

【Python】基礎

目次

- 環境編
 - 実行環境に関して
 - 環境構築（VirtualBox 編）
 - 環境構築（Anaconda 編）
 - 困ったときは
- 基礎文法編
 - 初歩的注意
 - 基礎
 - 標準入出力
 - 条件分岐
 - 繰り返し処理
 - 関数
 - 例外処理
 - よくやる間違い
- オブジェクト編
 - データ型（組み込み）
 - オブジェクト一般
 - 数値
 - Sizedなオブジェクト全般
 - コンテナ全般
 - イテラブルオブジェクト全般
 - コレクション全般
 - 文字列
 - リスト
 - 辞書
 - タプル
 - 集合
 - fileオブジェクト (stream)
 - その他のオブジェクト
- クラス編
 - クラスの用語とその役割

【Python】基礎

目次

- 環境編
 - 実行環境に関して
 - 環境構築（VirtualBox 編）
 - 環境構築（Anaconda 編）
 - 困ったときは
- 基礎文法編
 - 初歩的注意
 - 基礎
 - 標準入出力
 - 条件分岐
 - 繰り返し処理
 - 関数
 - 例外処理
 - よくやる間違い
- オブジェクト編
 - データ型（組み込み）
 - オブジェクト一般
 - 数値
 - Sizedなオブジェクト全般
 - コンテナ全般
 - イテラブルオブジェクト全般
 - コレクション全般
 - 文字列
 - リスト
 - 辞書
 - タプル
 - 集合
 - fileオブジェクト (stream)
 - その他のオブジェクト
- クラス編
 - クラスの用語とその役割

- クラスの定義
- クラスの操作
- その他のクラスの定義と操作

環境編

■実行環境に関して

- ▶ バージョンを確認
- ▶ インタラクティブシェルを起動
- ▶ インタラクティブシェルを抜ける
- ▶ ワンライナー
- ▶ ☆ IPythonシェルを利用する
- ▶ ☆ Jupyter Notebook（=Webブラウザ上で動作するインタラクティブなPython実行環境）

■環境構築（Ubuntu 編）

（共通）Pythonのインストール

- ▶ ☆ マシンに直接インストールする場合
- ▶ ☆ DockerのPythonコンテナを実行する場合
- ▶ ☆ Dockerで立ち上げたコンテナにインストールする場合

venvによる仮想環境を使う場合

- ▶ ※ venvとはPythonが動く仮想環境を利用できるようにする標準モジュール。
- ▶ 仮想環境を新規作成
- ▶ 仮想環境を有効に（接続する）
- ▶ 仮想環境を無効に（抜ける）
- ▶ 仮想環境を削除

Pipenvによる仮想環境を使う場合

- ▶ 1. Pipenvをインストール
- ▶ ☆ 2. Pipenvによる仮想環境を新しく作成し、開く

virtualenvによる仮想環境を使う場合

- ▶ ※ virtualenvはvenvと似て異なるもの。

- クラスの定義
- クラスの操作
- その他のクラスの定義と操作

環境編

■実行環境に関して

- ▶ バージョンを確認 `$ python -V` ※大文字！
- ▶ インタラクティブシェルを起動 `$ python`
- ▶ インタラクティブシェルを抜ける `exit()` か `{Ctrl + D}`
- ▶ ワンライナー `$ python -c "code"` ※ `;` で文の区切りに
- ▶ ☆ IPythonシェルを利用する
- ▶ ☆ Jupyter Notebook（=Webブラウザ上で動作するインタラクティブなPython実行環境）

■環境構築（Ubuntu 編）

（共通）Pythonのインストール

- ▶ ☆ マシンに直接インストールする場合
- ▶ ☆ DockerのPythonコンテナを実行する場合
- ▶ ☆ Dockerで立ち上げたコンテナにインストールする場合

venvによる仮想環境を使う場合

- ▶ ※ venvとはPythonが動く仮想環境を利用できるようにする標準モジュール。
- ▶ 仮想環境を新規作成 `$ python -m venv newEnvDirPath`
- ▶ 仮想環境を有効に（接続する） `$. envDirPath/bin/activate` ※ `source` でも可
- ▶ 仮想環境を無効に（抜ける） `$ deactivate` ※仮想環境のなかで実行
- ▶ 仮想環境を削除 `$ rm -rf envDirPath`

Pipenvによる仮想環境を使う場合

- ▶ 1. Pipenvをインストール `$ pip install pipenv`
- ▶ ☆ 2. Pipenvによる仮想環境を新しく作成し、開く

virtualenvによる仮想環境を使う場合

- ▶ ※ virtualenvはvenvと似て異なるもの。

▶ virtualenvをインストール

■環境構築（Anaconda 編）

▶ ※ Anaconda を使って JupyterLab を立ち上げてやってゆく

▶ ☆ Anaconda をインストール

▶ Anaconda のバージョンを確認

▶ ☆ Anaconda（Anaconda Navigator）をアップデート

▶ ☆ Anaconda を完全にアンインストール

▶ ☆ Python をアップデート

▶ ☆ JupyterLab でプログラムを実行する

▶ ☆ JupyterLab で外部ライブラリをインストール

▶ ☆ Anaconda 付属のコマンドラインを起動

▶ ☆ (Windowsにおいて) AnacondaのPythonからPYファを実行するBATファ

仮想環境を作成し、そこで開発する場合

▶ ※ Anaconda内で作ることができる仮想環境をconda環境やconda仮想環境と呼ばれる。

▶ ※ 以下のコマンドはすべてAnaconda付属のコマンドラインシェルにおいて実行すること。

▶ 今ある仮想環境の一覧

▶ 新たに仮想環境を作成

▶ YAMLファイルに書出し

▶ YAMLファイルから作成

▶ ※ **仮想環境作成後にそれを起動（そこへ移動）するのを忘れるな！！**

▶ ※ Anacondaにインストールしたライブラリ（Anacondaにデフォルトで入っているものではなく、新たにインストールしたもの）を仮想環境で使いたい場合は、仮想環境に再度インストールしなければならない（引き継がれない）！

▶ ※ Anacondaにまだインストールしていないライブラリを仮想環境で初めてインストールしても、Anacondaのほうにはインストールされない！

▶ 仮想環境を有効に（接続する）

▶ ※ 仮想環境を起動してもカレントディは移動しない！

▶ 仮想環境を無効に（抜ける）

▶ 仮想環境を削除

■困ったときは

▶ virtualenvをインストール

\$ pip install virtualenv

■環境構築（Anaconda 編）

▶ ※ Anaconda を使って JupyterLab を立ち上げてやってゆく

▶ ☆ Anaconda をインストール

▶ Anaconda のバージョンを確認 conda -V

▶ ☆ Anaconda（Anaconda Navigator）をアップデート

▶ ☆ Anaconda を完全にアンインストール

▶ ☆ Python をアップデート

▶ ☆ JupyterLab でプログラムを実行する

▶ ☆ JupyterLab で外部ライブラリをインストール

▶ ☆ Anaconda 付属のコマンドラインを起動

▶ ☆ (Windowsにおいて) AnacondaのPythonからPYファを実行するBATファ

仮想環境を作成し、そこで開発する場合

▶ ※ Anaconda内で作ることができる仮想環境をconda環境やconda仮想環境と呼ばれる。

▶ ※ 以下のコマンドはすべてAnaconda付属のコマンドラインシェルにおいて実行すること。

▶ 今ある仮想環境の一覧 \$ conda info -e

▶ 新たに仮想環境を作成 \$ conda create -n 環境名 python=バージョン

▶ YAMLファイルに書出し \$ conda env export > *ymlPath*

▶ YAMLファイルから作成 \$ conda env create -f *ymlPath*

▶ ※ **仮想環境作成後にそれを起動（そこへ移動）するのを忘れるな！！**

▶ ※ Anacondaにインストールしたライブラリ（Anacondaにデフォルトで入っているものではなく、新たにインストールしたもの）を仮想環境で使いたい場合は、仮想環境に再度インストールしなければならない（引き継がれない）！

