

【Python】基礎

目次

- 環境編
 - 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

【Python】基礎

目次

- 環境編
 - 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

■環境構築（全般）

- ▶ 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

■環境構築（Ubuntu 編）

- ▶ ☆ 1. Pythonをインストール
- ▶ ☆ 2. venvによる仮想環境を新しく作成し、開く
- ▶ ☆ F. 仮想環境を無効にする
- ▶ ☆ R. 再び仮想環境を開く

■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

- クラスの操作
- その他のクラスの定義と操作

環境編

■環境構築（全般）

- ▶ 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}`

■環境構築（Ubuntu 編）

- ▶ ☆ 1. Pythonをインストール
- ▶ ☆ 2. venvによる仮想環境を新しく作成し、開く
- ▶ ☆ F. 仮想環境を無効にする
- ▶ ☆ R. 再び仮想環境を開く

■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
- ▶ 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！
- ▶ 仮想環境を起動
- ▶ ※ 仮想環境を起動してもカレントディレクトリは移動しない！
- ▶ 仮想環境を終了
- ▶ 仮想環境を削除

■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

- ▶ 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！
- ▶ 仮想環境を起動 Win： \$ conda activate 環境 Mac： \$ source activate 環境
- ▶ ※ 仮想環境を起動してもカレントディレクトリは移動しない！
- ▶ 仮想環境を終了 Win： \$ conda deactivate Mac： \$ source 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） import keyword keyword.kwlist ※: 文字列リスト
- ▶ 組み込み関数、クラスの一覧 dir(__builtin__) ※: "

■基礎	
▶ コメントのしかた	
▶ 1文が長くなるとき	
▶ 変数を定義	
▶ 複数の変数に同時に代入	
▶ mainで実行する際は...	
▶ 処理をやめる	
▶ 何もしない	
▶ strを式として評価	
▶ str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	
▶ 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	

■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
▶ 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
▶ 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	pass #
▶ strを式として評価	import ast ast.literal_eval(str) か eval(str)
▶ strをコードとして実行	exec(str) ※複数行のコードも可能。
▶ 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	import sys sys.argv[1] ※ \$ python oo.py 第1引数
▶ 入力	変数 = input('名前は何? ') ※: 文字列型
▶ ☆ Yes/No 入力	
▶ ☆ 数値入力	
▶ 出力	print(式) #
▶ きれいに出力	from pprint import pprint pprint(式) #
▶ データ型を調べて出力	print(type(オブ)) #
▶ 音を鳴らす	print('\007') #
▶ プログレスバー	\$ pip install tqdm from tqdm import tqdm bar = tqdm(iterable) for .. in bar:
▶ 警告をすべて非表示にする	import warnings warnings.simplefilter('ignore') #
▶ 関数やクラスの使い方を表示	help(関数名やクラス名) #

■条件分岐	
▶ 条件分岐	if elif else
▶ 単純なif文の略記	if 条件: 1行の処理 ※ 改行が要らない!
▶ 比較演算子	== != > < >= <= ※ a > b > c も可能

▶ 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	
▶ bool以外でも評価可能	
▶ Noneかどうか	
▶ 早期リターン	
▶ ※ <code>==</code> は値が等価かを判定し、 <code>is</code> 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 <code>1 == 1.0</code> は真になる）。	
▶ ※ <code>True</code> <code>False</code> は算術演算子といっしょに使うと <code>1</code> <code>0</code> 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	
▶ 返り値・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	
▶ 呼出しで仮引数を書かせない	
▶ 呼出しで <code>''</code> を必ず書かせる	

▶ 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)</code> ・ <code>any('')</code> ※: 最後の評価式bool
▶ 2 股分岐の略記	真での値 if 条件 else 偽での値 や 真での処理 if 条件 else 偽での処理 #
▶ bool以外でも評価可能	<code>False</code> <code>None</code> <code>0</code> <code>0.0</code> 空のコンテナ(<code>'' []</code> など)
▶ Noneかどうか	オブ is None
▶ 早期リターン	if 条件: ↓ <code>continue</code> を一行目書きゃいい。
▶ ※ <code>==</code> は値が等価かを判定し、 <code>is</code> 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 <code>1 == 1.0</code> は真になる）。	
▶ ※ <code>True</code> <code>False</code> は算術演算子といっしょに使うと <code>1</code> <code>0</code> の扱いになる。	

