行すると以下の様になります。

```
(tenv) $ python main.py

Traceback (simplified message):
Call Path:
   File: /home/devel/tenv/lib/python3.11/site-packages/rdetoolkit/errors.py, Line: 43 in wrapper()
     └── File: /home/devel/tutorial/container/modules/datasets_process.py, Line: 24 in dataset()
         └── File: /home/devel/tutorial/container/modules/datasets_process.py, Line: 14 in custom_module()
             └──> L14: raise StructuredError("ERROR: input data should be one file")

Exception Type: StructuredError
Error: ERROR: input data should be one file
```

入力ファイルが1個でない場合のエラーメッセージ"input data should be one file"が出力されていることが確認できます。

> Tracebackに出力されるファイル名や行番号は、RDEToolKitのバージョンが変わると、リファクタリングやその他の理由で変更されることがあります。

`data/job.failed` の内容を確認してみます。

```
(tenv) $ cat data/job.failed
ErrorCode=1
ErrorMessage=Error: ERROR: input data should be one file
```

画面に出たエラーメッセージと同じものが格納されていることが分かります。

> このjob.failedのErrorMessage句にセットされた内容が、RDEのWeb画面の登録結果画面に表示されます。
>
> `data/job.failed` に画面表示と同じメッセージがセットされることが確認できましたので、以降は `data/job.failed` の確認は省略します。

次のチェックの確認に行く前に、上で作成したダミーファイルを消します。

```
rm data/inputdata/dummy.txt
```

続いて、"data/inputdata/"フォルダにファイルが無い状態でどうなるかを確認します。

ファイルを消してもよいですが、あとでまた使いますのでもとに戻せるようにフォルダをリネームして、新規に同名のフォルダを作成することで、入力ファイルが存在しない状態を作ります。

```
mv data/inputdata/ data/inputdata.org
mkdir data/inputdata/
```

構造化処理プログラムを実行します。

```
(tenv) $ python main.py

Traceback (simplified message):
Call Path:
   File: /home/devel/tenv/lib/python3.11/site-packages/rdetoolkit/errors.py, Line: 43 in wrapper()
     └── File: /home/devel/tutorial/container/modules/datasets_process.py, Line: 24 in dataset()
         └── File: /home/devel/tutorial/container/modules/datasets_process.py, Line: 11 in custom_module()
             └──> L11: raise StructuredError("ERROR: input data not found")

Exception Type: StructuredError
Error: ERROR: input data not found
```

想定のエラーメッセージが出力されることが確認できました。

元に戻して、今度は入力ファイルの拡張子を".data"から".dat"に変更して、構造化処理プログラムを実行してみます。

```
(tenv) $ rm -rf data/inputdata
(tenv) $ mv data/inputdata.org/ data/inputdata
(tenv) $ mv data/inputdata/sample.data data/inputdata/sample.dat

(tenv) $ python main.py

Traceback (simplified message):
Call Path:
   File: /home/devel/tenv/lib/python3.11/site-packages/rdetoolkit/errors.py, Line: 43 in wrapper()
     └── File: /home/devel/tutorial/container/modules/datasets_process.py, Line: 24 in dataset()
         └── File: /home/devel/tutorial/container/modules/datasets_process.py, Line: 18 in custom_module()
             └──> L18: raise StructuredError(

Exception Type: StructuredError
Error: ERROR: input file is not '*.data' : sample.dat
```

エラーメッセージから、入力ファイルの拡張子が想定通りでないことを検知し、異常終了したことが分かります。

以上で、入力ファイルのチェック処理がうまく実施されました。

以降の処理のために元に戻します。

```
mv data/inputdata/sample.dat data/inputdata/sample.data
```

前章で作成した `reinit.sh` を実行し、main.pyが正常に終了することを確認します。

```
(tenv) $ ./reinit.sh
./data/logs was removed
./data/meta was removed
./data/main_image was removed
./data/other_image was removed
./data/raw was removed
./data/nonshared_raw was removed
./data/structured was removed
./data/temp was removed
./data/thumbnail was removed
./data/attachment was removed
./data/invoice_patch was removed

(tenv) $ python main.py
(tenv) $ echo $?
0
```

> 何かを表示する(→標準出力に出力する)処理を実装していませんので、この時点では何も表示されません。

7. invoice.jsonの読み込みと保存

構造化処理プログラムが起動する時、データ登録者が生データファイルのアップロードと一緒にWeb画面から入力した情報は、RDEシステムにより送り状(Invoice)データ、つまり `data/invoice/invoice.json` ファイルとして生成され、生データファイルと共に提供されます。構造化処理プログラムは、これらのファイルを入力として処理を実行し、既定の出力ファイルを属性ごとにフォルダに出力します。

Invoiceデータを「ただ登録する」だけであれば、RDE構造化処理プログラムの中で読み込むことは必要はありません。

しかし、RDE構造化処理の内容によっては、プログラム中でInvoiceの情報を読み込んで、メタデータとして転記、グラフ作成処理時にパラメータとして使用するなど、Invoiceの情報を利用したい場合もあります。

送り状(Invoice)のデータを利用したい場合の例として以下の様な場合があります。

- グラフのXY 軸の設定（描画範囲）を指定したい
- 解析処理として `ガウス分布にフィッティングする処理を行う` が、そのとき使用する `ピーク数` を入力データ毎に指定したい

これらのように、事前にRDE構造化処理プログラムの中では定義できず、ユーザが実験データを登録時に自由に決めたいような場合です。

あるいは、Invoiceの内容を生データファイルの値によって書き換えたいという要望もあるかもしれません。

そういった場合に、構造化処理プログラムの初期段階で `invoice.json` を読み込んでおくと扱いやすくなります。

7.1 想定

本書では、以下のような処理が必要である、と想定することにします。

- 通常の構造化処理プログラムは、アップロードされたファイルの公開/非公開、つまり"生データファイル"の保存先を、`raw/` (→公開の場合)または `nonshared_raw/` (→非公開の場合)のいずれのフォルダにするのかが、データセット単位で決まっていることが多い。本ハンズオンではinvoice.json内で指定されたある項目を、アップロードされたファイルを公開/非公開のいずれのフォルダにするのかの判断に利用することにする。
    - 本章では、判断に利用するための変数をセットするところまでを実装する。入力ファイルを、公開または非公開フォルダにコピーする処理(→ 実験データの永続化)は、後述する。
    - 本章ではこの変数を、改変前のinvoice.jsonをバックアップとして保存するフォルダ名を決めるために使用する方法を、例として示す。
- RDE構造化処理プログラムの中でinvoice.jsonの一部を変更する。具体的には、タイトル部に'/ (2024)'と言う文字列を付加する。新しいinvoice.jsonは、もとのファイルを上書きする。

> 実際の構造化処理プログラムでは、タイトル部に"(2024)"という文字列を付与するような処理は行わないと思われますが、「こういうことも出来る」という例として実装します。

7.2 custom_module()に読み込みロジックを実装

modules/datasets_process.py内に、以下の処理を追加します。

```
:
from rdetoolkit.invoicefile import InvoiceFile
:
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    :
    # Read invoice file
    invoice_file = resource_paths.invoice  / "invoice.json"
    invoice = InvoiceFile(invoice_file)

    # Check public or private
    is_private_raw = False if invoice["custom"]["is_private_raw"] == "share" else True

    # Backup(=Copy) invoice.json to shared/nonshared folder
    raw_dir = resource_paths.nonshared_raw if is_private_raw else resource_paths.raw
    invoice_file_backup = raw_dir / "invoice.json.orig"
    InvoiceFile.copy_original_invoice(invoice_file, invoice_file_backup)

    # Rewrite invoice
    original_data_name = invoice.invoice_obj["basic"]["dataName"]
    additional_title = "(2024)"
    if original_data_name.find(additional_title) < 0:
        # update title if not applied yet
        invoice.invoice_obj["basic"]["dataName"] = original_data_name + " / " + additional_title
        # overwrite original invoice
        invoice_file_new = resource_paths.invoice  / "invoice.json"
        invoice.overwrite(invoice_file_new)
    :
```

> 上記を手入力にて確認する際には、import文の追加を忘れないようにしてください。
>
> `Exception Type: NameError` が表示される場合は、`from rdetoolkit.invoicefile import InvoiceFile` の追加を忘れている場合が考えられます。確認し、抜けている場合は追加してください。

本書の古い版では、invoice.jsonファイルのパスを求める箇所を以下の様にしていました。

```
    invoice_file = srcpaths.invoice  / "invoice.json"
```

しかし、上記の様にした場合、Excelインボイス等複数ファイルを一度に導入するモードでの実行時に、常に1行目に記述したインボイスの内容が有効になるといった不具合が発生します。上に示したように、`resource_paths` を使用してください

```
    invoice_file = resource_paths.invoice  / "invoice.json"
```

> 絶対にExcelインボイスを使用することはないという場合は、`srcpaths` の方を利用しても同じ結果となりますが、将来Excelインボイスモードを使用する場合に備え、srcpathsの利用は控えてください。

上記例では'(2024)'がタイトルに含まれていない場合にのみ追記しています。これは、べき等性、つまり何度同じ処理をしても同じ結果となることを想定しています。より厳密には、末尾に当該文字列があるかどうか、といった別のチェックが必要となる場合があることに注意してください。

また、今回の `invoice.json` では、以下の様にprivateつまり"非公開"を指定しています。

```
:
       "is_private_raw": "private"
:
```

そのため、実行の結果として `invoice.json.orig` ファイルは、nonshared_raw/フォルダにコピーされます。

```
(tenv) $ python main.py

(tenv) $ tree data/nonshared_raw/
data/nonshared_raw/
├── invoice.json.orig
└── sample.data

1 directory, 2 files
```

入力ファイル(上記の場合は"sample.data")は、構造化処理プログラム内では何も処理を記述していませんが、RDEToolKitのデフォルト設定で"data_nonshared_raw"フォルダにコピーされるため、上記の様になります。

送り状ファイルのオリジナルと変更後のものを比べると以下の様になります。

```
(tenv) $ diff -ru data/nonshared_raw/invoice.json.orig data/invoice/invoice.json
--- data/nonshared_raw/invoice.json.orig        2025-09-01 17:18:56.038536493 +0900
+++ data/invoice/invoice.json   2025-09-01 17:18:56.038536493 +0900
@@ -3,7 +3,7 @@
     "basic": {
         "dateSubmitted": "2023-01-26",
         "dataOwnerId": "119cae3d3612df5f6cf7085fb8deaa2d1b85ce963536323462353734",
-        "dataName": "NIMS_TRIAL_20230126b",
+        "dataName": "NIMS_TRIAL_20230126b / (2024)",
         "instrumentId": "413e53fb-aec9-41f8-ae55-3f88f6cd8d41",
         "experimentId": null,
         "description": ""
```

> 想定通りにdataNameの値が変更されていることが分かります。
>
> もとのinvoice.jsonファイルの作り方により、最後に ファイル末尾に改行がありません と表示される場合があります。
> ファイル末尾の改行の有無は処理に影響しませんので、出力されなくても問題ありません。

なお、上記実装は、readf_jsonとwritef_jsonを使って以下の様に記述することもできます。

```
:
from rdetoolkit.fileops import readf_json, writef_json
:
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    :
    # Read invoice file
    invoice_file = resource_paths.invoice  / "invoice.json"
    invoice = readf_json(invoice_file)

    # Check public or private
    is_private_raw = False if invoice["custom"]["is_private_raw"] == "share" else True

    # Backup(=Copy) invoice.json to shared/nonshared folder
    raw_dir = resource_paths.nonshared_raw if is_private_raw else resource_paths.raw
    invoice_file_backup = raw_dir / "invoice.json.orig"
    writef_json(invoice_file_backup, invoice)

    # Rewrite invoice
    original_data_name = invoice["basic"]["dataName"]
    additional_title = "(2024)"
    if original_data_name.find(additional_title) < 0:
```

```
        # update title if not applied yet
        invoice["basic"]["dataName"] = original_data_name + " / " + additional_title
        invoice_file_new = resource_paths.invoice  / "invoice.json"
        writef_json(invoice_file_new, invoice)
    :
```

> この方法を使う場合は、`import InvoiceFile` は不要となります。代わりに `import readf_json, writef_json` の追加が必要になります。
>
> こちらを使う場合も、import文の記述を忘れがちですので、注意してください。

8. 入力データの読み込み

続いて、データ登録者がアップロードした入力ファイル(実験データ)を読み込む処理を考えます。

ここは、構造化処理プログラムを書く上で難しい箇所の1つです。

なぜなら、装置ごとに実験データの形式は全く別物であり、データの読み込み方は多種多様になるためです。つまり、基本的には `実験データ` それぞれに特化したデータ読み取り関数を実装する必要があります。

8.1 想定

- `data/inputdata/` フォルダにある、`sample.data` ファイルを読み込み、適切なオブジェクトとして保持する。
- 実験データのメタデータ部と計測データ部をそれぞれrawMetaObj とrawDataDf という変数に保存する。関数化した場合は、その順に返すように実装する。

8.2 custom_module()に読み込みロジックを実装

```
import io

import pandas as pd
:
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    :
    # Read input data
    DELIM = "="
    raw_data_df = None
    raw_meta_obj = None
    #
    input_file = resource_paths.rawfiles[0] # read one file only

    with open(input_file) as f:
        lines = f.readlines()
    # omit new line codes (\r and \n)
    lines_strip = [line.strip() for line in lines]

    meta_row = [i for i, line in enumerate(lines_strip) if "[METADATA]" in line]
    data_row = [i for i, line in enumerate(lines_strip) if "[DATA]" in line]
    if (meta_row != []) & (data_row !=[]):
        meta = lines_strip[(meta_row[0]+1):data_row[0]]
        # metadata to dict
        raw_meta_obj = dict(map(lambda x: tuple([x.split(DELIM)[0],
                    DELIM.join(x.split(DELIM)[1:])]), meta))
    else:
        raise StructuredError("ERROR: invalid RAW METADATA or DATA")
    # read data to data.frame
    if (int(raw_meta_obj["series_number"])!= len(data_row)):
        raise StructuredError("ERROR: unmatch series number")
    raw_data_df = []
    for i in data_row:
        series_name = lines_strip[i+1]
        cnt = int(lines_strip[i+2])
        csv = "".join(lines[(i+3):(i+3+cnt)])
        temp_df = pd.read_csv(io.StringIO(csv),header=None)
        temp_df.columns = ["x", series_name]
        raw_data_df.append(temp_df)
```