▶ ※ Anacondaにまだインストールしていないライブラリを仮想環境で初めてインストールしても、Anacondaのほうにはインストールされない！

▶ 仮想環境を有効に（接続する） \$ conda activate 環境

▶ ※ 仮想環境を起動してもカレントディは移動しない！

▶ 仮想環境を無効に（抜ける） \$ conda deactivate

▶ 仮想環境を削除 \$ conda remove -n 環境 --all

■困ったときは

- ▶ ☆ 一般に、Pythonファイルを実行し、エラーが出たら
- ▶ ☆ `IndentationError: unindent does not match any outer indentation level` と出たら
- ▶ ☆ `.. : Permission denied` や `Please ask your administrator.` と出たら
- ▶ ☆ `TypeError: 'str' object is not callable` と出たら (`'str'` 以外の場合あり)

基礎文法編

■初歩的注意

- ▶ ※ 大文字と小文字を区別する言語である。
- ▶ ※ すべての値はオブジェクト（つまりメソッドを従えている）。
- ▶ ※ 変数は使い回さないこと（予期せぬ不具合を避けるため）。
- ▶ ※ 定数はサポートされていない。
- ▶ ※ 標準の文字コードは `UTF-8` 。
- ▶ ☆ 命名規則（慣習）
- ▶ ☆ 演算子の優先順位
- ▶ 予約語の一覧（リスト）
- ▶ 組み込み関数、クラスの一覧
- ▶ ※ **スコープはモジュール、関数定義 (`def`) でのみ作られ、制御構文 (`if` `for` `while` `try`) のなかでは作られない。**

■基礎

- ▶ コメントのしかた
- ▶ 1 文が長くなるとき
- ▶ ※ 実は、改行の代わりに `;` でも文の終わりになる。しかし、ワンライナー以外では使うべきでない。
- ▶ 変数を定義
- ▶ 複数の変数に同時に代入
- ▶ 変数が定義済みか
- ▶ mainで実行する際は...
- ▶ 処理をやめる

- ▶ ☆ 一般に、Pythonファイルを実行し、エラーが出たら
- ▶ ☆ `IndentationError: unindent does not match any outer indentation level` と出たら
- ▶ ☆ `.. : Permission denied` や `Please ask your administrator.` と出たら
- ▶ ☆ `TypeError: 'str' object is not callable` と出たら (`'str'` 以外の場合あり)

基礎文法編

■初歩的注意

- ▶ ※ 大文字と小文字を区別する言語である。
- ▶ ※ すべての値はオブジェクト（つまりメソッドを従えている）。
- ▶ ※ 変数は使い回さないこと（予期せぬ不具合を避けるため）。
- ▶ ※ 定数はサポートされていない。
- ▶ ※ 標準の文字コードは `UTF-8` 。
- ▶ ☆ 命名規則（慣習）
- ▶ ☆ 演算子の優先順位
- ▶ 予約語の一覧（リスト） `import keyword` `keyword.kwlist` ※: 文字列リスト
- ▶ 組み込み関数、クラスの一覧 `dir(__builtin__)` ※: "
- ▶ ※ **スコープはモジュール、関数定義 (`def`) でのみ作られ、制御構文 (`if` `for` `while` `try`) のなかでは作られない。**

■基礎

- ▶ コメントのしかた `#`で行末まで、あるいは `'''` か `"""` で囲めば改行可能。
- ▶ 1 文が長くなるとき 括弧 `()` `{}` `[]` のなかでは自由に改行できる。
また `\` を書いて改行すれば行が継続しているのと同じ。
※長いメソッドチェーンは `()` で挟んで改行しがち
- ▶ ※ 実は、改行の代わりに `;` でも文の終わりになる。しかし、ワンライナー以外では使うべきでない。
- ▶ 変数を定義 `hoge = 値` ※一応 `hoge :型 = 値` で型の明示も可能
- ▶ 複数の変数に同時に代入 `hoge, foga = 値1, 値2` ※これで値の交換もできる
- ▶ 変数が定義済みか `'hoge' in locals()` や `'hoge' in globals()`
- ▶ mainで実行する際は... `if __name__ == '__main__':`
- ▶ 処理をやめる `exit()` # か `import sys sys.exit()` #

▶ 何もしない	
▶ <i>str</i> を式として評価	
▶ <i>str</i> をコードとして実行	
▶ 式のなかで変数に代入	

■標準入出力

▶ コマンドラインでの引数	
▶ 入力	
▶ ☆ Yes/No 入力	
▶ ☆ 数値入力	

▶ 出力	
▶ きれいに出力	
▶ データ型を調べて出力	
▶ 音を鳴らす	<code>print('\007')</code> #
▶ プログレスバー	
▶ 警告をすべて非表示にする	
▶ 関数やクラスの使い方を表示	

■条件分岐

▶ 条件分岐	
▶ 単純なif文の略記	
▶ 比較演算子	
▶ 同じ(IDの)オブか	
▶ 論理演算子	
▶ AND関数・OR関数 的な	
▶ 2 股分岐の略記	

▶ ※ [Tips] <code>x = ○○ if x is None else x</code> は <code>x = x or ○○</code> と書ける。	
▶ bool以外でも評価可能	
▶ Noneかどうか	
▶ 早期リターン	

▶ 何もしない	<code>pass</code> #
▶ <i>str</i> を式として評価	<code>import ast ast.literal_eval(str) か eval(str)</code>
▶ <i>str</i> をコードとして実行	<code>exec(str) ※複数行のコードも可能。</code>
▶ 式のなかで変数に代入	ある式のなかで (変数 := 別の式) ※ 代入式という。

■標準入出力

▶ コマンドラインでの引数	<code>import sys sys.argv[1]</code> ※ <code>\$ python oo.py 第1引数</code>
▶ 入力	<code>変数 = input('名前は?')</code> ※: 文字列型
▶ ☆ Yes/No 入力	
▶ ☆ 数値入力	

▶ 出力	<code>print(式) #</code>
▶ きれいに出力	<code>from pprint import pprint pprint(式) #</code>
▶ データ型を調べて出力	<code>print(type(オブ)) #</code>
▶ 音を鳴らす	<code>print('\007')</code> #
▶ プログレスバー	<code>\$ pip install tqdm from tqdm import tqdm bar = tqdm(iterable) for .. in bar:</code>
▶ 警告をすべて非表示にする	<code>import warnings warnings.simplefilter('ignore') #</code>
▶ 関数やクラスの使い方を表示	<code>help(関数名やクラス名) #</code>