■繰り返し処理

▶ <i>n</i> 回処理を繰り返す	<code>for</code> 好きな変数 in <code>range(n)</code> : ※ <code>range(stop)</code> は <code>stop</code> の手前まで
▶ foreach文	<code>for</code> i1, ... in <i>iterable</i> :
▶ 逆順でforeach文	·· in <code>reversed(sequence)</code> :
▶ 添え字も取得してforeach	<code>for</code> idx, i in <code>enumerate(sequence)</code> :
▶ 辞書でforeach	<code>for</code> key, i in <code>d.items()</code> : <code>for</code> key in <code>d</code> : <code>for</code> i in <code>d.values()</code> :
▶ 複数のシーケンスでforeach	<code>for</code> x1, x2, x3 in <code>zip(s1, s2, s3)</code> : ※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	<code>continue</code> ・ <code>break</code>
▶ 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	<code>def</code> hoge_hoge(p1, ...): ↓ ··· ↓ <code>return</code> 値
▶ 返り値・引数の型を指定	<code>def</code> hoge(p1: 型1, ...) -> 返り値の型名:
▶ 値返すだけの即席関数定義	<code>lambda</code> p1, ...: 引数を使った式
▶ デフォルト値を設定	<code>def</code> hoge(p1, p2=80): ※必ず後へ
▶ 呼出しで仮引数を書かせない	<code>def</code> hoge(p1, /, p2=80): ※ <code>/</code> 以前の引数に適用される
▶ 呼出しで <code>''</code> を必ず書かせる	<code>def</code> 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	
▶ 辞書型の可変長引数	
▶ 関数外の変数を使う	

▶ 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	
▶ 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

▶ 可変長引数を設定	<code>def hoge(p1, p2 *p3):</code>	※極力後へ
▶ 辞書型の可変長引数	<code>def hoge(p1, p2, **p3):</code>	※極力後へ
▶ 関数外の変数を使う	<code>global 変数名</code>	#（任意）（推奨）ほぼ必須

▶ 関数の呼び出し	<code>hoge()</code>	<code>hoge(<i>arg1</i>, ...)</code>
▶ 関数名の文字列で呼び出し	<code>eval('hoge')()</code>	か <code>eval('hoge()')</code>
▶ 引数名を指定して渡す	<code>hoge(p1=<i>arg1</i>, p2=<i>arg2</i>)</code>	
▶ イテオブの要素全部を渡す	<code>hoge(..., *<i>iterable</i>, ...)</code>	

- ▶ ※ 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	<code>assert 真偽値, 'エラーメッセージ' #</code>	※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	<code>raise 例外型名</code>	※ <code>raise 例外型名(メッセージ)</code> も可
▶ 例外処理を始める	<code>try:</code>	
▶ ある例外の発生時に限り処理	<code>except 例外型名 as e:</code>	
▶ 例外の発生時に限り処理	<code>except:</code>	
▶ 正常終了時に限り処理	<code>else:</code>	
▶ 異常正常によらず最後に処理	<code>finally:</code>	※ <code>except</code> や <code>else</code> 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

▶ 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	
▶ 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	
▶ 何かが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。	
▶ 何かがコンテナかどうか	
▶ ある要素が含まれるか	
▶ 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	

▶ 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 _ はつけられる 例) <code>2_000_000</code>
▶ 算術演算子	+ - * / // % ** ※多重代入 <code>a = b = 3</code> 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	'{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	
▶ 何かが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。	
▶ 何かがコンテナかどうか	import collections.abc as ca isinstance(式, ca.Container)
▶ ある要素が含まれるか	x in <i>container</i>
▶ 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	

- ▶ 何かがイテオブかどうか
- ▶ 全要素に関数を適用
- ▶ ※ `map()` 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
- ▶ ※ 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。

- ▶ 何かがイテオブかどうか `import collections.abc as ca isinstance(式, ca.Iterable)`
- ▶ 全要素に関数を適用 `convFunc※(map(func, iterable))` ※ `list` `''.join` `set` 等
- ▶ ※ `map()` 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`
 - 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

- ▶ ※ `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	
▶ 反対から読んだ文字列	
▶ <i>str</i> を含んでいるか	
▶ <i>str</i> で始まっているか	
▶ <i>str</i> で終わっているか	
▶ 置換	
▶ ?文字目を置換	
▶ 先頭だけ大文字に	
▶ すべて大文字に・すべて小文字に	
▶ 単語の先頭だけ大文字に	
▶ 文字列の前後の空白を除去 (trim)	
▶ <i>str</i> が何文字目に初登場するか	
▶ <i>str</i> が登場する回数	
▶ TEXTJOIN	
▶ ひらがなをカタカナに カタカナをひらがなに	
▶ ※ <code>TypeError: string indices must be integers</code> と出たら → スライスの表記のミスがある	
▶ すべて数字か	
▶ ☆ 数値に変換できるか	
▶ UTF-8でエンコード	