ここでは、ファイルを指定して、それを読み込み、dict形式オブジェクトにするまでを実装しています。実装内容は、利用する入力ファイルにより異なります。

上記により、2つのオブジェクト(raw_data_dfとraw_meta_obj)の2つが作成されます。

pprint()などを使って内容を確認できます。現在の `custom_module` 関数の末尾に以下を挿入します。

```
:
    from pprint import pprint
    pprint(raw_data_df)
    print("----")
    pprint(raw_meta_obj)
:
```

上記4行を追記したのち、実行すると以下の様になります。

```
(tenv) $ python main.py
[       x  series1
0    1.0    101.0
1    2.0    102.0
2    3.0    103.0
3    4.0    104.0
4    5.0    105.0
5    6.2    106.0
6    7.0      NaN
7    8.0    108.0
8    9.5    109.0
9   10.0    101.0
10  11.0    103.0
11  12.0    105.0
12  13.0    104.0
13  14.0    100.0
14  15.0     98.0
15  16.0     82.0,
      x  series2
0    1      101
1    2      102
2    3      103
3    4      104
4    5      105
5    6      106
6    9      109
7   10       90
8   12       60
9   13       52
10  14       41,
        x  series3
0    1.0    101.0
1    1.5    101.2
2    2.0    102.0
3    3.0    103.0
4    4.0    104.0
5    5.0    105.0
6    6.0    106.0
7    9.0    109.0
8   10.0     90.0
9   12.0     60.0
10  13.0     52.0
11  14.0     41.0]
----
{'data_title': 'data_title 2',
 'measurement_date': '2023/1/25 23:08',
 'series_number': '3',
```

```
 'x_label': 'x(unit)',
 'y_label': 'y(unit)'}
```

> 以降の章で、これら(raw_data_dfとraw_meta_obj)を利用します。
>
> 上で確認のために追記した4行は、対象のオブジェクトの内容確認のために追加したものです。後続の処理では使いませんので、確認後削除してください。

9. メタデータの出力

構造化処理プログラムの大きな目的の一つは"メタデータの抽出"です。基本的に、どういった構造化処理プログラムでも必ずメタデータの抽出は行われることが想定されます(溜めるだけを想定したデータセットではメタデータファイルの作成が行われない場合もあります)。

> 図の作成や数値データの可読化(→ CSVファイルへの出力) などはオプション実装、つまり要件次第で実装する場合もあれば実装しない場合もあります。

RDEでは抽出したメタデータを、"metadata.json" というJSON形式のファイルに出力する必要があります。

抽出されたメタデータがどのような項目を持ちうるかは、データセットを開設前に事前に準備する "metadata-def.json"という JSON ファイルを用意して記述しておく必要があります。本書では、すでに述べたようにmetadata-def.jsonがすでに用意された状態を想定しますが、通常は開発者が記述/作成する必要があります。

多くの場合、「入力ファイルを読み込み取得したメタデータを、ファイル(medatada.json)として出力する」ことが想定されます。本書では、それ以外にもいくつかの方法でメタデータを収集する手段について実装例を示すことにします。

9.1 想定

- 以下の順でメタデータを収集する。
    - 入力ファイルを読み込んだ際にパースしたメタデータ部分
    - invoiceファイルの `custom` 部分
    - 入力データの、数値部分
- 上記データをマージします。同じキーのデータがある場合は、後から指定したもので上書きする。
- マージしたメタデータを、ファイル(medatada.json)として出力する。

> ここでいう"パース"とは、文字データを解析して一定の形式のデータ構造に変換する処理 であり、
> 変換した結果から、 メタデータや数値データを取り出す ことを含みます。

メタデータには、大きく分けて2つのパートがあります。

1つ目は"constant"部で、本書の場合、前者の2つ、つまり「入力ファイルのメタデータ部」と「invoiceの"custom"部分」が出力される部分となります。

2つ目は、"variable"部で、上記想定の3つ目、つまり「入力ファイルの数値部」から出力されます。

9.2 custom_module()にロジックを実装

おおよその流れとしては、以下の様になります。

1. metadata-def.jsonを指定してMetaオブジェクト(RDEToolKitが提供するメタデータ操作用オブジェクト)を作成する。なお、metadata-def.jsonは、"data/tasksupport"フォルダに格納されている前提なので、フォルダ名を含む形で指定する必要があります。
2. 何らかの方法で、メタデータを収集する。
3. 上で収集したメタデータを、Metaオブジェクトに登録する。
4. metadata.jsonを出力する。

> 上記の様に作成することで、RDEToolKitがmetadata-def.jsonと連携して、metadata.jsonを作成します。

上記の流れに沿って、以下のように処理を追記します。

modules/datasets-process.py

```
import statistics as st
from rdetoolkit.rde2util import Meta

def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    :
    # Retrieve Meta data

    # create Meta instance
    metadata_def_file = srcpaths.tasksupport / "metadata-def.json"
    meta_obj = Meta(metadata_def_file)

    # get meta data
    # #1 Read Input File
    # -> input fileは前述の処理にて作成済み

    # #2 Read Inovoice
    # -> invoice.invoice_obj["custom"] は前述の処理にて作成(→読み込み)済み

    # #3 From raw data numeric data part
    s_name = []
    s_count = []
    s_mean = []
    s_median = []
    s_max = []
    s_min = []
    s_stdev = []
    for df in raw_data_df:
        d = df.dropna(axis=0)
        y = d.iloc[:, 1]
        s_name.append(d.columns[1])
        s_count.append(len(y))
        s_mean.append("{:.2f}".format(st.mean(y)))
        s_median.append("{:.2f}".format(st.median(y)))
        s_max.append(max(y))
        s_min.append(min(y))
        s_stdev.append("{:.2f}".format(st.stdev(y)))

    meta_vars = {
        "series_name": s_name,
        "series_data_count": s_count,
        "series_data_mean": s_mean,
        "series_data_median": s_median,
        "series_data_max": s_max,
```

```
        "series_data_min": s_min,
        "series_data_stdev": s_stdev,
    }

    # Merge 2 types of metadata
    const_meta_info = raw_meta_obj | invoice.invoice_obj["custom"]
    repeated_meta_info = meta_vars

    # Set metadata to meta instance
    meta_obj.assign_vals(const_meta_info)
    meta_obj.assign_vals(repeated_meta_info)

    # Write metadata
    metadata_json = resource_paths.meta / "metadata.json"
    meta_obj.writefile(metadata_json)
```

9.3 (参考) metadata.json

参考のため、上記(サンプル)コードでの出力結果を以下に示します。

```
(tenv) $ cat data/meta/metadata.json
{
    "constant": {
        "data_title": {
            "value": "data_title 2"
        },
        "measurement_date": {
            "value": "2023-01-25T00:00:00"
        },
        "x_label": {
            "value": "x(unit)"
        },
        "y_label": {
            "value": "y(unit)"
        },
        "series_number": {
            "value": 3,
            "unit": "PCS"
        },
        "invoice_string1": {
            "value": "送状文字入力値1"
        }
    },
    "variable": [
        {
            "series_name": {
                "value": "series1"
            },
            "series_data_count": {
                "value": 15,
                "unit": "PCS"
            },
            "series_data_mean": {
                "value": 102.07
            },
            "series_data_median": {
                "value": 103.0
            },
            "series_data_max": {
                "value": 109.0
            },
            "series_data_min": {
                "value": 82.0
            },
            "series_data_stdev": {
```

```
                "value": 6.27
            }
        },
        {
            "series_name": {
                "value": "series2"
            },
            "series_data_count": {
                "value": 11,
                "unit": "PCS"
            },
            "series_data_mean": {
                "value": 88.45
            },
            "series_data_median": {
                "value": 102.0
            },
            "series_data_max": {
                "value": 109
            },
            "series_data_min": {
                "value": 41
            },
            "series_data_stdev": {
                "value": 24.88
            }
        },
        {
            "series_name": {
                "value": "series3"
            },
            "series_data_count": {
                "value": 12,
                "unit": "PCS"
            },
            "series_data_mean": {
                "value": 89.52
            },
            "series_data_median": {
                "value": 101.6
            },
            "series_data_max": {
                "value": 109.0
            },
            "series_data_min": {
                "value": 41.0
            },
            "series_data_stdev": {
                "value": 24.01
            }
        }
    ]
}(tenv) $
```


> (参考)metadata-def.jsonで定義されている"invoice_string2"の項目は、metadata.jsonには出力されません。これは基となるinvoice.json中で値が"Null" (ダブルクォーテーションで囲まれていないNull)、つまり画面上での入力がないためです。invoice.json中の"invoice_string2"の値(→ Null)をダブルクォーテーションで囲んで処理を実行すれば、文字列として処理されますのでmetadata.jsonに書き出されることが確認できます。

9.4 おまけ

処理には影響しませんがmetadata-def.jsonにタイポ(誤字)があります。

以下の部分にあります。分かりますでしょうか?

```
:
   "invoice_string2": {
       "name": {
           "ja": "送状文字入力値2",
           "en": "inboice_string2"
       },
       "schema": {
           "type": "string"
       },
       "order": 17,
       "mode": "Invoice"
   },
:
```

> 答え:inboice_string2(誤) --> invoice_string2(正)

今回は"表示名部分"のタイポ(誤字)なので英語表示のブラウザで実行した場合には、表示がおかしいことになります。その後の構造化処理には影響しません。

しかし、表示名ではなく"項目名"部分を間違えた場合は、構造化処理プログラム内での指定と異なることになり、意図しない動きになる可能性があります。タイポ(誤字)には十分注意してください。

10. 実験データの可読化

データ登録者によりアップロードされる実験データのファイルは、必ずしもテキストファイルとは限りません。バイナリファイルの場合は、そのまま保存すると、データをダウンロードしたユーザ(データ利用者)が中のデータを確認したい場合、ユーザが読める形に変換する必要があり不便です。

そのため、構造化処理プログラムでバイナリデータの実験データを読み込んで処理するのであれば、それを誰もが読みやすいファイル書式に変換して保存する、つまり、可読化するとよいと考えられます。

本章では、そういった"可読化"の例として、CSV形式でのファイル出力を実施します。

10.1 想定

- 前章までの処理にて、実験データの読み込みにpandas というデータ解析ライブラリを使用したので、そのライブラリが用意しているCSV 出力関数（DataFrame クラスのto_csv メソッド）を使用し処理する。
- CSV の保存先は `structured` フォルダとする。

> 構造化処理では、可視化した画像ファイルと実験の生データ以外は `structured` フォルダに保存します。

10.2 custom_module()にロジックを実装

modules/datasets_process.py

```
:
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    :
    # Write CSV file(s)
    for d in raw_data_df:
        fname = d.columns[1].replace(" ", "") + ".csv"
        csv_file_path = resource_paths.struct / fname
        d.to_csv(csv_file_path, header=True, index=False)
```

> 変数 `raw_data_df` を使っていますので、当該変数をセットする処理以降に記述する必要があります。

10.3 (参考)出力結果の確認

上記のロジックを追加後、構造化処理プログラムを実行すると、"data/strucutured/"フォルダの下に以下のようなファイルができます。

```
(tenv) $ python main.py

(venv) $ tree data/structured/
data/structured/
├── series1.csv
├── series2.csv
└── series3.csv
```

```
1 directory, 3 files
```

それぞれのファイルは以下の様な内容となります。

```
(venv) $ cat data/structured/series1.csv
x,series1
1.0,101.0
2.0,102.0
3.0,103.0
4.0,104.0
5.0,105.0
6.2,106.0
7.0,
8.0,108.0
9.5,109.0
10.0,101.0
11.0,103.0
12.0,105.0
13.0,104.0
14.0,100.0
15.0,98.0
16.0,82.0

(venv) $ cat data/structured/series2.csv
x,series2
1,101
2,102
3,103
4,104
5,105
6,106
9,109
10,90
12,60
13,52
14,41

(venv) $ cat data/structured/series3.csv
x,series3
1.0,101.0
1.5,101.2
2.0,102.0
3.0,103.0
4.0,104.0
5.0,105.0
6.0,106.0
9.0,109.0
10.0,90.0
12.0,60.0
13.0,52.0
14.0,41.0
```

11. 実験データの可視化

構造化処理では、実験データから数値部分を読み込んだ"後処理"として、その可視化、すなわちグラフ画像作成を実施することがあります。

Python では"matplotlib"という広く使われているグラフライブラリがあり、折れ線、散布図、カラーマップ等、通常利用される種類のグラフ形式のほとんどに対応しています。

本書では、可視化作業として"matplotlib"を使ったグラフファイル(画像ファイル)の作成を実行します。

11.1 想定

- 複数の計測結果が実験データに収録されている。
- 個別の計測結果に対して、個別の画像を作成する。
- さらに、全計測結果を重ね書きした画像を作成し、それを `メイン画像` にする。