■条件分岐

▶ 条件分岐	<code>if elif else</code>
▶ 単純なif文の略記	<code>if 条件: 1行の処理</code> ※ 改行が要らない！
▶ 比較演算子	<code>== != > < >= <=</code> ※ <code>a > b > c</code> も可能
▶ 同じ(IDの)オブか	<code>obj1 is obj2</code>
▶ 論理演算子	<code>and or not</code> ※ <code>and</code> <code>or</code> はbool以外だと最後の評価式を返す
▶ AND関数・OR関数 的な	<code>all(bool_like_iterable) ・ any(")</code> ※: 最後の評価式bool
▶ 2 股分岐の略記	真での値 <code>if 条件 else 偽での値</code> や 真での処理 <code>if 条件 else 偽での処理 #</code>

▶ ※ [Tips] <code>x = ○○ if x is None else x</code> は <code>x = x or ○○</code> と書ける。	
▶ bool以外でも評価可能	<code>False None 0 0.0</code> 空のコンテナ(" [] など)
▶ Noneかどうか	<code>オブ is None</code>
▶ 早期リターン	<code>if 条件: ↓ continue</code> を一行目書きゃいい。

- ▶ ※ `==` は値が等価かを判定し、`is` はオブジェクトのID番号が同じかを判定する。なお、人間にとって自然な等価判定がされる（たとえば `1 == 1.0` は真になる）。
- ▶ ※ `True` `False` は算術演算子といっしょに使うと `1` `0` の扱いになる。

■繰り返し処理

▶ <i>n</i> 回処理を繰り返す	
▶ foreach文	
▶ 逆順でforeach文	
▶ 添え字も取得してforeach	
▶ 辞書でforeach	
▶ 複数のシーケンスでforeach	
▶ while 文	
▶ 無限ループ	
▶ 中断し、次へ・脱出	
▶ 条件に外れた後の処理	

■関数

▶ 関数を定義	
▶ 戻り値・引数の型を指定	
▶ 値返すだけの即席関数定義	
▶ デフォルト値を設定	
▶ 呼出して仮引数を書かせない	
▶ 呼出して <code>//</code> を必ず書かせる	
▶ 可変長引数を設定	
▶ 辞書型の可変長引数	
▶ 一番外の変数を使う	
▶ 一つ外側の変数を使う	
▶ 関数の呼び出し	
▶ 関数名の文字列で呼び出し	
▶ 引数名を指定して渡す	

- ▶ ※ `==` は値が等価かを判定し、`is` はオブジェクトのID番号が同じかを判定する。なお、人間にとって自然な等価判定がされる（たとえば `1 == 1.0` は真になる）。
- ▶ ※ `True` `False` は算術演算子といっしょに使うと `1` `0` の扱いになる。

■繰り返し処理

▶ <i>n</i> 回処理を繰り返す	for 好きな変数 in range(<i>n</i>): ※ <code>range(stop)</code> は <code>stop</code> の手前まで
▶ foreach文	for i1, ... in iterable:
▶ 逆順でforeach文	... in reversed(sequence):
▶ 添え字も取得してforeach	for idx, i in enumerate(sequence):
▶ 辞書でforeach	for key, i in d.items(): for key in d: for i in d.values():
▶ 複数のシーケンスでforeach	for x1, x2, x3 in zip(s1, s2, s3): ※最少要素数回繰り返す
▶ while 文	while 条件:
▶ 無限ループ	while True: ※中を <code>pass</code> にすれば完全無限ループに
▶ 中断し、次へ・脱出	continue break
▶ 条件に外れた後の処理	else: ※正常終了 (<code>break</code> でない終了) 時に実行される

■関数

▶ 関数を定義	def hoge_hoge(p1, ...): ... return 値
▶ 戻り値・引数の型を指定	def hoge(p1: 型1, ...) -> 戻り値の型名:
▶ 値返すだけの即席関数定義	lambda p1, ...: 引数を使った式
▶ デフォルト値を設定	def hoge(p1, p2=80): ※必ず後へ
▶ 呼出して仮引数を書かせない	def hoge(p1, /, p2=80): ※ <code>/</code> 以前の引数に適用される
▶ 呼出して <code>//</code> を必ず書かせる	def hoge(p1, *, p2=80): ※ <code>*</code> 以降の引数に適用される
▶ 可変長引数を設定	def hoge(p1, p2, *p3): ※極力後へ
▶ 辞書型の可変長引数	def hoge(p1, p2, **p3): ※極力後へ
▶ 一番外の変数を使う	global 変数名 # （変数への代入から始まるなら必須）
▶ 一つ外側の変数を使う	nonlocal 変数名 #
▶ 関数の呼び出し	hoge() hoge(<i>arg1</i> , ...)
▶ 関数名の文字列で呼び出し	eval('hoge')() か eval('hoge()')
▶ 引数名を指定して渡す	hoge(p1= <i>arg1</i> , p2= <i>arg2</i>)

- ▶ イテオブの要素全部を渡す
- ▶ ※ ドキュメンテーション文字列（docstring）を冒頭に書くようにしよう。
- ▶ ※ 関数名が衝突すると上書きされる。組み込み関数にさえも上書きできる。
- ▶ ☆ ある関数に機能を追加して上書きする（デコレータ）
- ▶ ※ 関数内で定義された変数はその関数内でしか使えない。
- ▶ ※ もちろんリスト等も返せる。
- ▶ ☆ 関数に関数を渡す：高階関数
- ▶ ☆ 初めて実行されたときだけ別の挙動をする関数

■例外処理

- ▶ エラーを防ぐべく確認する
- ▶ ☆ 例外オブジェクトの型
- ▶ わざと例外を投げる
- ▶ 例外処理を始める
- ▶ ある例外の発生時に限り処理
- ▶ 例外の発生時に限り処理
- ▶ 正常終了時に限り処理
- ▶ 異常正常によらず最後に処理
- ▶ ※ tryブロック中の変数をexceptブロックで参照したい → tryの外で 変数名 = None
- ▶ ※ 当然、tryの中でエラーが発生してもエラーメッセージは出力されない。
- ▶ ☆ tryでのエラーのスタックトレース（エラーメッセージ）を外部ファイルに書き出し

■よくやる間違い

- ▶ ※ 真偽値は大文字から始まり、**True** と **False** である
- ▶ ※ インデント直前の行でコロンを忘れるな
- ▶ ※ リスト[0] = 新要素 では追加できない。リスト.append(新要素)
- ▶ ※ 辞書の繰り返しは d.items() であり enumerate() ではない！
- ▶ ※ 関数外の変数をつかうときに global 変数 を忘れがち。

オブジェクト編

- ▶ イテオブの要素全部を渡す hoge(..., *iterable, ...)
- ▶ ※ ドキュメンテーション文字列（docstring）を冒頭に書くようにしよう。
- ▶ ※ 関数名が衝突すると上書きされる。組み込み関数にさえも上書きできる。
- ▶ ☆ ある関数に機能を追加して上書きする（デコレータ）
- ▶ ※ 関数内で定義された変数はその関数内でしか使えない。
- ▶ ※ もちろんリスト等も返せる。
- ▶ ☆ 関数に関数を渡す：高階関数
- ▶ ☆ 初めて実行されたときだけ別の挙動をする関数

■例外処理

- ▶ エラーを防ぐべく確認する assert 真偽値, 'エラーメッセージ' # ※第2引数省略可
- ▶ ☆ 例外オブジェクトの型
- ▶ わざと例外を投げる raise 例外型名 ※ raise 例外型名(メッセージ) も可
- ▶ 例外処理を始める try:
- ▶ ある例外の発生時に限り処理 except 例外型名 as e:
- ▶ 例外の発生時に限り処理 except:
- ▶ 正常終了時に限り処理 else:
- ▶ 異常正常によらず最後に処理 finally: ※ except や else より後に処理される
- ▶ ※ tryブロック中の変数をexceptブロックで参照したい → tryの外で 変数名 = None
- ▶ ※ 当然、tryの中でエラーが発生してもエラーメッセージは出力されない。
- ▶ ☆ tryでのエラーのスタックトレース（エラーメッセージ）を外部ファイルに書き出し