▶ 文字のUnicode値	
▶ Unicode値から文字に	
▶ ☆ 文字が全角か半角か	

■bytesオブジェクト

▶ UTF-8で文字列にデコード	
▶ バイナリファイルを読み込み	
▶ ※ for で回せば 0~255 の整数が返る。	

▶ 変数展開	<code>f'Hello, {name}'</code> ほかに <code>"</code> でも。また <code>F</code> でも。
▶ 文字列の反復	<code>str * n</code>
▶ 途中の文字を取得	<code>s[n]</code>
▶ 途中の複数の文字を取得	スライスで
▶ 反対から読んだ文字列	<code>s[::-1]</code>
▶ <i>str</i> を含んでいるか	<code>str in s</code>
▶ <i>str</i> で始まっているか	<code>s.startswith(str)</code>
▶ <i>str</i> で終わっているか	<code>s.endswith(str)</code> ※パスの拡張子判別に便利
▶ 置換	<code>s.replace(old, new)</code>
▶ ?文字目を置換	インデックスやスライスで
▶ 先頭だけ大文字に	<code>s.capitalize()</code>
▶ すべて大文字に・すべて小文字に	<code>s.upper()</code> ・ <code>s.lower()</code>
▶ 単語の先頭だけ大文字に	<code>s.title()</code>
▶ 文字列の前後の空白を除去 (trim)	<code>s.strip()</code>
▶ <i>str</i> が何文字目に初登場するか	<code>s.index(str)</code> か <code>s.find(str)</code> ※不登場ならエラー <code>-1</code>
▶ <i>str</i> が登場する回数	<code>s.count(〇〇)</code>
▶ TEXTJOIN	<code>'区切り文字'.join(iterable)</code>
▶ ひらがなをカタカナに カタカナをひらがなに	☆☆☆ <code>import jaconv</code> <code>jaconv.hira2kata(str)</code> <code>"</code> <code>jaconv.kata2hira(str)</code>
▶ ※ <code>TypeError: string indices must be integers</code> と出たら → スライスの表記のミスがある	
▶ すべて数字か	<code>s.isdigit()</code> ※全角数字でもTrueに。
▶ ☆ 数値に変換できるか	
▶ UTF-8でエンコード	<code>s.encode('UTF-8')</code> ※: bytesオブ

▶ 文字のUnicode値	<code>ord(char)</code>
▶ Unicode値から文字に	<code>chr(整数)</code>
▶ ☆ 文字が全角か半角か	

■bytesオブジェクト

▶ UTF-8で文字列にデコード	<code>b.decode('UTF-8')</code> ※デコードできない場合はエラー
▶ バイナリファイルを読み込み	<code>with open(filePath, 'rb') as f:</code> <code>f.read()</code> ※: bytesオブ
▶ ※ for で回せば 0~255 の整数が返る。	

■リスト

▶ リストを作成	
▶ 空のリスト	
▶ Splitでリスト作る	
▶ 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	

■リスト

▶ リストを作成	<code>l = [80, 90, 40]</code>
▶ 空のリスト	<code>[]</code> か <code>list()</code>
▶ Splitでリスト作る	<code>str.split('delimiter')</code>
▶ 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]</code> <code>l1.extend(l2)</code> # <code>l = l1 + l2</code>
▶ 要素を削除	<code>del l[n]</code> # 変数 = <code>l.pop(n)</code> <code>del [n:m]</code> # <code>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	<code>l.sort()</code> # か <code>sorted(l)</code>
▶ 降順に並び替え	<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</code> <code>l2 = copy.copy(l1)</code> ・ <code>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	
■辞書	
▶ 辞書を作成	
▶ 空の辞書	
▶ ※ 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。	
▶ 名タプを使う準備	
▶ 名タプを定義	

▶ 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	<code>d = dict(zip(keys, values))</code>
▶ 要素を参照	<code>d[key]</code> ※無ければエラー ※ <code>d[n]</code> は不可 <code>d.get(key, defalut)</code> ※無ければ <code>default</code> に
▶ 要素を追加、上書き	<code>d[key] = value</code>
▶ 存在しない時に限り要素を追加	<code>d.setdefault(key, value)</code> ※: 値 ※上書きはされない
▶ 要素を削除	<code>del d[key]</code>
▶ 値orキーを取り出してリストに	<code>list(d.values())</code> <code>list(d.keys())</code>
▶ 値を合計する	<code>sum(d.values())</code>
▶ 複数の辞書を結合	<code>d = {**d1, **d2}</code>
▶ キーがすでにあるか	<code>key in d</code>

defaultdict

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

■タプル

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

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。	
▶ 名タプを使う準備	<code>from collections import namedtuple</code>
▶ 名タプを定義	<code>Foo = namedtuple('Foo', ['field1', 'field2', ...])</code>