> "メイン画像"は、"data/main_image"フォルダに保存された画像ファイルで、基本的に1つのファイルだけが置かれます。

11.2 custom_module()にロジックを実装

modules/datasets_process.py

```
:
import matplotlib.pyplot as plt
:
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    :
    # Write Graph
    title = const_meta_info["data_title"]
    x_label = const_meta_info["x_label"]
    y_label = const_meta_info["y_label"]

    ## by series
    for d in raw_data_df:
        x = d.iloc[:, 0]
        y = d.iloc[:, 1]
        label = d.columns[1]
        fname = resource_paths.other_image / f"{label}.png"
        fig, ax = plt.subplots(figsize=(5, 5), facecolor="white")
        ax.plot(x, y, label=label)
        ax.set_title(title)
        ax.set_xlabel(x_label)
        ax.set_ylabel(y_label)
        ax.legend()
        fig.savefig(fname)
        plt.close(fig)

    ## all series
    fig, ax = plt.subplots(figsize=(5, 5), facecolor="lightblue")
    ax.set_xlabel(x_label)
    ax.set_ylabel(y_label)
```

```
    ax.set_title(title)
    for d in raw_data_df:
        x = d.iloc[:, 0]
        y = d.iloc[:, 1]
        label = d.columns[1]
        ax.plot(x, y, label=label)
    ax.legend()
    fname = resource_paths.main_image / "all_series.png"
    fig.savefig(fname)
    plt.close(fig)
```

> "all_series.png"は、シリーズ毎の画像ファイルとは出力先が異なり、main_imageフォルダに出力することに注意してください。

`ModuleNotFoundError: No module named 'matplotlib'` が表示される場合があります。

```
(tenv) $ python main.py
:
ModuleNotFoundError: No module named 'matplotlib'
```

この場合は、matplotlibが導入されていないと思われます。`pip list` コマンドでインストールされていないことを確認し、インストールしてください。

```
(tenv) $ pip list | grep matplotlib
(tenv) $

(tenv) $ pip install matplotlib
```

再度main.pyを実行します。

```
python main.py
```

11.3 (参考)結果の確認

以下の様に、個別のグラフ画像が3つ、それらを重ね合わせた画像が1つ、計4つのファイルが出力されます。

```
(tenv) $ tree
:
├── data
│   ├── main_image
│   │   └── all_series.png
│   ├── other_image
│   │   ├── series1.png
│   │   ├── series2.png
│   │   └── series3.png
:
```

12. サムネイル画像の作成

RDEシステムでの サムネイル画像 は、登録されたデータの一覧がタイル状に並んだときに小さく表示される画像を指します。

標準で"286 px * 200 px に収まるサイズ"の画像をサムネイル画像として想定していますが、RDEでは、それより大きい画像であっても自動的に縮小して表示されるので問題ありません。従って厳密にはサムネイル画像を作成する必要は、必ずしもありません。

また、RDEToolKitではイメージフォルダ(main_image、other_image)にある画像データを、そのままサムネイル画像としてコピーする機能をオプションとして提供しています。

本書では、前章にて作成している"全シリーズを重ねた画像"を元に、サムネイル画像を作成します。

12.1 想定

- メインイメージフォルダ("data/main_image"にあるファイル、1個に限定される)の画像ファイルを読み込み、サイズを"286 * 200"以下に縮小し、サムネイル画像保存フォルダ("data/thumbnail")に保存する。
- 保存するファイルは、(メインイメージと同様)PNG形式とし、ファイル名は"thumbnail.png"とする。

12.2 custom_module()にロジックを実装

"main_image"にある画像をもとに、表示サイズを縮小した"thumbnail.png"として作成する。

modules/datasets_process.py

```
:
from PIL import Image
:
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    :
    # Write thumbnail images
    src_img_file_path = resource_paths.main_image / "all_series.png"
    out_img_file_path = resource_paths.thumbnail / "thumbnail.png"
    closure_w = 286
    closure_h = 200

    Image.MAX_IMAGE_PIXELS = None   # オープンする画像のピクセルサイズ制限を解除する
    img_org = Image.open(src_img_file_path)
    ratio_w = closure_w / img_org.width
    ratio_h = closure_h / img_org.height
    ratio = min(ratio_w, ratio_h)
    img_re = img_org.resize((int(img_org.width * ratio), int(img_org.height * ratio)), Image.BILINEAR)  # image magickのデフォルトに
合わせてバイリニアとしている
    img_re.save(out_img_file_path)
```

実行して、確認してみます。

```
(tenv) $ python main.py
```

```
(tenv) $ tree data/thumbnail
data/thumbnail
└── thumbnail.png

1 directory, 1 file
```

> thumbnail.pngが出力されていることが確認できます。RDEシステムでは、サムネイル画像のファイル名の規定はないので、上記以外のファイル名でも問題ありません。
>
> ただし、現時点(2025年09月)では、画像ファイルの形式に制限があります。GIF形式(.gif)、JPEG形式(.jpg、.jpeg)、あるいはPNG形式(.png)のいずれかである必要があります。SVG形式といった上記想定以外のファイル形式を使った場合、データセット利用ができない(登録は出来るがダウンロードできない、等)場合があります。想定される画像ファイル形式で保存するようにしてください。

12.3 (参考)画像データをそのままサムネイル画像として使う

前述の通り、RDEToolKitはイメージフォルダにある画像ファイルをそのままthumbnail/フォルダにコピーする機能があります。その機能を利用する場合は、以下の様にします。

最初に、当該機能を使うように設定します。

> RDEToolKitのコンフィグ設定は、下記の様にmain.pyで指定する方法のほか、data/tasksupportフォルダにrdeconfig.yamlまたはrdeconfig.ymlを設置し、指定する方法もあります。詳しくはRDEToolKitのマニュアル等を参照ください。
>
> なお、どちらの方法でコンフィグ設定を行っても同じ結果となりますが、両方設定した場合は、main.py内の設定のみが反映されますので注意してください。

最初にRDEToolKitの画像をサムネイル画像として自動コピーする機能を有効にします。

main.py

```
import rdetoolkit
from modules import datasets_process


def main():
    config = rdetoolkit.config.Config()
    config.system.save_thumbnail_image = True
    rdetoolkit.workflows.run(
        config=config,
        custom_dataset_function=datasets_process.dataset
    )

if __name__ == '__main__':
    main()
```

次に、modules/datasets_process.py内に追加した"サムネイル画像生成部分"をコメントアウトします。

この状態で実行してみます。それまでの開発で出力されたファイルを消してから実行します。

```
(tenv) $ ./reinit.sh
./data/logs was removed
```

```
./data/meta was removed
./data/main_image was removed
./data/other_image was removed
./data/raw was removed
./data/nonshared_raw was removed
./data/structured was removed
./data/temp was removed
./data/thumbnail was removed
./data/attachment was removed
./data/invoice_patch was removed

(tenv) $ python main.py

(tenv) $ ls -l data/main_image/
合計 28
-rw-rw-r-- 1 enami enami 28473  5月  8 06:52 all_series.png

(tenv) $ ls -l data/thumbnail/
合計 28
-rw-rw-r-- 1 enami enami 28473  5月  8 06:52 all_series.png
```

> main_image/ フォルダにある"all_series.png"が、ファイル名も含め、そのまま data/thumbnail/ フォルダにコピーされていることが分かります。

13. 実験データの永続化

RDE では、アップロードされたデータは構造化処理前は、`data/inputdata/` フォルダに保存されています。

> タイトルに「実験データ」と付いていますが、入力されたデータファイル全般を、同様に考えることができます。

この「inputdata」フォルダに保存されたデータは、構造化処理後の"永続化"フォルダには保存されません。そのため、実験データを永続的に保存する、つまり他のユーザがダウンロードして利用できるようにするためには「inputdata」フォルダから「raw」または「nonshared_raw」フォルダにコピーする必要があります。

実験データを永続化したい場合は、公開／非公開の取扱いに応じてコピーすることになります。

- 実験データを公開する場合は、rawフォルダに保存
- 実験データを非公開にする場合は、nonshared_rawフォルダに保存

> 公開されるのは、決められた期日(→エンバーゴ日付)を過ぎてからになります。それまではrawフォルダにあったとしても、アクセス出来るのは、当該データセットを利用可能な研究グループに所属するユーザのみとなります。

13.1 想定

登録したデータが、"公開"なのか"非公開"なのかについては、通常データセット開設時に決定することになります。
RDEToolKitのデフォルトは"非公開"となります。

設定を変更することで、すべての入力ファイルを"公開"とすることができます。

また、送り状(invoice.json)などの指定値を読み込んで、1つのデータセットの中で、公開/非公開を選択することができます。

しかし、その"判定"方法についての決まった方式は存在せず、それぞれの構造化処理で自由に決定することができます。

本書では、以下の様に想定します。

- data/invoice/invoice.jsonの"custom" → "is_private_raw"の値から、データ毎に決定する。
- "is_provate_raw"の値が"share"の場合は"公開"とする。それ以外の値の場合は非公開とする。
- 公開の場合は"data/raw"フォルダに、非公開の場合は"data/nonshared_raw"フォルダにコピーする。

13.2 custom_module()にロジックを実装

RDEToolKitのデフォルト設定では、追加コーディング無しで、nonshared_rawフォルダ、つまり非公開フォルダにコピーされます。

> 古いバージョンのRDEToolKit(→ v1.0.2より前のバージョン)では、"公開"つまりrawフォルダへのコピーがデフォルトでした。なにも設定しない場合、どちらのフォルダにコピーされるのかは、RDEToolKitのバージョンによって変わる可能性がありますのでご注意ください。

```
(tenv) $ python main.py
(tenv) $ tree
:
|    ├──── nonshared_raw
|    |    └──── sample.data
:
```

すべての入力ファイルをnonshared_rawフォルダに置く場合は、デフォルト処理で問題ありません。

逆に、すべての入力ファイルをrawフォルダに置く場合は以下の様にします。

```
import rdetoolkit
from rdetoolkit.config import Config, MultiDataTileSettings, SystemSettings

from modules import datasets_process


def main():
    config = Config(
        system=SystemSettings(
            save_raw=True,
            save_nonshared_raw=False,
        ),
    )

    rdetoolkit.workflows.run(
        config=config,
        custom_dataset_function=datasets_process.dataset
    )

if __name__ == '__main__':
    main()
```

> "save_raw"に `True` を、"save_nonshared_raw"に `False` を設定します(デフォルトはそれぞれ逆になっています)。
>
> 前述のようにコンフィグ設定(上記プログラム例の"config=〜"の部分)を、外部ファイル(data/tasksupport/rdeconfig.ymlなど)に置く場合は、main.pyの修正は不要となります。

前章で示した"想定"の様に、入力内容によって変えるには、自動的にnonshared_rawまたはrawフォルダにコピーされてしまっては問題があります。そのため、`raw/` にコピーする処理および `nonshared_raw/` にコピーする処理の双方を無効にします。

自動でnonshared_rowフォルダまたはrawフォルダにコピーする処理を無効にするために、main.pyを以下の様にします。

```
import rdetoolkit
from rdetoolkit.config import Config, MultiDataTileSettings, SystemSettings
```

```
from modules import datasets_process


def main():
    config = Config(
        system=SystemSettings(
            save_raw=False,
            save_nonshared_raw=False,
        ),
    )

    rdetoolkit.workflows.run(
        config=config,
        custom_dataset_function=datasets_process.dataset
    )

if __name__ == '__main__':
    main()
```

続いて、datasets_process.pyで、rawフォルダかnonshared_rawフォルダのどちらのフォルダにコピーすれば良いかを取得し、それに応じて入力データを、適切なフォルダにコピーするように処理を記述します。

modules/datasets_process.py

```
import shutil
:
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    :
    # Copy inputdata to public (raw/) or non_public (nonshared_raw/)

    # raw_dir = resource_paths.nonshared_raw if is_private_raw else resource_paths.raw
    # input_file = resource_paths.rawfiles[0] # read one file only
    shutil.copy(input_file, raw_dir)
```

> 入力ファイルおよびコピー先のフォルダは、ここまでの処理で変数にセットされているため、そのまま利用します。そのため、上記ではコメントとしてあります。

実行すると以下の様になります。

> 送り状にて、is_private_raw項目で"private"を選択した場合の例です。

```
(tenv) $ python main.py

(tenv) $ tree
.
├── data
│   ├── raw
│   ├── nonshared_raw
│   │   ├── invoice.json.orig
│   │   └── sample.data
:
```

14. 応用編

ここからは、前半部分(以下初級編と呼びます)を踏まえ"応用編"として、構成を変更した構造化処理プログラムについて記述します。

構成を変更する場合、以下の2つが考えられます。

- 入力ファイルのチェックやグラフ作成などを、処理毎に別々のファイルに処理を記述するパターン
- 複数の似たデータファイルに対応するため、入力ファイルごとに処理クラスとして記述するパターン

本章では、初級編と同じ処理内容を、入力ファイルのチェックやグラフ作成などを、処理毎に別々のファイルに処理を記述し、全体の見通しが良くなるように書き換えます。

14.1 準備

14.1.1 RDEToolKit

初級編で使用したRDEToolKitをそのまま利用します。従ってPython仮想環境の"tenv"をそのまま利用します。

仮想環境を有効にしていない場合は有効にします。

```
$ source tenv/bin/activate
(tenv) $
```

14.1.2 フォルダ構成

初級編では、`$HOME/handson/tutorial` フォルダを利用しましたが、応用編では `$HOME/handson/advanced` にファイルを設置することとします。

フォルダを作成し、RDEToolKitの初期化を実施します。

```
(tenv) $ cd $HOME/handson

(tenv) $ mkdir advanced
(tenv) $ cd advanced

(tenv) $ python -m rdetoolkit init
Ready to develop a structured program for RDE.
Created: /home/devel/handson/advanced/container/requirements.txt
Created: /home/devel/handson/advanced/container/Dockerfile
Created: /home/devel/handson/advanced/container/data/invoice/invoice.json
Created: /home/devel/handson/advanced/container/data/tasksupport/invoice.schema.json
Created: /home/devel/handson/advanced/container/data/tasksupport/metadata-def.json
Created: /home/devel/handson/advanced/templates/tasksupport/invoice.schema.json
Created: /home/devel/handson/advanced/templates/tasksupport/metadata-def.json
Created: /home/devel/handson/advanced/input/invoice/invoice.json

Check the folder: /home/devel/handson/advanced
Done!
```