■よくやる間違い

- ▶ ※ 真偽値は大文字から始まり、**True** と **False** である
- ▶ ※ インデント直前の行でコロンを忘れるな
- ▶ ※ リスト[0] = 新要素 では追加できない。リスト.append(新要素)
- ▶ ※ 辞書の繰り返しは d.items() であり enumerate() ではない！
- ▶ ※ 関数外の変数をつかうときに global 変数 を忘れがち。

オブジェクト編

■データ型（組み込み）

- int型
 - float型
 - complex型 複素数 `c = 3 + 4j` `c = 3 + 1j` `c = 4j` `c = 3 + 0j` `c = complex(3, 4)`
 - bool型
 - list型
 - tuple型
 - set型
 - dict型
 - range型
 - str型
 - bytes型 bを付けて `' '` `" "` `' '' ''` `"" ""` で囲めばこの型に (例: `b'hello'`)
 - function型 ※型ヒントで使う場合は `from typing import Callable` して `Callable` 。
 - module型
 - NoneType型 None の型
- ▶ ※ ミュータブル : リスト、集合、辞書 （ほぼこれらだけ）
イミュータブル: 文字列、タプル、数値 など

■オブジェクト一般

- ▶ オブのID番号
- ▶ オブのデータ型 (文字列で)
- ▶ オブのデータ型を判定
- ▶ その型のアトリビュート一覧

■数値

- ▶ 2 8 16進数を表現
- ▶ 2 8 16進数表記に
- ▶ カンマ書きたい！
- ▶ 算術演算子
- ▶ 複合代入演算子
- ▶ ビット演算子
- ▶ 文字列を数値に変換

■データ型（組み込み）

- int型
 - float型
 - complex型 複素数 `c = 3 + 4j` `c = 3 + 1j` `c = 4j` `c = 3 + 0j` `c = complex(3, 4)`
 - bool型
 - list型
 - tuple型
 - set型
 - dict型
 - range型
 - str型
 - bytes型 bを付けて `' '` `" "` `' '' ''` `"" ""` で囲めばこの型に (例: `b'hello'`)
 - function型 ※型ヒントで使う場合は `from typing import Callable` して `Callable` 。
 - module型
 - NoneType型 None の型
- ▶ ※ ミュータブル : リスト、集合、辞書 （ほぼこれらだけ）
イミュータブル: 文字列、タプル、数値 など

■オブジェクト一般

- ▶ オブのID番号 id(オブ)
- ▶ オブのデータ型 (文字列で) type(オブ).__name__ （か オブ.__class__.__name__ ）
- ▶ オブのデータ型を判定 isinstance(オブ, 型) や isinstance(オブ, (型1, 型2, ...))
- ▶ その型のアトリビュート一覧 dir(オブ)

■数値

- ▶ 2 8 16進数を表現 数値の先頭に 0b 0o 0x をつける 例) `0b10111001`
- ▶ 2 8 16進数表記に bin(x) oct(x) hex(x) ※: **str型**(先頭に0b等がつく)
- ▶ カンマ書きたい！ 数値の途中に _ ならつけられる 例) `2_000_000`
- ▶ 算術演算子 + - * / // % ** ※多重代入 `a = b = 3` 可能。
- ▶ 複合代入演算子 x += 1 x -= 1 x *= 2 x /= 2
- ▶ ビット演算子 | ^ & << >> ~ ※: 数値
- ▶ 文字列を数値に変換 int(str※) float(str※) ※細かいフォーマット制限がある

▶ 0埋め	
▶ 四捨五入	
▶ 絶対値	
▶ 実部・虚部	
▶ 共役な複素数	
▶ 無限大を生成	
▶ 非数を生成	

■Sizedなオブジェクト全般

▶ Sizedなオブジェクトとは	
▶ ※ Sizedなオブジェクト：リスト、タプル、辞書、集合、文字列	
▶ 何かSizedなオブかどうか	
▶ 要素数	

■コンテナ全般

▶ コンテナとは	
▶ ※ コンテナ：文字列、リスト、タプル、辞書、集合	
▶ ※ コンテナの定義はあいまいらしい。	
▶ 何かコンテナかどうか	
▶ ある要素が含まれるか	
▶ ある要素が含まれないか	
▶ 複数の要素が含まれるか	

■イテラブルオブジェクト全般

▶ イテラブルオブジェクトとは	
▶ ※ イテラブルである：文字列、リスト、タプル、辞書、集合、frozensetオブ、rangeオブ、mapオブ、zipオブ、enumerateオブ イテラブルでない：数値	
▶ 何かイテオブかどうか	
▶ 全要素に関数を適用	
▶ ※ <code>map()</code> の関数にはlamdaも使えるが、それをやるなら内法表記のほうが速い。	
▶ 内法表記	

▶ 0埋め	'{数値:0桁数}だよ' か format(数値, '0桁数')
▶ 四捨五入	round(数値, 小数点以下桁数※) ※略せば0に
▶ 絶対値	abs(数値) ※ 複素数型にも対応
▶ 実部・虚部	複素数.real・複素数.imag ※ 取得のみ
▶ 共役な複素数	複素数.conjugate()
▶ 無限大を生成	float('inf') か import math math.inf
▶ 非数を生成	float('nan') か import math math.nan

■Sizedなオブジェクト全般

▶ Sizedなオブジェクトとは	len()で要素数を返すオブジェクト
▶ ※ Sizedなオブジェクト：リスト、タプル、辞書、集合、文字列	
▶ 何かSizedなオブかどうか	import collections.abc as ca isinstance(式, ca.Sized)
▶ 要素数	len(sized)

■コンテナ全般

▶ コンテナとは	in演算子を利用できるオブジェクト
▶ ※ コンテナ：文字列、リスト、タプル、辞書、集合	
▶ ※ コンテナの定義はあいまいらしい。	
▶ 何かコンテナかどうか	import collections.abc as ca isinstance(式, ca.Container)
▶ ある要素が含まれるか	x in <i>container</i>
▶ ある要素が含まれないか	x not in <i>container</i> か not x in <i>container</i>
▶ 複数の要素が含まれるか	all(x in <i>container</i> for x in (x1, x2, ...))