14.1.3 入力ファイル

入力ファイルとしてのsample.data、invoice.json、invoice.schema.json および matadata-def.jsonは、初級編と同じものを使うので、コピーします。

> 初級編と同様、本テキストからのコピー&ペーストにより作成しても構いません。

```
(tenv) $ cd container/

(tenv) $ pwd
/home/devel/handson/advanced/container

(tenv) $ cp $HOME/handson/tutorial/container/data/inputdata/sample.data data/inputdata/
(tenv) $ cp $HOME/handson/tutorial/container/data/invoice/invoice.json data/invoice/
(tenv) $ cp $HOME/handson/tutorial/container/data/tasksupport/invoice.schema.json data/tasksupport/
(tenv) $ cp $HOME/handson/tutorial/container/data/tasksupport/metadata-def.json data/tasksupport/
```

invoice.jsonを確認し、変更されている場合は、初期状態に戻します。

```
(tenv) $ vi data/invoice/invoice.json
```

変更箇所は以下です。

変更前

```
        "dataName": "NIMS_TRIAL_20230126b / (2024)",
```

変更後

```
        "dataName": "NIMS_TRIAL_20230126b",
```

> " / (2024)"が追加されていたら、その部分を削除します。

14.1.4 初期化用スクリプト

初期化用スクリプトも初級編で使用したものと同じものを利用するのでコピーします。

```
(tenv) $ cp $HOME/handson/tutorial/container/reinit.sh .
```

初期化用スクリプトが正常に実行できることを確認します。

```
(tenv) $ ./reinit.sh
./data/logs was removed
./data/meta was removed
./data/main_image was removed
./data/other_image was removed
./data/raw was removed
./data/nonshared_raw was removed
./data/structured was removed
./data/temp was removed
./data/thumbnail was removed
./data/attachment was removed
./data/invoice_patch was removed
```

14.1.5 ベーススクリプト

応用編のベースとなるスクリプトを配置していきます。

> これらはNIMS内で標準的なサンプルとして利用されていたものをベースとしています。これを基に改変していきますので、ここでは以下のままスクリプトを配置してください。
>
> 初級編と同様コメント部分は削除しています。

modules/inputfile_handler.py

```
from __future__ import annotations

from pathlib import Path
from typing import Any

import pandas as pd
from rdetoolkit.models.rde2types import MetaType

from modules.interfaces import IInputFileParser


class FileReader(IInputFileParser):

    def read(self, path: Path) -> tuple[MetaType, pd.DataFrame]:
        # Caution! dummy data
        self.data = pd.DataFrame([[1, 11], [2, 22], [3, 33]])
        self.meta: dict[str, str | int | float | list[Any] | bool] = {"meta1": "value1", "meta2": 2}  # Example with int value to
match the expected type
        return self.meta, self.data
```

> readメソッドには、ダミー処理が記述されています。以後の処理で書き換えていきますので、ここではこのままコピーしてください。以後のファイルも同様とします。

modules/meta_handler.py

```
from __future__ import annotations

from pathlib import Path
from typing import Any

from rdetoolkit import rde2util
from rdetoolkit.models.rde2types import MetaType, RepeatedMetaType

from modules.interfaces import IMetaParser


class MetaParser(IMetaParser[MetaType]):

    def parse(self, data: MetaType) -> tuple[MetaType, RepeatedMetaType]:
        # dummy return
        self.const_meta_info: MetaType = data
        self.repeated_meta_info: RepeatedMetaType = {"key": ["key_value1", "key_value2"], "sample": ["sample_value1",
"sample_value2"]}
        return self.const_meta_info, self.repeated_meta_info

    def save_meta(  # noqa: ANN201
        self,
        save_path: Path,
        metaobj: rde2util.Meta,
        *,
```

```
        const_meta_info: MetaType | None = None,
        repeated_meta_info: RepeatedMetaType | None = None,
    ) -> Any:
        if const_meta_info is None:
            const_meta_info = self.const_meta_info
        if repeated_meta_info is None:
            repeated_meta_info = self.repeated_meta_info
        metaobj.assign_vals(const_meta_info)
        metaobj.assign_vals(repeated_meta_info)

        # dummy return
        return metaobj.writefile(str(save_path))
```

modules/structured_handler.py

```
from __future__ import annotations

from pathlib import Path

import pandas as pd

from modules.interfaces import IStructuredDataProcessor


class StructuredDataProcessor(IStructuredDataProcessor):

    def to_csv(self, dataframe: pd.DataFrame, save_path: Path, *, header: list[str] | None = None) -> None:

        if header is not None:
            dataframe.to_csv(save_path, header=header, index=False)
        else:
            dataframe.to_csv(save_path, index=False)
```

modules/graph_handler.py

```
from __future__ import annotations

from pathlib import Path

import pandas as pd

from modules.interfaces import IGraphPlotter


class GraphPlotter(IGraphPlotter[pd.DataFrame]):

    def plot(self, data: pd.DataFrame, save_path: Path, *, title: str | None = None, xlabel: str | None = None, ylabel: str | None =
None) -> pd.DataFrame:

        return data  # dummy return value for testing purposes
```

modules/interfaces.py

```
from __future__ import annotations

from abc import ABC, abstractmethod
from pathlib import Path
from typing import Any, Generic, TypeVar

import pandas as pd
from rdetoolkit.models.rde2types import MetaType, RepeatedMetaType
from rdetoolkit.rde2util import Meta
```

```
T = TypeVar("T")


class IInputFileParser(ABC):

    @abstractmethod
    def read(self, path: Path) -> Any:  # noqa: D102
        raise NotImplementedError


class IStructuredDataProcessor(ABC):

    @abstractmethod
    def to_csv(self, dataframe: pd.DataFrame, save_path: Path, *, header: list[str] | None = None) -> Any:  # noqa: D102
        raise NotImplementedError


class IMetaParser(Generic[T], ABC):

    @abstractmethod
    def parse(self, data: T) -> Any:  # noqa: D102
        raise NotImplementedError

    @abstractmethod
    def save_meta(
        self, save_path: Path, meta: Meta, *, const_meta_info: MetaType | None = None, repeated_meta_info: RepeatedMetaType | None =
None,
    ) -> Any:
        raise NotImplementedError


class IGraphPlotter(Generic[T], ABC):

    @abstractmethod
    def plot(self, data: T, save_path: Path, *, title: str | None = None, xlabel: str | None = None, ylabel: str | None = None) ->
Any:  # noqa: D102
        raise NotImplementedError
```

modules/datasets_process.py

```
from rdetoolkit.errors import catch_exception_with_message
from rdetoolkit.models.rde2types import RdeInputDirPaths, RdeOutputResourcePath
from rdetoolkit.rde2util import Meta

from modules.graph_handler import GraphPlotter
from modules.inputfile_handler import FileReader
from modules.meta_handler import MetaParser
from modules.structured_handler import StructuredDataProcessor


class CustomProcessingCoordinator:

    def __init__(
        self,
        file_reader: FileReader,
        meta_parser: MetaParser,
        graph_plotter: GraphPlotter,
        structured_processor: StructuredDataProcessor,
    ):
        self.file_reader = file_reader
        self.meta_parser = meta_parser
        self.graph_plotter = graph_plotter
        self.structured_processor = structured_processor


def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
```

```
    module = CustomProcessingCoordinator(FileReader(), MetaParser(), GraphPlotter(), StructuredDataProcessor())
    # Read Input File
    meta, df_data = module.file_reader.read(srcpaths.inputdata)
    # Save csv
    module.structured_processor.to_csv(df_data, resource_paths.struct.joinpath("sample.csv"))
    # Meta
    module.meta_parser.parse(meta)
    module.meta_parser.save_meta(resource_paths.meta.joinpath("metadata.json"), Meta(srcpaths.tasksupport.joinpath("metadata-
def.json")))
    # Graph
    module.graph_plotter.plot(df_data, resource_paths.main_image.joinpath("sample.png"))


@catch_exception_with_message(error_message="ERROR: failed in data processing", error_code=50)
def dataset(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:

    custom_module(srcpaths, resource_paths)
```

"`from __future__ import annotations`"について

上記で示したスクリプトの1行目には、"`from __future__ import annotations`"の記述があります。

`__future__` はPythonの標準モジュールで、importするとPythonに新たな構文を実装することができます。これにより新しいPythonバージョンで追加された構文を、古いバージョンのPythonでもエラーなく実行することが可能となります。つまり同じソースコードを、複数バージョン(3.9、3.10および3.11など)で実行することが可能となります。

> すべての差異を吸収出来るわけではなく、本書では主に 型ヒント に関する記述の差異を吸収するために使用しています。

実行するPythonバージョンが"3.10以降"のように確定出来る場合は、この記述はなくても構いません。不要の場合でも不具合はでませんので、本書では入れることを前提とします。

> 本書で例示するスクリプトであってもこの記述が抜けている場合があります。実行時のエラーがPythonバージョン間の差異に起因する場合は、記述を追加して実行してみてください。

14.2 実装

初級編で実装した内容を、そのまま上の構成を使って実装します。

> 前述の通り、上で用意したスクリプト内にはダミー処理が書かれています。ダミー処理は上書きしていきます。

14.2.1 main.py

基本的に、初級編の実装と変わりません。

```
import rdetoolkit

from modules import datasets_process

def main():
    rdetoolkit.workflows.run(
        custom_dataset_function=datasets_process.dataset
    )

if __name__ == '__main__':
    main()
```

前の章にて設定したmodules/dataset_process.pyにはダミー処理が組み込まれていますので、この時点で実行すると以下の様になります。

最初に実行前のフォルダ構成を確認します。

```
(tenv) $ tree data
data
├── inputdata
│   └── sample.data
├── invoice
│   └── invoice.json
└── tasksupport
    ├── invoice.schema.json
    └── metadata-def.json

4 directories, 4 files
```

実行し、フォルダ構成を再度確認します。

```
(tenv) $ python main.py

(tenv) $ tree data
data
├── attachment (★)
├── inputdata
│   └── sample.data
├── invoice
│   └── invoice.json
├── invoice_patch (★)
├── logs (★)
│   └── rdesys.log (★)
├── main_image (★)
├── meta (★)
│   └── metadata.json (★)
├── nonshared_raw (★)
```

```
│   └── sample.data (★)
├── other_image (★)
├── raw (★)
├── structured (★)
│   └── sample.csv (★)
├── tasksupport
│   ├── invoice.schema.json
│   └── metadata-def.json
├── temp (★)
└── thumbnail (★)

15 directories, 8 files
```

・実行により、新規に作成された部分に"(★)"を付けています。

> 必要なフォルダ構成が生成されていること(新規に11フォルダ作成)と、デフォルト設定に従い、入力ファイルが `nonshared_raw/` にコピーされていることが確認できます。

14.2.2 生データのraw/フォルダへの自動コピー停止

RDEToolKitでは、生データ(data/inputdata/*)は、何も指定しないと自動的に"data/nonshared_raw"フォルダ(→非公開フォルダ)にコピーされます。

初級編で示したように、生データのコピーをスクリプト内で明示的に行うため、自動コピーを停止します。

自動コピーの停止は、`"main.py"の中で設定する` 方法と、`設定ファイルを使って設定する` 方法の2つから選択できます。

main.pyの中で設定する方法は初級編内で実施しましたので、応用編では、"設定ファイル"を置くことで同様の設定を行います。

以下の内容で、ファイルを作成します。

data/tasksupport/rdeconfig.yaml

```yaml
system:
    save_raw: False
    save_nonshared_raw: False
```

> 設定ファイルを使う場合、"main.py"内で設定内容を指定する必要はありません。"main.py"内で設定を行った場合は上記ファイルは利用されません。
>
> 設定ファイルの名称としては、`rdeconfig.yaml` の他、`rdeconfig.yml` や `pyproject.toml` が利用できます。
>
> RDEToolKit内では、main.py内の設定 → rdeconfig.yaml → rdeconfig.yml → pyproject.tomlの順に確認し、最初に見つかったものだけを使用します。
>
> また一部のバージョンのRDEToolKit(v1.3.0など)で、`save_raw: False` と `save_nonshared_raw: False` のように双方を `False` にした場合にエラーとなる場合があります。本書の例のように、双方を `False` にしたい場合は、その他のバージョンのRDEToolKitを利用してください。RDEToolKit v1.3.1以降ではエラーにならないように修正されていますので、特に制限がなければ最新版の利用を推奨します。

上記設定ファイルを設置後、それまでの実行で出力された出力ファイルを削除し、再度実行してみます。

```
(tenv) $ ./reinit.sh
:

(tenv) $ python main.py
(tenv) $ tree data
data
├── attachment
├── inputdata
│   └── sample.data
├── invoice
│   └── invoice.json
├── invoice_patch
├── logs
│   └── rdesys.log
├── main_image
├── meta
│   └── metadata.json
├── nonshared_raw
├── other_image
├── raw
├── structured
│   └── sample.csv
├── tasksupport
│   ├── invoice.schema.json
│   ├── metadata-def.json
│   └── rdeconfig.yaml
├── temp
└── thumbnail

15 directories, 8 files
```

> data/nonshared_raw/ "フォルダにも、data/raw/ フォルダにも、生データ(→ data/inputdata/)がコピーされていない*ことが確認できます。

14.2.3 modules/datasets_process.pyから余計な処理を削除

先に示したdatasets_process.pyには、ダミー処理として不必要な処理が多く設定されていますので、処理を記述する前に削除します。

具体的には、"custom_module()"関数内の記述をすべて削除します。

```
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    pass
```