■イテラブルオブジェクト全般

▶ イテラブルオブジェクトとは	for演算子で繰り返し可能なもの
▶ ※ イテラブルである：文字列、リスト、タプル、辞書、集合、frozensetオブ、rangeオブ、mapオブ、zipオブ、enumerateオブ イテラブルでない：数値	
▶ 何かイテオブかどうか	import collections.abc as ca isinstance(式, ca.Iterable)
▶ 全要素に関数を適用	<i>convFunc</i> ※(map(<i>func</i> , <i>iterable</i>)) ※ <code>list</code> <code>''.join</code> <code>set</code> 等
▶ ※ <code>map()</code> の関数にはlamdaも使えるが、それをやるなら内法表記のほうが速い。	
▶ 内法表記	一般項 for i in イテオブ 一般項 for i in イテオブ if 条件式

▶ ※ 内法表記は括弧をつけることで成り立つ。ただ、リスト内法表記、辞書内法表記、集合内法表記はあるが、タプル内法表記はなく、`()` をつけるとジェネレータ式になる。

▶ ※ 内法表記は、一般項の部分が1つの**組み込み関数**による単純な処理の場合、`map()` のほうが速い。

▶ ※ forを複数使う内法表記について、**より外側のループに対応する `for` を先に書く**こと。

▶ リスト・タプル・集合に

▶ 昇・降順に並び替えてリストに

▶ 関数の結果で並び替えてリストに

▶ 関数の結果でフィルター

▶ 最大値・最小値

▶ 合計

▶ ※ 上記の *iterable* (イテオブ) はイテレータに置き換えることも可能。`convFunc()` を用いているものに関しては、`convFunc()` を `iter(list())` にすればよい。

イテレータについて

▶ イテレータとは

▶ ☆ classによるイテレータの実装例

▶ イテオブからイテレータを作る

▶ イテレータの現在の値を取得しつつ次へ進める

▶ ジェネレータとは

▶ ※ `return` ではなく `yield` を使えば即ち、ジェネレータを実装したことになる。

▶ ※ `yield` した回数だけ値が出てくる。

▶ ※ 関数が呼び出されたとき、実行されるのは `yield` のところまで。そして、2回目以降の関数呼び出しの際には、前回の `yield` の次の行から実行される。

▶ ※ `変数 = yield 値` とすることで、返すのと同時に変数に代入もできる。

▶ ※ `yield` 以前にある変数に値が代入されているとき、以降の `yield` でもその変数は値を保持している。

▶ ※ ジェネレータ関数を関数呼び出しするとイテレータオブジェクトになる。

▶ ☆ ジェネレータの実装例1

一般項 **if 条件式 else 式** for i in イテオブ

一般項 for i1 in イテオブ₁ for i2 in イテオブ₂ ...

▶ ※ 内法表記は括弧をつけることで成り立つ。ただ、リスト内法表記、辞書内法表記、集合内法表記はあるが、タプル内法表記はなく、`()` をつけるとジェネレータ式になる。

▶ ※ 内法表記は、一般項の部分が1つの**組み込み関数**による単純な処理の場合、`map()` のほうが速い。

▶ ※ forを複数使う内法表記について、**より外側のループに対応する `for` を先に書く**こと。

▶ リスト・タプル・集合に `list(iterable)` ・ `tuple(iterable)` ・ `set(iterable)`

▶ 昇・降順に並び替えてリストに `sorted(iterable)` ・ `sorted(iterable, reverse=True)`

▶ 関数の結果で並び替えてリストに `sorted(iterable, key=func)`

▶ 関数の結果でフィルター `convFunc(filter(func, iterable))`

▶ 最大値・最小値 `max(iterable)` ・ `min(iterable)`

▶ 合計 `sum(iterable)`

▶ ※ 上記の *iterable* (イテオブ) はイテレータに置き換えることも可能。`convFunc()` を用いているものに関しては、`convFunc()` を `iter(list())` にすればよい。

イテレータについて

▶ イテレータとは イテラブルオブジェクトを操作するためのオブジェクト

▶ ☆ classによるイテレータの実装例

▶ イテオブからイテレータを作る `i = iter(collection)`

▶ イテレータの現在の値を取得しつつ次へ進める `変数 = next(i)`

▶ ジェネレータとは イテレータの一種で、要素を取り出そうとするごとに処理をおこない、その都度、要素を生成するもの

▶ ※ `return` ではなく `yield` を使えば即ち、ジェネレータを実装したことになる。

▶ ※ `yield` した回数だけ値が出てくる。

▶ ※ 関数が呼び出されたとき、実行されるのは `yield` のところまで。そして、2回目以降の関数呼び出しの際には、前回の `yield` の次の行から実行される。

▶ ※ `変数 = yield 値` とすることで、返すのと同時に変数に代入もできる。

▶ ※ `yield` 以前にある変数に値が代入されているとき、以降の `yield` でもその変数は値を保持している。

▶ ※ ジェネレータ関数を関数呼び出しするとイテレータオブジェクトになる。

▶ ☆ ジェネレータの実装例1

- ▶ ☆ ジェネレータの実装例2
- ▶ ☆ ジェネレータの実装例3
- ▶ ※ 上の実装例のような複雑な計算を行うジェネレータはdefを使わないと厳しいが、シンプルなジェネレータならジェネレータ式（ `()` で囲んだ内法表記）によって作成できる。

■コレクション全般

- ▶ コレクションとは
- ▶ ※ コレクションである：文字列、リスト、タプル、辞書、集合
コレクションでない：数値
- ▶ 何かコレクションかどうか

シーケンス全般

- ▶ シーケンスとは
- ▶ ※ シーケンスである：リスト、タプル、文字列、bytesオブ、rangeオブ
シーケンスでない：辞書、集合
- ▶ 何かシーケンスかどうか
- ▶ 1つの要素を参照
- ▶ スライス
- ▶ ある値が何番目に初登場か
- ▶ ある値の要素をいくつ含むか
- ▶ 順番を維持して重複なくす
- ▶ 逆順にする

■文字列

- ▶ 特殊な文字を表現
- ▶ 複数行にわたる文字列
- ▶ ☆ 長い文字列をコード上で複数行にわけて記述して表現
- ▶ 何かを文字列に変換
- ▶ 文字列の結合
- ▶ 変数展開
- ▶ 文字列の反復
- ▶ 途中の文字を取得
- ▶ 途中の複数の文字を取得

- ▶ ☆ ジェネレータの実装例2
- ▶ ☆ ジェネレータの実装例3
- ▶ ※ 上の実装例のような複雑な計算を行うジェネレータはdefを使わないと厳しいが、シンプルなジェネレータならジェネレータ式（ `()` で囲んだ内法表記）によって作成できる。

■コレクション全般

- ▶ コレクションとは Sized かつ イテラブル かつ コンテナ なオブジェクト
- ▶ ※ コレクションである：文字列、リスト、タプル、辞書、集合
コレクションでない：数値
- ▶ 何かコレクションかどうか `import collections.abc as ca isinstance(式, ca.Collection)`

シーケンス全般

- ▶ シーケンスとは インデックスを用いて要素を指定できるコレクション
- ▶ ※ シーケンスである：リスト、タプル、文字列、bytesオブ、rangeオブ
シーケンスでない：辞書、集合
- ▶ 何かシーケンスかどうか `import collections.abc as ca isinstance(式, ca.Sequence)`
- ▶ 1つの要素を参照 `s[n]` ※*n*は負も可
- ▶ スライス `s[n:m]` `s[n:]` `s[:m]` `s[n:m:step]` ※*m*はその手前まで
- ▶ ある値が何番目に初登場か `s.index(value)` ※: int; ValueError
- ▶ ある値の要素をいくつ含むか `s.count(value)`
- ▶ 順番を維持して重複なくす `convFunc`(dict.fromkeys(s)) ※ `list` `''.join` 等
- ▶ 逆順にする `s[::-1]` か `convFunc(reversed(s))`

■文字列

- ▶ 特殊な文字を表現 `\n \\ \' \t`
- ▶ 複数行にわたる文字列 `'''` か `"""` で囲んでコード上で改行
- ▶ ☆ 長い文字列をコード上で複数行にわけて記述して表現
- ▶ 何かを文字列に変換 `str(式)`
- ▶ 文字列の結合 `+` ※数値とは結合できない→ `str(str)` か 変数展開
- ▶ 変数展開 `f'Hello, {name}'` ほか ※ `"` でも。また `F` でも。
- ▶ 文字列の反復 `str * n`
- ▶ 途中の文字を取得 `s[n]`
- ▶ 途中の複数の文字を取得 スライスで