> メソッド(関数)内に処理がないとエラーになりますので、何も実行しない pass だけを記述します。後述する処理を追記した場合は、不要となりますので、削除してください。

この状態で python main.py を実行すると、以前同様出力用のフォルダが作成されます。

```
(tenv) $ ./reinit.sh
:

(tenv) $ python main.py
(tenv) $ tree data
data
├── attachment
├── inputdata
│   └── sample.data
├── invoice
│   └── invoice.json
```

```
├── invoice_patch
├── logs
│   └── rdesys.log
├── main_image
├── meta
├── nonshared_raw
├── other_image
├── raw
├── structured
├── tasksupport
│   ├── invoice.schema.json
│   ├── metadata-def.json
│   └── rdeconfig.yaml
├── temp
└── thumbnail

15 directories, 6 files
```

> 上記削除により、data/meta/metadata.json と data/structured/sample.csv の2つのファイルが生成されなくなります。

14.2.4 コーディネータクラスのインスタンスの準備

各種処理を記述する際に利用すると便利なクラスをまとめた CustomProcessingCoordinator が用意されているので、このインスタンスを用意します。

modules/datasets_process.py

```
:
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:

    coordinator = CustomProcessingCoordinator(FileReader(), MetaParser(), GraphPlotter(), StructuredDataProcessor())
:
```

> 変更前は、"module"という変数を用いていましたが、pythonモジュールの格納場所が modules/ であり、区別が付けにくいので、本書では変数として coordinator を使用します。
>
> 何も実行しないという意味で"pass"を記述していた場合は、削除します。

上記状態で保存して、次に進みます。

14.2.5 入力ファイルのチェック

入力ファイルのチェック部分をinputfile_handler.pyに実装します。

modules/inputfile_handler.py

```
:
from rdetoolkit.errors import StructuredError
:
class FileReader(IInputFileParser):
    :
    def check(self, input_files: list[Path]) -> bool:
        # Check input file
        if len(input_files) == 0:
            raise StructuredError("ERROR: input data not found")

        if len(input_files) > 1:
```

```
            raise StructuredError("ERROR: input data should be one file")

        raw_file_path = input_files[0]
        if raw_file_path.suffix.lower() != ".data":
            raise StructuredError(
                f"ERROR: input file is not '*.data' : {raw_file_path.name}"
            )

        return True
```

> 関数化に伴い、最後に"return True"が追加されています。真偽値の戻り値が期待されますが、True以外の場合は、raise処理が実行されるため、False(偽)が返ることはありません。同様にraise処理が実行された場合は、それ以降は実行されませんので、else句でうける必要もないので、ここでは、elif/else句を使わない形を使用しています。
>
> readメソッドについては後で修正しますので、この時点では変更しません。

呼び出し側(modules/datasets_process.py)の方を変更します。

```
:
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    :
    # Check input file
    coordinator.file_reader.check(resource_paths.rawfiles)
:
```

生データが存在していることを確認し、構造化処理を実行してみます。

```
(tenv) $ cat data/inputdata/sample.data
[METADATA]
data_title=data_title 2
measurement_date=2023/1/25 23:08
:

(tenv) $ python main.py
(tenv) $
```

問題無く終了しました。

続いて、エラー状態を試します。

```
(tenv) $ touch data/inputdata/test.data

(tenv) $ python main.py

Traceback (simplified message):
Call Path:
   File: /home/devel/handson/tenv/lib/python3.12/site-packages/rdetoolkit/errors.py, Line: 43 in wrapper()
     └── File: /home/devel/handson/advanced/container/modules/datasets_process.py, Line: 44 in dataset()
         └── File: /home/devel/handson/advanced/container/modules/datasets_process.py, Line: 29 in custom_module()
             └── File: /home/devel/handson/advanced/container/modules/inputfile_handler.py, Line: 27 in check()
                 └──> L27: raise StructuredError("ERROR: input data should be one file")

Exception Type: StructuredError
Error: ERROR: input data should be one file
```

> 「入力ファイルが1個だけある」という前提が崩れているため、想定通りエラーとなります。

確認がすみましたので、先ほど作成したファイルを消します。

```
(tenv) $ rm data/inputdata/test.data
(tenv) $ tree data/inputdata
data/inputdata
└── sample.data

1 directory, 1 file
```

入力ファイルのチェックの内容は初級編と同じですので、他のエラー処理の確認は省略します。

14.2.6 invoice.jsonの読み込みと保存

invoice.jsonを扱うクラスを新規に作成します。

modules/invoice_handler.py

```python
from pathlib import Path

from rdetoolkit.errors import StructuredError
from rdetoolkit.invoicefile import InvoiceFile


class InvoiceParser():
    """RDE invoice.json handle
    """
    additional_title = "(2024)"

    def __init__(self):
        self.invoice_file = None
        self.invoice_obj = None
        self.raw_dir = None
        self.nonshared_raw_dir = None

    def parse(self, invoice_file):
        self.invoice_path = invoice_file
        self.invoice_obj = InvoiceFile(invoice_file)

    @property
    def is_private_raw(self) -> bool:
        if self.invoice_obj is None:
            raise StructuredError("ERROR: invoice does not set")

        invoice_dict = self.invoice_obj.invoice_obj
        # Check private or not
        custom = invoice_dict.get("custom")
        if custom is None:
            return False
        is_private_raw = custom.get("is_private_raw")
        if is_private_raw == "share":
            return False
        # other case -> "private"
        return True

    def set_dirs(self, *, raw_dir: Path|None = None, nonshared_raw_dir: Path|None = None):
        self.raw_dir = raw_dir
        self.nonshared_raw_dir = nonshared_raw_dir

    def change_title(self):
        if self.invoice_obj is None:
            raise StructuredError("ERROR: invoice does not set")
        # Update invoice title
        original_data_name = self.invoice_obj["basic"]["dataName"]
        additional_title = self.additional_title
        if original_data_name.find(additional_title) < 0:
            # update title if not applied yet
```

```
            self.invoice_obj["basic"]["dataName"] = \
                original_data_name + " / " + additional_title
            # Overwrite
            invoice_file_new = self.invoice_path
            self.invoice_obj.overwrite(invoice_file_new)

    def backup(self):
        # Backup(=Copy) invoice.json to shared/nonshared folder
        is_private_raw = self.is_private_raw
        invoice_file = self.invoice_path
        if is_private_raw:
            backup_dir = self.nonshared_raw_dir
        else:
            backup_dir = self.raw_dir
        # check backup dir
        if backup_dir is None:
            raise StructuredError("ERROR: backup dir does not set")

        invoice_backup_file = backup_dir / "invoice.json.orig"
        InvoiceFile.copy_original_invoice(invoice_file, invoice_backup_file)
```

> 上位(datasetes_process.pyのdataset()関数、もしくはcustom_module()関数)で利用していた変数 resource_paths はここでは利用できませんので、set_dirs() メソッドを利用してセットするようにしています。
>
> StorageDir クラスや DirectoryOps クラスを使用してしても実装できますが、Excelインボイス利用時の処理などが複雑になります。上記で示しているようにresource_pathsの内容をそのまま利用することをお勧めします。

これらを使って処理を実行するように"modules/datasets_process.py"に追加します。

```
:
from modules.invoice_handler import InvoiceParser
:
class CustomProcessingCoordinator:

    def __init__(
        self,
        file_reader: FileReader,
        meta_parser: MetaParser,
        graph_plotter: GraphPlotter,
        structured_processor: StructuredDataProcessor,
        invoice_parser: InvoiceParser,
    ):
        self.file_reader = file_reader
        self.meta_parser = meta_parser
        self.graph_plotter = graph_plotter
        self.structured_processor = structured_processor
        self.invoice_parser = invoice_parser


def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:

    coordinator = CustomProcessingCoordinator(
             FileReader(),
             MetaParser(),
             GraphPlotter(),
             StructuredDataProcessor(),
             InvoiceParser(),
    )

    # Check input file
    coordinator.file_reader.check(resource_paths.rawfiles)

    # Read and Update Invoice
    invoice = coordinator.invoice_parser
    invoice.parse(resource_paths.invoice / "invoice.json")
```

```
    invoice.set_dirs(
        raw_dir = resource_paths.raw,
        nonshared_raw_dir = resource_paths.nonshared_raw,
    )
    invoice.backup()
    invoice.change_title()
:
```

> CustomProcessingCoordinator クラスに、InvoiceParser クラスのインスタンスも管理するように変更します。それに伴い coordinator 変数を初期化する部分が長くなったので、複数行になるように変更します。
>
> 実行する順番に注意してください。invoice.jsonの変更を実施するメソッドの前にbackupメソッドを実行しないと、変更後の内容を"バックアップ"することになります。

main.pyを実行します。

```
(tenv) $ python main.py
(tenv) $ tree
:
|    ├── nonshared_raw
|    |    └── invoice.json.orig
:
(tenv) $ diff -ru data/nonshared_raw/invoice.json.orig data/invoice/invoice.json
--- data/nonshared_raw/invoice.json.orig        2025-09-04 15:53:19.019172504 +0900
+++ data/invoice/invoice.json   2025-09-04 15:53:19.019172504 +0900
@@ -3,7 +3,7 @@
     "basic": {
         "dateSubmitted": "2023-01-26",
         "dataOwnerId": "119cae3d3612df5f6cf7085fb8deaa2d1b85ce963536323462353734",
-        "dataName": "NIMS_TRIAL_20230126b",
+        "dataName": "NIMS_TRIAL_20230126b / (2024)",
         "instrumentId": "413e53fb-aec9-41f8-ae55-3f88f6cd8d41",
         "experimentId": null,
         "description": ""
```

次の処理に備え、上で出力したファイルを削除するなど、実行前の状態に戻しておきます。

```
(tenv) $ ./reinit.sh
:
```

> 本来は、invoice.jsonファイルもバックアップしたファイルに戻した方がよいですが、大きな影響はないため、ここではそのまま改変後のinvoice.jsonを入力値として扱うことにします。

14.2.7 入力データの読み込み

続いて、入力データを読み込む処理を記述します。

> すでに存在しているreadメソッド処理は削除してから、上の記述に変更してください。

modules/inputfile_handler.py

```
import io
:
    def read(self, input_files: list[Path]) -> tuple[MetaType, list[pd.DataFrame]]:
        # Read input data
        DELIM = "="
        rawDataDf = None
```

```
        rawMetaObj = None
        #
        input_file = input_files[0]

        with open(input_file) as f:
            lines = f.readlines()
        # omit new line codes (\r and \n)
        lines_strip = [line.strip() for line in lines]

        meta_row = [i for i, line in enumerate(lines_strip) if "[METADATA]" in line]
        data_row = [i for i, line in enumerate(lines_strip) if "[DATA]" in line]
        if (meta_row != []) & (data_row !=[]):
            meta = lines_strip[(meta_row[0]+1):data_row[0]]
            # metadata to dict
            raw_meta_obj = dict(map(lambda x: tuple([x.split(DELIM)[0],
                        DELIM.join(x.split(DELIM)[1:])]), meta))
        else:
            raise StructuredError("ERROR: invalid RAW METADATA or DATA")

        # read data to data.frame
        if (int(raw_meta_obj["series_number"])!= len(data_row)):
            raise StructuredError("ERROR: unmatch series number")
        raw_data_df = []
        for i in data_row:
            series_name = lines_strip[i+1]
            cnt = int(lines_strip[i+2])
            csv = "".join(lines[(i+3):(i+3+cnt)])
            temp_df = pd.read_csv(io.StringIO(csv),header=None)
            temp_df.columns = ["x", series_name]
            raw_data_df.append(temp_df)
        #
        return raw_meta_obj, raw_data_df
```

> 戻り値の型ヒント部分も変更されます。tuple[MetaType, pd.DataFrame] → tuple[MetaType, list[pd.DataFrame]]

これらを使って処理を実行するように"modules/datasets_process.py"に追加します。

```
:
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    :
    # Read input data
    meta, df_data = coordinator.file_reader.read(resource_paths.rawfiles)

:
```

pprintなどを使って内容を確認すると以下の様になっています。

meta

```
{'data_title': 'data_title 2',
 'measurement_date': '2023/1/25 23:08',
 'series_number': '3',
 'x_label': 'x(unit)',
 'y_label': 'y(unit)'}
```

df_data

```
[      x  series1
0    1.0    101.0
1    2.0    102.0
2    3.0    103.0
3    4.0    104.0
4    5.0    105.0
```

```
5    6.2    106.0
6    7.0      NaN
7    8.0    108.0
8    9.5    109.0
9   10.0    101.0
10  11.0    103.0
11  12.0    105.0
12  13.0    104.0
13  14.0    100.0
14  15.0     98.0
15  16.0     82.0,
      x  series2
0     1      101
1     2      102
2     3      103
3     4      104
4     5      105
5     6      106
6     9      109
7    10       90
8    12       60
9    13       52
10   14       41,
        x  series3
0     1.0    101.0
1     1.5    101.2
2     2.0    102.0
3     3.0    103.0
4     4.0    104.0
5     5.0    105.0
6     6.0    106.0
7     9.0    109.0
8    10.0     90.0
9    12.0     60.0
10   13.0     52.0
11   14.0     41.0]
```