▶ 末尾に要素を追加	
▶ イテオブの全要素を追加	
▶ 好きな位置に要素を挿入	
▶ 連結	
▶ 要素を削除	
▶ 末尾の要素を削除	
▶ すべての要素を削除	
▶ 昇順に並び替え 降順に並び替え	
▶ 関数の結果で並び替え	
▶ 逆順にする	
▶ 順番を維持して重複なくす	
▶ リストをコピー・深いコピー	
▶ 2次元リストを転置する	
▶ 関数の結果でフィルター	
▶ ☆ リストを分割 (Split) する	
▶ ☆ 要素の出現回数のランキング	
▶ ※ 'generator' object is not subscriptable と出たら → リスト化してから処理。	

デック (deque)

- ▶ ※ デックとは両端キューのことで、スタックとキューの両方の機能をもっている。シーケンスのどちらの端でも要素を追加、削除できるようにしたいときに便利。

▶ デックを使う準備	
▶ デックを作成	
▶ 空のデック	
▶ 先頭・末尾に要素を追加	
▶ 先頭・末尾の要素を削除	

■辞書

▶ 辞書を作成	
▶ 空の辞書	

▶ 末尾に要素を追加	<code>l.append(x) #</code>
▶ イテオブの全要素を追加	<code>l.extend(iterable) #</code> ※リストなら <code>1 += 12</code> でも
▶ 好きな位置に要素を挿入	<code>l.insert(n, x) #</code>
▶ 連結	<code>l = [*l1, *l2] l1.extend(l2) # l = l1 + l2</code>
▶ 要素を削除	<code>del l[n] # 変数 = l.pop(n) del [n:m] # l[n:m] = []</code> <code>l.remove(x) #</code> のどれか
▶ 末尾の要素を削除	<code>l.pop() #</code> ※削除した値を返す
▶ すべての要素を削除	<code>l.clear() #</code> か <code>l[:] = []</code>
▶ 昇順に並び替え 降順に並び替え	<code>l.sort() #</code> か <code>sorted(l)</code> <code>l.sort(reverse=True) #</code> か <code>sorted(l, reverse=True)</code>
▶ 関数の結果で並び替え	<code>l.sort(key=func) #</code> か <code>sorted(l, key=func)</code>
▶ 逆順にする	<code>l.reversed() #</code> か <code>list(reversed(l))</code>
▶ 順番を維持して重複なくす	<code>list(dict.fromkeys(l))</code>
▶ リストをコピー・深いコピー	<code>import copy l2 = copy.copy(l1) # copy.deepcopy(l1)</code>
▶ 2次元リストを転置する	<code>a_t = list(zip(*a))</code>
▶ 関数の結果でフィルター	<code>list(filter(func, l))</code>
▶ ☆ リストを分割 (Split) する	
▶ ☆ 要素の出現回数のランキング	
▶ ※ 'generator' object is not subscriptable と出たら → リスト化してから処理。	

デック (deque)

- ▶ ※ デックとは両端キューのことで、スタックとキューの両方の機能をもっている。シーケンスのどちらの端でも要素を追加、削除できるようにしたいときに便利。

▶ デックを使う準備	<code>from collections import deque</code>
▶ デックを作成	<code>dq = deque(iterable)</code>
▶ 空のデック	<code>deque()</code>
▶ 先頭・末尾に要素を追加	<code>dq.appendleft(x) #</code> ・ <code>dq.append(x) #</code>
▶ 先頭・末尾の要素を削除	<code>dq.popleft() #</code> ・ <code>dq.pop() #</code> ※削除した値を返す

■辞書

▶ 辞書を作成	<code>d = {'abc': 2, ('d', 'e',): True}</code> や <code>dict(abc=2, def=True)</code>
▶ 空の辞書	<code>{}</code> か <code>dict()</code>

▶ ※ キーには設定できるのはイミュータブルオブジェクトだけ。	
▶ 2つのシーケンスから辞書に	
▶ 要素を参照	
▶ 要素を追加、上書き	
▶ 存在しない時に限り要素を追加	
▶ 要素を削除	
▶ 値orキーを取り出してリストに	
▶ 値を合計する	
▶ 複数の辞書を結合	
▶ キーがすでにあるか	

defaultdict

▶ デ辞書を使う準備	
▶ デ辞書を作成	
▶ 要素を参照	
▶ 要素を上書き	

■タプル

▶ タプルを作成	
▶ 空のタプル	
▶ 要素数 1 のタプル	
▶ 要素(1つ,複数)を参照	

名前付きタプル

▶ ※ 型ヒントを使うには継承するしかない。それをするなら構造体のほうが適する。	
▶ 名タブを使う準備	
▶ 名タブを定義	
▶ 名タブを作成	
▶ リストを名タブに	
▶ 辞書を名タブに	
▶ ある属性値を取得	

▶ ※ キーには設定できるのはイミュータブルオブジェクトだけ。	
▶ 2つのシーケンスから辞書に	d = dict(zip(keys, values))
▶ 要素を参照	・ d[key] ※無ければエラー ※ d[n] は不可 ・ d.get(key, defalut) ※無ければ default に
▶ 要素を追加、上書き	d[key] = value
▶ 存在しない時に限り要素を追加	d.setdefault(key, value) ※: 値 ※上書きはされない
▶ 要素を削除	del d[key]
▶ 値orキーを取り出してリストに	list(d.values()) list(d.keys())
▶ 値を合計する	sum(d.values())
▶ 複数の辞書を結合	d = {**d1, **d2}
▶ キーがすでにあるか	key in d

defaultdict

▶ デ辞書を使う準備	from collections import defaultdict
▶ デ辞書を作成	dd = defaultdict(初期値を返す関数)
▶ 要素を参照	・ d[key] ※上書きがまだなら初期値に ・ d.get(key, defalut) ※ " なら default に
▶ 要素を上書き	dd[key] = value ※初期値があるから += など使える

■タプル

▶ タプルを作成	t = (80, 90, 40)
▶ 空のタプル	() か tuple()
▶ 要素数 1 のタプル	('abc',)
▶ 要素(1つ,複数)を参照	t[n] t[n:m] t[n:] t[:m] t[n:m:step]

名前付きタプル

▶ ※ 型ヒントを使うには継承するしかない。それをするなら構造体のほうが適する。	
▶ 名タブを使う準備	from collections import namedtuple
▶ 名タブを定義	Foo = namedtuple('Foo', ['field1', 'field2', ...])
▶ 名タブを作成	foo = Foo(value1, ...) や Foo(field1=value1, ...)
▶ リストを名タブに	Foo(*l) か Foo._make(l)
▶ 辞書を名タブに	Foo(**d)
▶ ある属性値を取得	foo.field2

▶ ある属性値を上書き	
▶ ※ 属性の追加はできない。	
▶ 属性名たちを取得	
▶ 名タブを辞書に	

■集合	
▶ 集合を作成	
▶ 空の集合	
▶ ※ 要素にできるのは、イミュータブルな型のものだけ（つまりリスト、辞書、集合は要素にできない）。	
▶ 要素を追加	
▶ 要素を削除	
▶ すべての要素を削除	
▶ 集合演算	
▶ 他の集合と一致するか	
▶ " を包含するか	
▶ " の部分集合か	
▶ " と互いに素か	