14.2.8 メタデータの出力

メタデータの取得と出力を実装します。基本的なロジックは、初級編と同じです。

> もともとサンプルとして実装されていた部分はほとんど使いません。下記に置き換えてください。

modules/meta_handler.py

```python
from __future__ import annotations

from pathlib import Path
from typing import Any
import statistics as st

from rdetoolkit import rde2util
from rdetoolkit.models.rde2types import MetaType, RepeatedMetaType

from modules.interfaces import IMetaParser


class MetaParser(IMetaParser[MetaType]):

    def __init__(self):
        self.const_meta_info = dict()
        self.repeated_meta_info = dict()

    def parse_from_invoice(self, invoice) -> None:
        # Merge invoice info to meta
```

```
        if invoice["custom"] is not None:
            self.const_meta_info = self.const_meta_info | invoice["custom"]

    def parse_from_inputdata(self, meta, data) -> None:

        # From meta
        self.const_meta_info = self.const_meta_info | meta

        # From raw data numeric data part
        s_name = []
        s_count = []
        s_mean = []
        s_median = []
        s_max = []
        s_min = []
        s_stdev = []

        for df in data:
            d = df.dropna(axis=0)
            y = d.iloc[:, 1]
            s_name.append(d.columns[1])
            s_count.append(len(y))
            s_mean.append("{:.2f}".format(st.mean(y)))
            s_median.append("{:.2f}".format(st.median(y)))
            s_max.append(max(y))
            s_min.append(min(y))
            s_stdev.append("{:.2f}".format(st.stdev(y)))

        metaVars = {
            "series_name": s_name,
            "series_data_count": s_count,
            "series_data_mean": s_mean,
            "series_data_median": s_median,
            "series_data_max": s_max,
            "series_data_min": s_min,
            "series_data_stdev": s_stdev,
        }

        # Set variable meta
        self.repeated_meta_info = metaVars

    def parse(self, data: MetaType) -> tuple[MetaType, RepeatedMetaType]:
        #
        return self.const_meta_info, self.repeated_meta_info

    def save_meta(
        self,
        save_path: Path,
        metaobj: rde2util.Meta,
        *,
        const_meta_info: Optional[MetaType] = None,
        repeated_meta_info: Optional[RepeatedMetaType] = None
    ):
        """
        Save parsed metadata to a file using the provided Meta object
        """

        if const_meta_info is None:
            const_meta_info = self.const_meta_info
        if repeated_meta_info is None:
            repeated_meta_info = self.repeated_meta_info
        metaobj.assign_vals(const_meta_info)
        metaobj.assign_vals(repeated_meta_info)

        metaobj.writefile(save_path)
```

これらを使って処理を実行するように"modules/datasets_process.py"に追加します。

```
:
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    :
    # Read input data
    meta, df_data = coordinator.file_reader.read(resource_paths.rawfiles)

    # Meta （※ここ以下を追加する）
    meta_parser = coordinator.meta_parser
    meta_parser.parse_from_invoice(invoice.invoice_obj)
    meta_parser.parse_from_inputdata(meta, df_data)
    meta_parser.save_meta(
        resource_paths.meta.joinpath("metadata.json"),
        Meta(srcpaths.tasksupport.joinpath("metadata-def.json"))
    )
:
```

14.2.9 生データの可読化

続いて、生データの可読化、つまり、生データをCSVファイルとして出力する部分を実装します。

modules/structured_handler.py

```
from __future__ import annotations

from pathlib import Path

import pandas as pd

from modules.interfaces import IStructuredDataProcessor


class StructuredDataProcessor(IStructuredDataProcessor):

    def to_csv(self, dataframes: list[pd.DataFrame], save_path: Path, *, header: list[str] | None = None) -> None:
        # Write CSV file(s)
        for d in dataframes:
            fname = d.columns[1].replace(" ", "") + ".csv"
            csvFilePath = save_path / fname
            if header is not None:
                d.to_csv(csvFilePath, header=header, index=False)
            else:
                d.to_csv(csvFilePath, header=True, index=False)
```

これらを使って処理を実行するように"modules/datasets_process.py"に追加します。

```
:
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    :
    # Read input data
    meta, df_data = coordinator.file_reader.read(resource_paths.rawfiles)
:
    # Save csv （※ここ以下を追加する）
    coordinator.structured_processor.to_csv(df_data, resource_paths.struct)

:
```

14.2.10 生データの可視化

可読化に続き、生データの可視化を実装します。

modules/graph_handler.py

```
from __future__ import annotations

from pathlib import Path

import pandas as pd
import matplotlib.pyplot as plt

from modules.interfaces import IGraphPlotter


class GraphPlotter(IGraphPlotter[pd.DataFrame]):

    def plot(self, data: list[pd.DataFrame], save_path: Path, *, title: Optional[str] = None, xlabel: str | None = None, ylabel: str
| None = None):
        # Write Graph
        title = title if title is not None else "No title"
        xlabel = xlabel if xlabel is not None else "X-label"
        ylabel = ylabel if ylabel is not None else "Y-label"

        ## by series
        for d in data:
            x = d.iloc[:, 0]
            y = d.iloc[:, 1]
            label = d.columns[1]
            fname = save_path.other_image / f"{label}.png"
            fig, ax = plt.subplots(figsize=(5, 5), facecolor="white")
            ax.plot(x, y, label=label)
            ax.set_title(title)
            ax.set_xlabel(xlabel)
            ax.set_ylabel(ylabel)
            ax.legend()
            fig.savefig(fname)
            plt.close(fig)

        ## all series
        fig, ax = plt.subplots(figsize=(5, 5), facecolor="lightblue")
        ax.set_xlabel(xlabel)
        ax.set_ylabel(ylabel)
        ax.set_title(title)
        for d in data:
            x = d.iloc[:, 0]
            y = d.iloc[:, 1]
            label = d.columns[1]
            ax.plot(x, y, label=label)
        ax.legend()
        fname = save_path.main_image / "all_series.png"
        fig.savefig(fname)
        plt.close(fig)
```

これらを使って処理を実行するように"modules/datasets_process.py"に追加します。

```
:
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    :
    # Read input data
    meta, df_data = coordinator.file_reader.read(resource_paths.rawfiles)
:
    # Graph （※ここ以下を追加する）
    const_meta_info = meta # or coordinator.meta_parser.const_meta_info
    coordinator.graph_plotter.plot(
        df_data,
        resource_paths,
        title=const_meta_info["data_title"],
        xlabel=const_meta_info["x_label"],
        ylabel=const_meta_info["y_label"]
```

```
    )
:
```

14.2.11 サムネイル画像の作成

サムネイル画像を扱うクラスが用意されていませんので、新規に作成します。

基礎編のように、独自にコーディングしてもよいですが、RDEToolKit v1.0.2よりサムネイル画像作成に利用できる関数が追加になったので、ここではそれを使ってサムネイル画像をつくります。

以下の内容で、新規に作成します。

modules/thumbnail_handler.py

```python
from __future__ import annotations

from pathlib import Path

from rdetoolkit.models.rde2types import RdeOutputResourcePath
from rdetoolkit.img2thumb import resize_image


class ThumbnailDrawer():

    def draw(self, resource_paths: RdeOutputResourcePath) -> None:
        # Write thumbnail images
        src_img_file_path = resource_paths.main_image / "all_series.png"
        out_img_file_path = resource_paths.thumbnail / "thumbnail.png"

        # Size of thumbnail
        closure_w = 286
        closure_h = 200

        resize_image(
            src_img_file_path,
            closure_w,
            closure_h,
            out_img_file_path
        )
```

これらを使って処理を実行するように"modules/datasets_process.py"に追加します。

```python
:
from modules.thumbnail_handler import ThumbnailDrawer
:
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    :
    # Graph
    const_meta_info = meta # or coordinator.meta_parser.const_meta_info
    coordinator.graph_plotter.plot(
        df_data,
        resource_paths,
        title=const_meta_info["data_title"],
        xlabel=const_meta_info["x_label"],
        ylabel=const_meta_info["y_label"]
    )
    :
    # Thumbnail  (※ここ以下を追加する)
    thumbnail = ThumbnailDrawer()
```

```
    thumbnail.draw(resource_paths)
:
```

> グラフ画像をもとにサムネイル画像を作成しますので、グラフ画像生成の後に実行する必要があります。
>
> また、基礎編でのサムネイル画像作成とはことなり、縦横比の調整などは行っていないため、上記実装ではサムネイル画像の左右端に白色帯が作成されてしまいます。問題がある場合は、作成されるサムネイル画像の縦横サイズ設定を変更するか、基礎編での実装を使ってください。
>
> invoiceファイルの場合は、CustomProcessingCoordinator クラスに追加しましたが、ここでは追加せずに利用するようにしています。追加する形で利用しても問題ありません。

14.2.12 生データの永続化

設定により生データの永続化(nonshared_rawまたはrawフォルダへの自動コピー)を抑止しましたので、永続化処理を実装します。

> 別ファイルとして切り出すほどのロジックではないので、datasets_process.pyに直接記述することにします。

modules/datasets_process.py

```
:
import shutil
:
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    :
    # Copy inputdata to public (raw/) or non_public (nonshared_raw/)
    is_private_raw = invoice.is_private_raw
    raw_dir = resource_paths.nonshared_raw if is_private_raw else resource_paths.raw
    for input_file in resource_paths.rawfiles:
        shutil.copy(input_file, raw_dir)
```

> invoice.jsonの設定内容を使いますので、invoiceの処理の後にに記述する必要があります。
>
> 今回の例では、入力ファイルは1個であることが確定していますのでfor文を使う必要はありませんが、複数の入力ファイルを扱う場合と同じコードで実行できるようfor文を使っています。

14.2.13 まとめ

上記の様な記述を行うことにより、custom_module()関数内の記述がすっきりします。

念のため、最終的なmodules/datasets_process.pyを以下に示します。

```
import shutil

from rdetoolkit.errors import catch_exception_with_message
from rdetoolkit.models.rde2types import RdeInputDirPaths, RdeOutputResourcePath
from rdetoolkit.rde2util import Meta

from modules.graph_handler import GraphPlotter
from modules.inputfile_handler import FileReader
from modules.meta_handler import MetaParser
from modules.structured_handler import StructuredDataProcessor
from modules.invoice_handler import InvoiceParser
```

```
from modules.thumbnail_handler import ThumbnailDrawer


class CustomProcessingCoordinator:

    def __init__(
        self,
        file_reader: FileReader,
        meta_parser: MetaParser,
        graph_plotter: GraphPlotter,
        structured_processor: StructuredDataProcessor,
        invoice_parser: InvoiceParser,
    ):
        self.file_reader = file_reader
        self.meta_parser = meta_parser
        self.graph_plotter = graph_plotter
        self.structured_processor = structured_processor
        self.invoice_parser = invoice_parser


def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    coordinator = CustomProcessingCoordinator(
             FileReader(),
             MetaParser(),
             GraphPlotter(),
             StructuredDataProcessor(),
             InvoiceParser(),
    )
    # Check input file
    coordinator.file_reader.check(resource_paths.rawfiles)

    # Read and Update Invoice
    invoice = coordinator.invoice_parser
    invoice.parse(resource_paths.invoice / "invoice.json")
    invoice.set_dirs(
        raw_dir = resource_paths.raw,
        nonshared_raw_dir = resource_paths.nonshared_raw,
    )
    invoice.backup()
    invoice.change_title()

    # Read Input File
    meta, df_data = coordinator.file_reader.read(resource_paths.rawfiles)

    # Meta
    meta_parser = coordinator.meta_parser
    meta_parser.parse_from_invoice(invoice.invoice_obj)
    meta_parser.parse_from_inputdata(meta, df_data)
    meta_parser.save_meta(
        resource_paths.meta.joinpath("metadata.json"),
        Meta(srcpaths.tasksupport.joinpath("metadata-def.json"))
    )

    # Save csv
    csv_save_path = resource_paths.struct #.joinpath("sample.csv")
    coordinator.structured_processor.to_csv(df_data, csv_save_path)
    # Graph
    const_meta_info = meta # or coordinator.meta_parser.const_meta_info
    coordinator.graph_plotter.plot(
        df_data,
        resource_paths,
        title=const_meta_info["data_title"],
        xlabel=const_meta_info["x_label"],
        ylabel=const_meta_info["y_label"]
    )

    # Thumbnail
    thumbnail = ThumbnailDrawer()
    thumbnail.draw(resource_paths)
```

```
    # Copy inputdata to public (raw/) or non_public (nonshared_raw/)
    is_private_raw = invoice.is_private_raw
    raw_dir = resource_paths.nonshared_raw if is_private_raw else resource_paths.raw
    for input_file in resource_paths.rawfiles:
        shutil.copy(input_file, raw_dir)

@catch_exception_with_message(error_message="ERROR: failed in data processing", error_code=50)
def dataset(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    custom_module(srcpaths, resource_paths)
```

> 前述のように `CustomProcessingCoordinator` クラスに、ThumbnailDrawerクラスを追記した方がよいかもしませんが、本章ではここまでにしておきます。

15. 応用編2

"応用編2"として、別の実装例を示します。

本章では、以下の様なことを実装します。

- 入力ファイルごとに処理クラスとして記述する。
- 設定ファイルとして、`pyptoject.toml` を使用する。
- 実行ログを、コンソール画面上、またはユーザ用のログファイルに出力する。

15.1 準備

応用編と同じように準備していきます。

15.1.1 RDEToolKit

初級編、応用編で使用したRDEToolKitをそのまま利用します。従ってPython仮想環境の"tenv"をそのまま利用します。

仮想環境を有効にしていない場合は有効にします。

```
$ source tenv/bin/activate
(tenv) $
```

15.1.2 フォルダ構成

応用編2では"$HOME/handson/advanced2"にファイルを設置することとします。

フォルダを作成し、RDEToolKitの初期化を実施します。

```
(tenv) $ cd $HOME/handson

(tenv) $ mkdir advanced2
(tenv) $ cd advanced2

(tenv) $ python -m rdetoolkit init
Ready to develop a structured program for RDE.
Created: /home/devel/advanced2/container/requirements.txt
Created: /home/devel/advanced2/container/Dockerfile
Created: /home/devel/advanced2/container/data/invoice/invoice.json
Created: /home/devel/advanced2/container/data/tasksupport/invoice.schema.json
Created: /home/devel/advanced2/container/data/tasksupport/metadata-def.json
Created: /home/devel/advanced2/templates/tasksupport/invoice.schema.json
Created: /home/devel/advanced2/templates/tasksupport/metadata-def.json
Created: /home/devel/advanced2/input/invoice/invoice.json

Check the folder: /home/devel/advanced2
Done!
```

> 他と同様、`devel` ユーザとして実行した場合の例となります。

15.1.3 入力ファイル

入力ファイルとしてのsample.data、invoice.json、invoice.schema.json および matadata-def.jsonは、初級編と同じものを使うので、コピーしておきます。

> 初級編と同様、本テキストからのコピー&ペーストにより作成しても構いません。

```
(tenv) $ cd container/

(tenv) $ pwd
/home/devel/handson/advanced2/container

(tenv) $ cp $HOME/handson/tutorial/container/data/inputdata/sample.data data/inputdata/
(tenv) $ cp $HOME/handson/tutorial/container/data/invoice/invoice.json data/invoice/
(tenv) $ cp $HOME/handson/tutorial/container/data/tasksupport/invoice.schema.json data/tasksupport/
(tenv) $ cp $HOME/handson/tutorial/container/data/tasksupport/metadata-def.json data/tasksupport/
```

15.1.4 初期化用スクリプト

初期化用スクリプトは初級編で使用したものと同じものを利用するのでコピーします。

```
(tenv) $ cp $HOME/handson/tutorial/container/reinit.sh .
```

初期化用スクリプトが正常に実行できることを確認します。

```
(tenv) $ ./reinit.sh
./data/logs was removed
./data/meta was removed
./data/main_image was removed
./data/other_image was removed
./data/raw was removed
./data/nonshared_raw was removed
./data/structured was removed
./data/temp was removed
./data/thumbnail was removed
./data/attachment was removed
./data/invoice_patch was removed
```

15.1.5 invoice.json

invoice.jsonの"basic" → "dataName"に"/ (2024)"を付加したままかどうかを確認し、付加されている場合は修正します。

data/invoice/invoice.json

変更前

```
        "dataName": "NIMS_TRIAL_20230126b / (2024)",
```

変更後

```
        "dataName": "NIMS_TRIAL_20230126b",
```

15.2 実装

> 処理内容はいままでの例と同じですので、コードのみ示します。また今までと同じ処理であっても、"こういう書き方もできる"ということを示すために、少し変更したバージョンとなっているものもあります。軽微な修正の場合は特にコメントを記述しない場合があります。

main.py

```
from rdetoolkit import workflows

from modules.datasets_process import dataset


def main():
    workflows.run(
        custom_dataset_function=dataset
    )

if __name__ == '__main__':
    main()
```

15.2.1 設定ファイル

本章のはじめに示したように `pyproject.toml` を用意して設定ファイルとします。

> 設置場所は、`data/` フォルダと同じフォルダ(つまり通常は `container/` フォルダ)下に設置します。

pyproject.toml

```
[tool.rdetoolkit.system]
save_raw = false
save_nonshared_raw = false
```

> `multidata_tile` にデータを設定する場合は、`[tool.rdetoolkit.multidata_tile]` 節に設定します。その他独自設定を追加する場合は `[tool.rdetoolkit.(独自設定名)]` 、具体的には `[tool.rdetoolkit.user]` のような節に設定します。独自設定項目は、他の方法で設定した場合と同様、スクリプト内ではdict形式で利用可能です。
>
> tomlファイルでの真偽値は、小文字(`true` または `false`)で指定します。

15.2.2 ログ出力

RDEToolKitは、RDEToolKitが行った `作業` を、`data/logs/rdesys.log` ファイルにログ出力します。ここでは、上記とは別に、独自のログファイル出力を実装します。

RDEToolKit v1.1.0から、ユーザログ出力用の専用クラスが実装されましたので、それを使うことにします。

> ユーザログは `data/logs/rdeuser.log` への書き込みと、画面への出力の片方、または双方を実装時に選択することができます。

利用するには、datasets_process.pyなどで、以下のような設定を行います。

modules/datasets_process.py

```
from rdetoolkit.models.rde2types import RdeInputDirPaths, RdeOutputResourcePath
from rdetoolkit.rdelogger import CustomLog, log_decorator


logger = CustomLog().get_logger()

@log_decorator()
def Huga():
    pass

@log_decorator()
def Hoge():
    logger.debug('Sample log!')

def dataset(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    Hoge()
    Huga()
```

関数の定義文(def文)にデコレートする(上部に`@log_decorator()`を前置する)ことで、デコレートした関数の実行のはじめと終わりにログを出力するようになります。

また関数内部で、取得した`logger`に対してdebug()やwarning()関数を実行することで、任意の場所で任意のメッセージを出力することができます。

実行すると、以下のようになります。

```
(tenv) $ python main.py
2025-09-08 11:14:29 +0900 (INFO) Hoge            --> Start
2025-09-08 11:14:29 +0900 (DEBUG) Sample log!
2025-09-08 11:14:29 +0900 (INFO) Hoge            <-- End
2025-09-08 11:14:29 +0900 (INFO) Huga            --> Start
2025-09-08 11:14:29 +0900 (INFO) Huga            <-- End
```

同時にログファイル(data/logs/rdeuser.log)にも同じ内容がセットされます。

```
(tenv) $ cat data/logs/rdeuser.log
2025-09-08 11:14:29 +0900 (INFO) Hoge            --> Start
2025-09-08 11:14:29 +0900 (DEBUG) Sample log!
2025-09-08 11:14:29 +0900 (INFO) Hoge            <-- End
2025-09-08 11:14:29 +0900 (INFO) Huga            --> Start
2025-09-08 11:14:29 +0900 (INFO) Huga            <-- End
```

開発が終了し、ログ出力が不要となった場合は、datasets_process.pyの上の方でセットした`get_logger()`の引数として`False`を与えればログ表示、および書き込みがなくなります。

変更前 (ログ表示あり)

```
logger = CustomLog().get_logger()
```

変更後 (ログ表示無し)

```
logger = CustomLog().get_logger(False)
```

> get_logger()の引数にFalseを指定するだけです。ロジック内のソースを変更する必要はありません。
>
> ログ出力が抑止されるのは、ユーザログ(data/logs/rdeuser.log)のみです。システムログ(data/logs/rdesys.log)は引き続き出力されます。

```
(tenv) $ python main.py
(tenv) $
```

> 画面(→標準出力)には、何も表示されなくなりました。
>
> なお、ログファイル(→data/logs/rdeuser.log)に追記はされませんが、ファイルのクリア(0バイトのファイルとして更新)や削除は行われません。ファイルのクリアや削除を実施したい場合は、別途処理(reinit.sh など)を実施してください。

```
$ ./reinit.sh
./data/logs was removed
./data/meta was removed
./data/main_image was removed
./data/other_image was removed
./data/raw was removed
./data/nonshared_raw was removed
./data/structured was removed
./data/temp was removed
./data/thumbnail was removed
./data/attachment was removed
./data/invoice_patch was removed
```

再度構造化処理プログラムを実行し、ログ出力がされていないことを確認します。

```
(tenv) $ python main.py

(tenv) $ ls -l data/logs/rdeuser.log
合計 0
```

> ログファイルが存在していないことが確認できます。ただしこの場合でも、システムログ(data/logs/rdesys.log)は出力されます。

15.2.3 それ以外の実装

ユーザログ出力が確認できたので、前述のように、前章と同様の処理を実装していきます。

> 上記でテストした modules/datasets_process.py がある場合は、下記内容で上書きしてください。

また、前述の様に、前章とはクラス構成を変更しています。こういう実装もある、という参考にしてください。

modules/datasets_process.py

```
from rdetoolkit.models.rde2types import RdeInputDirPaths, RdeOutputResourcePath
from rdetoolkit.errors import catch_exception_with_message

from modules.custom_process import CustomProcess
```

```
@catch_exception_with_message(error_message="ERROR: failed in data processing", error_code=50)
def dataset(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    # Create Instance of Custom Process
    custom_process = CustomProcess(srcpaths, resource_paths)

    # Do some ...
    custom_process.check_input()
    custom_process.parse_invoice()
    custom_process.backup_invoice_file("invoice.json.orig")
    custom_process.update_invoice()
    custom_process.parse_input()
    custom_process.make_meta()
    custom_process.make_struct()
    custom_process.make_graph()
    custom_process.make_thumbnail()
    custom_process.copy_raw_files()
```

modules/StructuredProcessBase.py

```
from rdetoolkit.models.rde2types import RdeInputDirPaths, RdeOutputResourcePath
from rdetoolkit.exceptions import StructuredError


class StructuredProcessBase:
    def __init__(self, srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath):
        self.srcpaths = srcpaths
        self.resource_paths = resource_paths

    def check_input(self):
        raise NotImplementedError

    def parse_invoice(self):
        raise NotImplementedError

    def update_invoice(self, newInvoiceDict = None):
        raise NotImplementedError

    def copy_raw_files(self):
        raise NotImplementedError

    def make_meta(self):
        raise NotImplementedError

    def make_struct(self):
        raise NotImplementedError

    def make_graph(self):
        raise NotImplementedError

    def make_thumbnail(self):
        raise NotImplementedError
```

modules/custom_process.py

```
import io
import statistics as st
import shutil

import pandas as pd
import matplotlib.pyplot as plt
from PIL import Image

from rdetoolkit.models.rde2types import RdeInputDirPaths, RdeOutputResourcePath
from rdetoolkit.errors import StructuredError
from rdetoolkit.fileops import readf_json, writef_json
```

```
from rdetoolkit.rde2util import Meta
from rdetoolkit.rdelogger import CustomLog, log_decorator

from modules.StructuredProcessBase import StructuredProcessBase


logger = CustomLog().get_logger()

class CustomProcess(StructuredProcessBase):
    additional_title = "(2024)"

    @log_decorator()
    def __init__(self, srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath):
        super().__init__(srcpaths, resource_paths)
        self.invoice_file = resource_paths.invoice.joinpath('invoice.json')
        self.data_df = None
        self.const_meta_info = dict()
        self.repeated_meta_info = dict()
        self.is_private_row = None

    @log_decorator()
    def check_input(self) -> bool:
        # Check input file
        input_files = self.resource_paths.rawfiles
        if len(input_files) == 0:
            raise StructuredError("ERROR: input data not found")
        if len(input_files) > 1:
            raise StructuredError("ERROR: input data should be one file")
        raw_file_path = input_files[0]
        if raw_file_path.suffix.lower() != ".data":
            raise StructuredError(
                f"ERROR: input file is not '*.data' : {raw_file_path.name}"
            )
        return True

    @log_decorator()
    def parse_input(self) -> None:
        # Read input data
        DELIM = "="
        raw_data_df = None
        raw_meta_obj = None
        #
        input_file = self.resource_paths.rawfiles[0]

        with open(input_file) as f:
            lines = f.readlines()
        # omit new line codes (\r and \n)
        lines_strip = [line.strip() for line in lines]

        meta_row = [i for i, line in enumerate(lines_strip) if "[METADATA]" in line]
        data_row = [i for i, line in enumerate(lines_strip) if "[DATA]" in line]
        if (meta_row != []) & (data_row != []):
            meta = lines_strip[(meta_row[0]+1):data_row[0]]
            # metadata to dict
            raw_meta_obj = dict(map(lambda x: tuple([x.split(DELIM)[0],
                        DELIM.join(x.split(DELIM)[1:])]), meta))
        else:
            raise StructuredError("ERROR: invalid RAW METADATA or DATA")

        # read data to data.frame
        if (int(raw_meta_obj["series_number"]) != len(data_row)):
            raise StructuredError("ERROR: unmatch series number")
        raw_data_df = []
        for i in data_row:
            series_name = lines_strip[i+1]
            cnt = int(lines_strip[i+2])
            csv = "".join(lines[(i+3):(i+3+cnt)])
            temp_df = pd.read_csv(io.StringIO(csv),header=None)
            temp_df.columns = ["x", series_name]
            raw_data_df.append(temp_df)
```

```
        #
        self.const_meta_info = self.const_meta_info | raw_meta_obj
        self.data_df = raw_data_df

    @log_decorator()
    def parse_invoice(self) -> None:
        self.invoice_dict = readf_json(self.invoice_file)
        self.is_private_raw = self._is_private_raw()

    def _is_private_raw(self) -> bool:
        invoice_dict = self.invoice_dict
        # Check private or not
        custom = invoice_dict.get("custom")
        if custom is None:
            return False
        is_private_raw = custom.get("is_private_raw")
        if is_private_raw == "share":
            return False
        # other case -> "private"
        return True

    @log_decorator()
    def update_invoice(self, new_invoice_dict = None) -> None:
        if new_invoice_dict is not None:
            self.invoice_dict = new_invoice_dict
        else:
            self._update_invoice()
        # overwrite invoice.json
        self._overwrite_invoice_file()

    def _update_invoice(self) -> None:
        # Update invoice title
        original_data_name = self.invoice_dict["basic"]["dataName"]
        additional_title = self.additional_title
        if original_data_name.find(additional_title) < 0:
            # update title if not applied yet
            self.invoice_dict["basic"]["dataName"] = \
                original_data_name + " / " + additional_title

    def _overwrite_invoice_file(self) -> None:
        # Overwrite Invoice file
        invoice_file_new = self.invoice_file
        writef_json(invoice_file_new, self.invoice_dict)

    def backup_invoice_file(self, backup_file = None) -> None:
        # Backup(=Copy) invoice.json to shared/nonshared folder
        is_private_raw = self.is_private_raw
        if is_private_raw:
            raw_dir = self.resource_paths.nonshared_raw
        else:
            raw_dir = self.resource_paths.raw
        if backup_file is None:
            backup_file = "invoice_backup.json"
        invoice_file_new = raw_dir.joinpath(backup_file)
        writef_json(invoice_file_new, self.invoice_dict)

    @log_decorator()
    def copy_raw_files(self) -> None:
        # Copy inputdata to public (raw/) or non_public (nonshared_raw/)
        is_private_raw = self.is_private_raw
        raw_dir = self.resource_paths.nonshared_raw if is_private_raw else self.resource_paths.raw
        for input_file in self.resource_paths.rawfiles:
            shutil.copy(input_file, raw_dir)

    @log_decorator()
    def make_meta(self) -> None:
        # create meta and save meta
        self._parse_from_invoice()
        self._parse_from_inputdata()
        save_path = self.resource_paths.meta.joinpath("metadata.json")
```

```
        self._save_meta(save_path)

    def _parse_from_invoice(self) -> None:
        # Merge invoice info to meta
        if self.invoice_dict["custom"] is not None:
            self.const_meta_info = self.const_meta_info | self.invoice_dict["custom"]

    def _parse_from_inputdata(self) -> None:
        data = self.data_df

        # From raw data numeric data part
        s_name = []
        s_count = []
        s_mean = []
        s_median = []
        s_max = []
        s_min = []
        s_stdev = []

        for df in data:
            d = df.dropna(axis=0)
            y = d.iloc[:, 1]
            s_name.append(d.columns[1])
            s_count.append(len(y))
            s_mean.append("{:.2f}".format(st.mean(y)))
            s_median.append("{:.2f}".format(st.median(y)))
            s_max.append(max(y))
            s_min.append(min(y))
            s_stdev.append("{:.2f}".format(st.stdev(y)))

        meta_vars = {
            "series_name": s_name,
            "series_data_count": s_count,
            "series_data_mean": s_mean,
            "series_data_median": s_median,
            "series_data_max": s_max,
            "series_data_min": s_min,
            "series_data_stdev": s_stdev,
        }

        # Set variable meta
        self.repeated_meta_info = meta_vars

    def _save_meta(self, save_path) -> None:
        # Save metadata to file
        metadef_filepath = self.srcpaths.tasksupport.joinpath("metadata-def.json")
        metaobj = Meta(metadef_filepath)
        metaobj.assign_vals(self.const_meta_info)
        metaobj.assign_vals(self.repeated_meta_info)

        metaobj.writefile(save_path)


    @log_decorator()
    def make_struct(self) -> None:
        # Write CSV file(s)
        for d in self.data_df:
            fname = d.columns[1].replace(" ", "") + ".csv"
            csv_file_path = self.resource_paths.struct.joinpath(fname)
            d.to_csv(csv_file_path, header=True, index=False)

    @log_decorator()
    def make_graph(self,title = None, xlabel = None , ylabel = None) -> None:
        # Write Graph
        title = title if title is not None else "No title"
        xlabel = xlabel if xlabel is not None else "X-label"
        ylabel = ylabel if ylabel is not None else "Y-label"

        ## by series
        for d in self.data_df:
```

```
            x = d.iloc[:, 0]
            y = d.iloc[:, 1]
            label = d.columns[1]
            fname = self.resource_paths.other_image.joinpath(f"{label}.png")
            fig, ax = plt.subplots(figsize=(5, 5), facecolor="white")
            ax.plot(x, y, label=label)
            ax.set_title(title)
            ax.set_xlabel(xlabel)
            ax.set_ylabel(ylabel)
            ax.legend()
            fig.savefig(fname)
            plt.close(fig)

        ## all series
        fig, ax = plt.subplots(figsize=(5, 5), facecolor="lightblue")
        ax.set_xlabel(xlabel)
        ax.set_ylabel(ylabel)
        ax.set_title(title)
        for d in self.data_df:
            x = d.iloc[:, 0]
            y = d.iloc[:, 1]
            label = d.columns[1]
            ax.plot(x, y, label=label)
        ax.legend()
        fname = self.resource_paths.main_image.joinpath("all_series.png")
        fig.savefig(fname)
        plt.close(fig)

    @log_decorator()
    def make_thumbnail(self) -> None:
        # Write thumbnail images
        src_path = self.resource_paths.main_image
        src_img_file_path = src_path.joinpath("all_series.png")
        save_path = self.resource_paths.thumbnail
        out_img_file_path = save_path.joinpath("thumbnail.png")

        # Size of thumbnail
        closure_w = 286
        closure_h = 200

        Image.MAX_IMAGE_PIXELS = None
        img_org = Image.open(src_img_file_path)
        ratio_w = closure_w / img_org.width
        ratio_h = closure_h / img_org.height
        ratio = min(ratio_w, ratio_h)
        img_re = img_org.resize(
            (int(img_org.width * ratio), int(img_org.height * ratio)),
            Image.BILINEAR
        )
        img_re.save(out_img_file_path)
```