Frozen Set（凍結集合）

▶ ※ Frozen Set とは変更不可能な集合のこと。要素の変更以外は集合と同じ操作ができる。	
▶ Frozen Set を作成	

■fileオブジェクト (stream)	
▶ io モ	

テキストストリーム

▶ テキストストリームを作成	
▶ パスを指定して "	
▶ エンコーディング指定しつつ "	
▶ ※ <code>UnicodeDecodeError: ~ codec can't decode byte ~</code> と出たら → エンコーディングを指定	
▶ エンコーディング	

▶ ある属性値を上書き	<code>foo = foo._replace(fieald2 = <i>newValue2</i>, ...)</code>
▶ ※ 属性の追加はできない。	
▶ 属性名たちを取得	<code>foo._fields</code>
▶ 名タブを辞書に	<code>foo._asdict()</code>

■集合	
▶ 集合を作成	<code>s = {1, 2, 3}</code>
▶ 空の集合	<code>set()</code>
▶ ※ 要素にできるのは、イミュータブルな型のものだけ（つまりリスト、辞書、集合は要素にできない）。	
▶ 要素を追加	<code>s.add(新要素) #</code>
▶ 要素を削除	<code>s.remove(値) か s.discard(値) ※後者は不存でも非エラー</code>
▶ すべての要素を削除	<code>s.clear()</code>
▶ 集合演算	<code>s1 s2 s1 & s2 s1 - s2 s1 ^ s2</code>
▶ 他の集合と一致するか	<code>s1 == s2 ※ s1 != s2 も可能。</code>
▶ " を包含するか	<code>s1 >= s2 ※ <code>s1 > s2</code> にすると等しい場合Falseに</code>
▶ " の部分集合か	<code>s1 <= s2 ※ <code>s1 < s2</code> にすると "</code>
▶ " と互いに素か	<code>s1.isdisjoint(s2)</code>

Frozen Set（凍結集合）

▶ ※ Frozen Set とは変更不可能な集合のこと。要素の変更以外は集合と同じ操作ができる。	
▶ Frozen Set を作成	<code>fs = frozenset(<i>iterable</i>)</code>

■fileオブジェクト (stream)	
▶ io モ	<code>import io</code>

テキストストリーム

▶ テキストストリームを作成	<code>f = io.StringIO()</code> ※ <code>f</code> でなく <code>buf</code> とされることが多い
▶ パスを指定して "	<code>・ f = open(パス, 'r')※ ※適宜 'a' 'w' に</code> <code>・ with open(パス1, 'r') as f1, open(パス2, 'r') as f2, ...:</code>
▶ エンコーディング指定しつつ "	<code>f = open(・・, encoding='エンコーディング')</code>
▶ ※ <code>UnicodeDecodeError: ~ codec can't decode byte ~</code> と出たら → エンコーディングを指定	
▶ エンコーディング	<code>f.encoding</code>

バイナリストリーム

- ▶ バイナリストリームを作成
- ▶ パスを指定して "

テキストストリーム・バイナリストリーム共通

- ▶ 改行コードを修整しつつ
 パスを指定してスト作成
- ▶ 現在のストリーム位置
- ▶ ストリーム位置を変更
- ▶ 現在地からEOFまで読み込み
- ▶ " 改行かEOFまで読み込み
- ▶ ※ f.getvalue() で現在地に関係なく全てを読み込める。ただし、open() で得たストリームには効かない。
- ▶ ※ open() で得たストリームに限り、f.name でパス、f.mode でモードを取得できる。
- ▶ 1 行ずつ順に読んで処理
- ▶ " 読んでリストに
- ▶ 書き込む
- ▶ ※ 読みみや書込みを行ったあとに f をバッファとして何かの関数やメソッドに渡すとき、必ず f.seek(0) で位置を最初に戻してから渡すこと！！
- ▶ フラッシュして閉じる

■その他のオブジェクト

キュー、スタック、優先度付きキュー

- ▶ ※ キューオブジェクト（キューおよびスタック、優先度付きキュー）は Sized でも イテラブル でも コンテナ でもない。
- ▶ queue モ
- ▶ 空のFIFOキュー
- ▶ 空のスタック
- ▶ 空の優先度付きキュー

バイナリストリーム

- ▶ バイナリストリームを作成 f = io.BytesIO() ※ f でなく buf とされることが多い
- ▶ パスを指定して "・ f = open(パス, 'rb'※) ※適宜 'ab' 'wb' に
 ・ with open(パス1, 'rb'※) as f1, open(パス2, 'rb'※) as f2, ...:

テキストストリーム・バイナリストリーム共通

- ▶ 改行コードを修整しつつ f = open(・, newline=newline)
 パスを指定してスト作成 ※ newLine には None , ' ' , '\n' , '\r' , '\r\n' 。
- ▶ 現在のストリーム位置 f.tell()
- ▶ ストリーム位置を変更 f.seek(offset) # ※先頭からoffset番目の位置に移動する
 ※第2引数で 1 とすると現在地から。 2 なら末尾から。
- ▶ 現在地からEOFまで読み込み f.read() ※ f.read(n) とすると最大n文字/nバイトまで
- ▶ " 改行かEOFまで読み込み f.readline()
- ▶ ※ f.getvalue() で現在地に関係なく全てを読み込める。ただし、open() で得たストリームには効かない。
- ▶ ※ open() で得たストリームに限り、f.name でパス、f.mode でモードを取得できる。
- ▶ 1 行ずつ順に読んで処理 for line in f:␣ ⋯ ※ line には改行コードまで入る
- ▶ " 読んでリストに data = f.readlines() data[行番号] ※ 要素末尾が \n にか
 か data = f.read().splitlines() " ※末尾は \n でない
- ▶ 書き込む f.write(x) #
- ▶ ※ 読みみや書込みを行ったあとに f をバッファとして何かの関数やメソッドに渡すとき、必ず f.seek(0) で位置を最初に戻してから渡すこと！！
- ▶ フラッシュして閉じる f.close() #

■その他のオブジェクト

キュー、スタック、優先度付きキュー

- ▶ ※ キューオブジェクト（キューおよびスタック、優先度付きキュー）は Sized でも イテラブル でも コンテナ でもない。
- ▶ queue モ import queue
- ▶ 空のFIFOキュー queue.Queue() ※引数 maxsize で最大要素数を指定可能
- ▶ 空のスタック queue.LifoQueue() ※ "
- ▶ 空の優先度付きキュー queue.PriorityQueue() ※ "

▶ 要素を 1 つ追加	
▶ 要素を 1 つ取り出す (取得しつつ削除する)	
▶ 空か・満杯かどうか	
▶ 現在の要素数	

- ▶ ※ こいつら、ただのデータコレクションじゃねえ...！（マルチスレッドプログラミングにおいて複数スレッド間で情報を安全に交換できるように設計されている！）
- ▶ 全要素のタスクが完了するまで待つ
- ▶ 1 要素のタスク完了をキューオブに伝える

dataclass

- ▶ ※ 名前付きタプルに似ているが、型ヒントを使えるところが違う。構造体に似ている。
- ▶ ※ dataclass は Sized でも イテラブル でも コンテナ でもない。
- ▶ dataclassを使う準備
- ▶ dataclassを定義
- ▶ 属性を設定