> サムネイル画像作成処理は、応用編1で示したresize_image()関数を使う方法もあります。

実行してみると、以下の様になります。

```
(tenv) $ ./reinit.sh
:
(tenv) $ python main.py
2025-09-08 11:49:50 +0900 (INFO) __init__        --> Start
2025-09-08 11:49:50 +0900 (INFO) __init__        <-- End
2025-09-08 11:49:50 +0900 (INFO) check_input     --> Start
2025-09-08 11:49:50 +0900 (INFO) check_input     <-- End
2025-09-08 11:49:50 +0900 (INFO) parse_invoice   --> Start
2025-09-08 11:49:50 +0900 (INFO) parse_invoice   <-- End
2025-09-08 11:49:50 +0900 (INFO) update_invoice  --> Start
2025-09-08 11:49:50 +0900 (INFO) update_invoice  <-- End
2025-09-08 11:49:50 +0900 (INFO) parse_input     --> Start
```

```
2025-09-08 11:49:50 +0900 (INFO) parse_input     <-- End
2025-09-08 11:49:50 +0900 (INFO) make_meta       --> Start
2025-09-08 11:49:50 +0900 (INFO) make_meta       <-- End
2025-09-08 11:49:50 +0900 (INFO) make_struct     --> Start
2025-09-08 11:49:50 +0900 (INFO) make_struct     <-- End
2025-09-08 11:49:50 +0900 (INFO) make_graph      --> Start
2025-09-08 11:49:50 +0900 (INFO) make_graph      <-- End
2025-09-08 11:49:50 +0900 (INFO) make_thumbnail  --> Start
2025-09-08 11:49:50 +0900 (INFO) make_thumbnail  <-- End
2025-09-08 11:49:50 +0900 (INFO) copy_raw_files  --> Start
2025-09-08 11:49:50 +0900 (INFO) copy_raw_files  <-- End
```

どういったファイルが生成されているかの確認をします。

```
(tenv) $ tree data
data
├── attachment
├── inputdata
│   └── sample.data
├── invoice
│   └── invoice.json
├── invoice_patch
├── logs
│   ├── rdesys.log
│   └── rdeuser.log
├── main_image
│   └── all_series.png
├── meta
│   └── metadata.json
├── nonshared_raw
│   ├── invoice.json.orig
│   └── sample.data
├── other_image
│   ├── series1.png
│   ├── series2.png
│   └── series3.png
├── raw
├── structured
│   ├── series1.csv
│   ├── series2.csv
│   └── series3.csv
├── tasksupport
│   ├── invoice.schema.json
│   └── metadata-def.json
├── temp
└── thumbnail
    └── thumbnail.png

15 directories, 17 files
```

> 初級編、応用編1と同じ結果となっています。

15.2.4 インボイスの値によって処理を変える

さらなる応用例として、以下を想定した実装を考えてみます。

- 上記をベースの処理とする。
- invoice.jsonの"custom"->"data_type"をいう項目を新設する。invoice.schema.jsonも合わせて修正する。
- "data_type"の値が"base"なら、いままで通り"dataName"に"/ (2024)"を付加する。"variant"なら"/ (2025)"を付加する。

invoice.schema.json

data/tasksupport/invoice.schema.json

```
:
    "properties": {
        "custom": {
            "type": "object",
            "label": {
                "ja": "固有情報",
                "en": "Custom Information"
            },
            "required": [
                "data_type"
            ],
            "properties": {
                "data_type": {
                    "label": {
                        "ja": "データタイプ",
                        "en": "data_type"
                    },
                    "type": "string"
                },
                "measurement_date": {
                    "label": {
                        "ja": "計測年月日",
                        "en": "measurement_date"
                    },
:
```

"required"にdata_typeを追加し、その項目のラベルやtypeの設定を追加します。

invoice.json

data/invoice/invoice.json

```
:
    "custom": {
        "data_type": "variant",
        "measurement_date": "2023-01-25",
        "invoice_string1": "送状文字入力値1",
        "invoice_string2": null,
        "is_devided": "devided",
        "is_private_raw": "private"
    },
:
```

`"data_type": "variant",` の行を追加します。

custom_process2.py

続いて、「タイトルに"/ (2024)"を付加する」処理を「タイトルに"/ (2025)"を付加する」処理に変更したクラスを新規に作成します。

modules/custom_process2.py

```
from modules.custom_process import CustomProcess
```

```
class CustomProcess2(CustomProcess):
    additional_title = "(2025)"
```

> CustomProcessを継承しているため、CustomProcessクラスからの変更点のみの記述となっていることに注意してください。つまり、CustomProcessから変更のない処理についての記述は不要です。今回の例では、追加するテキストが変更になる("(2024)" → "(2025)")だけですのでクラス変数部分の指定だけで済みます。
>
> もちろん他の方法での実装(初期化時の引数として、`additional_title` の内容を指定する、など)もあります。上記は1つの実装例として考えてください。

datasets_process.py

条件を判定して、上記で作成したクラスを使う処理を追加します。

modules/datasets_process.py

```
from rdetoolkit.models.rde2types import RdeInputDirPaths, RdeOutputResourcePath
from rdetoolkit.errors import catch_exception_with_message
from rdetoolkit.invoicefile import InvoiceFile

from modules.custom_process import CustomProcess
from modules.custom_process2 import CustomProcess2


@catch_exception_with_message(error_message="ERROR: failed in data processing", error_code=50)
def dataset(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    # Parse invoice
    invoice_file = resource_paths.invoice.joinpath("invoice.json")
    invoice = InvoiceFile(invoice_file).invoice_obj

    ## Create Instance of Custom Process
    if invoice["custom"]["data_type"] == 'variant':
        custom_process = CustomProcess2(srcpaths, resource_paths)
    else:
        custom_process = CustomProcess(srcpaths, resource_paths)

    # Do some ...
    custom_process.check_input()
    custom_process.parse_invoice()
    custom_process.backup_invoice_file("invoice.json.orig")
    custom_process.update_invoice()
    custom_process.parse_input()
    custom_process.make_meta()
    custom_process.make_struct()
    custom_process.make_graph()
    custom_process.make_thumbnail()
    custom_process.copy_raw_files()
```

> invoiceインスタンスを作成の後、"Create Instance ....."部分を `invoice`で指定された値に応じて処理を変更する ように変更します。同時にCustomProcess2のimportを忘れずに追加してください。

実行確認

"data_type"が"variant"であることを確認し、実行します。

実行前に、invoice.jsonの内容を確認します。、

```
(tenv) $ grep data_type data/invoice/invoice.json
        "data_type": "variant",
(tenv) $ grep Name data/invoice/invoice.json
        "dataName": "NIMS_TRIAL_20230126b",
```

> "/ (2024)"が付加されている場合は、エディタを使い取り除いてください。

```
(tenv) $ python main.py
:
(tenv) $ grep Name data/invoice/invoice.json
        "dataName": "NIMS_TRIAL_20230126b / (2025)",
```

> "dataName"に"/ (2025)"が付加されていることがわかります。

上記が確認できたら、次にエディタを使い、"/ (2025)"を取り除き、"data_type"の値を"base"に変更します。

```
(tenv) $ vi data/invoice/invoice.json

(tenv) $ grep data_type data/invoice/invoice.json
        "data_type": "base",
```

実行します。

```
(tenv) $ grep Name data/invoice/invoice.json
        "dataName": "NIMS_TRIAL_20230126b",

(tenv) $ python main.py
:
(tenv) $ grep Name data/invoice/invoice.json
        "dataName": "NIMS_TRIAL_20230126b / (2024)",
```

想定通り、"dataName"には"/ (2024)"が付加されています。

> 上記ソースでは、dataName 項目にすでに / (2024) が含まれている場合でも、/ (2025) を付加してしまう不具合があります。サンプルプログラムということで、ここでは対処しないことにします。

以上

16. 変更履歴

2.7.1 (2025.09.24)

- 対応RDEToolKit : v1.3.4
- 細かい文言修正

2.7.0 (2025.09.08)

- 対応RDEToolKit : v1.3.4
- Excelインボイスモードで実行する場合でもソースコード変更が不要になるように、ソースコードを変更
- ローカル開発環境についての記述を追加
- invoice.schema.jsonを使ったinvoice.jsonのバリデーション強化対応
- reinit.shの削除対象フォルダとして、`data/attachment` と `data/invoice_patch` を追加
- 作業フォルダを `$HOME` 直下から、`$HOME/handson` に変更

2.6.0 (2025.05.14)

- 対応RDEToolKit : v1.2.0
- 推奨の開発環境を `Debian12` から `Ubuntu24.04` に変更
- 上記に合わせて、推奨するPythonバージョンを `3.11.x` から `3.12.x` に変更
- フォルダ初期化用スクリプト(reinit.sh)で、ログ用フォルダも削除するように変更

2.5.1 (2025.03.11)

- 対応RDEToolKit : v1.1.1
- 15章にpyproject.tomlの設定例を追加

2.5.0 (2025.01.29)

- 対応RDEToolKit : v1.1.0

2.4.0 (2024.11.12)

- 対応RDEToolKit : v1.0.4

- PEP8準拠への修正

2.3.0 (2024.11.02)

- 対応RDEToolKit : v1.0.2