クラス編

■クラスの用語とその役割

▶ インスタンスメソッド	
▶ クラスメソッド	
▶ 静的メソッド	
▶ インスタンス変数	
▶ クラス変数	

■クラスの定義

▶ クラスを定義	
▶ インスタンスメソッド	
▶ コンストラクタ	

▶ 要素を 1 つ追加	追加できないならできるまで待つ： <code>.put(item) #</code> ＃ならエラー： <code>.put(item, block=False) #</code> か <code>.put_nowait(item) #</code> ＃なら <i>t</i> 秒まで待ってからエラー： <code>.put(item, timeout=t)</code>
▶ 要素を 1 つ取り出す (取得しつつ削除する)	取り出せないなら取り出せるまで待つ： <code>.get()</code> ＃ならエラー： <code>.get(block=False)</code> か <code>get_nowait()</code> ＃なら <i>t</i> 秒まで待ってからエラー： <code>.get(timeout=t)</code>
▶ 空か・満杯かどうか	<code>.empty()</code> ・ <code>.full()</code>
▶ 現在の要素数	<code>.qsize()</code>

- ▶ ※ こいつら、ただのデータコレクションじゃねえ...！（マルチスレッドプログラミングにおいて複数スレッド間で情報を安全に交換できるように設計されている！）
- ▶ 全要素のタスクが完了するまで待つ `.join()` ※ 普通メインスレにて
- ▶ 1 要素のタスク完了をキューオブに伝える `.task_done()` ※ 普通ワーカースレにて

dataclass

- ▶ ※ 名前付きタプルに似ているが、型ヒントを使えるところが違う。構造体に似ている。
- ▶ ※ dataclass は Sized でも イテラブル でも コンテナ でもない。
- ▶ dataclassを使う準備 `from dataclasses import dataclass`
- ▶ dataclassを定義 `@dataclass` `class Foo:` `...`
- ▶ 属性を設定 `hoge: 型` や `hoge: 型 = 初期値`

クラス編

■クラスの用語とその役割

▶ インスタンスメソッド	汎用化してできたオブをレシーバにして呼び出せるメソ
▶ クラスメソッド	クラスをレシーバにして呼び出せるメソ
▶ 静的メソッド	ただの関数と同じ（しかし何かの方針でクラスに属させたい）
▶ インスタンス変数	汎用化してできたオブごとに管理される変数
▶ クラス変数	クラスごとに管理される変数

■クラスの定義

▶ クラスを定義	<code>class Foo:</code> <code>...</code> ※ <code>Foo()</code> と括弧があってもいい
▶ インスタンスメソッド	<code>def hoge_hoge(self, p1, ...):</code> <code>...</code>
▶ コンストラクタ	<code>def __init__(self, p1, ...):</code> <code>...</code>

- ▶ インスタンス変数
- ▶ 読取のみ可能な `__` (正確には違うが!)
- ▶ 書込のみ可能な `__` (正確には違うが!)
- ▶ クラスメソッド
- ▶ インスタメソ内で `__` 呼ぶ
- ▶ クラス変数
- ▶ インスタメソ内で `__` 参照
- ▶ 静的メソッド
- ▶ ※ `__hoge` のように `__` から始まる名前をつければ、メソッドも変数も (特別な記述をしない限り) クラスの外から呼べない、すなわちプライベートにできる。逆にそうしなければパブリックになる。
- ▶ ☆ dunder メソッド (特殊メソッド)

継承

- ▶ ※ 継承すれば is-a の関係になる。
- ▶ ※ Pythonは多重継承ができる。
- ▶ ※ 作成したクラスは必ず Object クラスを継承している。
- ▶ クラスを継承
- ▶ インスタメソ内で親の同名のそれを呼ぶ

■クラスの操作

- ▶ ☆ 通常メンバ変数、プライベート変数、パブリッククラスメンバ変数、プライベートクラス変数、コンストラクタ、インスタンスメソッド、プライベートメソッド、クラスメソッド、プライベートクラスメソッド

■その他のクラスの定義と操作

列挙型クラス

- ▶ ※ 列挙型とは、複数の定数をひとまとめにして管理する型。
- ▶ 列挙型クラスを定義
- ▶ 定数 (複数) を定義
- ▶ `__` する際に値を連番に

- ▶ インスタンス変数 インスタメソ内で `self.hoge` ※クラス外から読書可能
- ▶ 読取のみ可能な `__` (正確には違うが!)
- ▶ 書込のみ可能な `__` (正確には違うが!)
- ▶ クラスメソッド `@classmethod`
- ▶ インスタメソ内で `__` 呼ぶ `Foo.hoge` か `self.__class__.hoge` ※前者だと継承時困るかもね
- ▶ クラス変数 クラス定義直下またはクラスメソ内で `hoge`
- ▶ インスタメソ内で `__` 参照 `self.hoge` ※selfで参照できちゃう！！！！
- ▶ 静的メソッド `@staticmethod`
- ▶ ※ `__hoge` のように `__` から始まる名前をつければ、メソッドも変数も (特別な記述をしない限り) クラスの外から呼べない、すなわちプライベートにできる。逆にそうしなければパブリックになる。
- ▶ ☆ dunder メソッド (特殊メソッド)

継承

- ▶ ※ 継承すれば is-a の関係になる。
- ▶ ※ Pythonは多重継承ができる。
- ▶ ※ 作成したクラスは必ず Object クラスを継承している。
- ▶ クラスを継承 `class Foo(Bar, Baz, ...):`
- ▶ インスタメソ内で親の同名のそれを呼ぶ 単継なら `super().hoge(arg1, ...)`
多継なら `Baz.hoge(arg1, ...)`

■クラスの操作

- ▶ ☆ 通常メンバ変数、プライベート変数、パブリッククラスメンバ変数、プライベートクラス変数、コンストラクタ、インスタンスメソッド、プライベートメソッド、クラスメソッド、プライベートクラスメソッド

■その他のクラスの定義と操作

列挙型クラス

- ▶ ※ 列挙型とは、複数の定数をひとまとめにして管理する型。
- ▶ 列挙型クラスを定義 `from enum import Enum` `class Foo(Enum):`
- ▶ 定数 (複数) を定義 クラス定義直下で `HOGE = 値1` `FUGA = 値2` ...
- ▶ `__` する際に値を連番に `from enum import auto` ... `HOGE = auto()` ...

- ▶ 定数の名前
- ▶ 定数の値
- ▶ ※ クラス定義の中でも外でも定数は書き換えられない！

抽象クラス

- ▶ ※ 抽象クラスでポリモーフィズム（多相性）を実現できる。
- ▶ 準備
- ▶ 抽象クラスを定義
- ▶ 抽象メソッドを定義

- ▶ ※ 抽象クラスはインスタンスを作成できない。実際に使用するクラスにおいて、抽象クラスを継承して親の抽象メソッドをオーバーライドすることでそのメソッドを動作させることができる。

- ▶ 定数の名前
- ▶ 定数の値
- ▶ ※ クラス定義の中でも外でも定数は書き換えられない！

抽象クラス

- ▶ ※ 抽象クラスでポリモーフィズム（多相性）を実現できる。
- ▶ 準備
- ▶ 抽象クラスを定義
- ▶ 抽象メソッドを定義

- ▶ ※ 抽象クラスはインスタンスを作成できない。実際に使用するクラスにおいて、抽象クラスを継承して親の抽象メソッドをオーバーライドすることでそのメソッドを動作させることができる。