▶ 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	
▶ 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	

<div>■集合</div> <div>▶ 集合を作成</div> <div>▶ 空の集合</div> <div>▶ ※ 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）。</div> <div>▶ 要素を追加</div> <div>▶ 要素を削除</div> <div>▶ すべての要素を削除</div> <div>▶ 集合演算</div> <div>▶ 他の集合と一致するか</div> <div>▶ " を包含するか</div> <div>▶ " の部分集合か</div> <div>▶ " と互いに素か</div>	
--	--

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	

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

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 "	

▶ 名タブを作成	foo = Foo(<i>value1</i> , ...) や Foo(field1= <i>value1</i> , ...)
▶ 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	foo = foo._replace(fieald2 = <i>newValue2</i> , ...)
▶ ※ 属性の追加はできない。	
▶ 属性名たちを取得	foo._fields
▶ 名タブを辞書に	foo._asdict()

<div>■集合</div> <div>▶ 集合を作成</div> <div>▶ 空の集合</div> <div>▶ ※ 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）。</div> <div>▶ 要素を追加</div> <div>▶ 要素を削除</div> <div>▶ すべての要素を削除</div> <div>▶ 集合演算</div> <div>▶ 他の集合と一致するか</div> <div>▶ " を包含するか</div> <div>▶ " の部分集合か</div> <div>▶ " と互いに素か</div>	<div>s = {1, 2, 3}</div> <div>set()</div> <div>s.add(新要素) #</div> <div>s.remove(値) か s.discard(値)</div> <div>s.clear()</div> <div>s1 s2 s1 & s2 s1 - s2 s1 ^ s2</div> <div>s1 == s2 ※ s1 != s2 も可能。</div> <div>s1 >= s2 ※ s1 > s2 にすると等しい場合Falseに</div> <div>s1 <= s2 ※ s1 < s2 にすると "</div> <div>s1.isdisjoint(s2)</div>
--	---

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	fs = frozenset(<i>iterable</i>)

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

テキストストリーム

▶ テキストストリームを作成	f = io.StringIO() ※ 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, 'r'※) ※適宜 'a' 'w' に
	・ with open(パス1, 'r'※) as f1, open(パス2, 'r'※) as f2, ...:

- ▶ エンコーディング指定しつつ `f = open(・・, encoding='エンコーディング')`
- ▶ ※ `UnicodeDecodeError: ~ codec can't decode byte ~` と出たら → エンコーディングを指定
- ▶ エンコーディング `f.encoding`

バイナリストリーム

- ▶ バイナリストリームを作成 `f = open(パス, 'rb'※)` ※ `f` でなく `buf` とされることが多い
- ▶ パスを指定して `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パスを指定してスト作成 `f = open(・・, newline=newline)` ※ `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 `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

- ▶ ※ 以下で説明するキュー、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。
- ▶ 準備 `import queue`

- ▶ エンコーディング指定しつつ `f = open(・・, encoding='エンコーディング')`
- ▶ ※ `UnicodeDecodeError: ~ codec can't decode byte ~` と出たら → エンコーディングを指定
- ▶ エンコーディング `f.encoding`

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パスを指定してスト作成 `f = open(・・, newline=newline)` ※ `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 `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

- ▶ ※ 以下で説明するキュー、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。
- ▶ 準備 `import queue`

▶ 空のキュー	
▶ 空のスタック	
▶ 空の優先度付きキュー	
▶ 要素を 1 つ追加	
▶ 要素を 1 つ取り出す	
▶ 空かどうか・満杯かどうか	
▶ 現在の要素数	
▶ 要素 1 つにたいするタスクの完了をキュー類に伝える	
▶ 全要素にたいしてタスクが完了するまでブロック	

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	
▶ インスタンス変数	

▶ 空のキュー	queue.Queue() ※引数 <code>maxsize</code> 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	.put(<i>item</i>) #
▶ 要素を 1 つ取り出す	.get() ※ 1 要素を取得しつつ削除
▶ 空かどうか・満杯かどうか	.empty() ・ .full()
▶ 現在の要素数	.qsize()
▶ 要素 1 つにたいするタスクの完了をキュー類に伝える	.task_done()
▶ 全要素にたいしてタスクが完了するまでブロック	.join()

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

- ▶ 書込のみ可能な `__`

- ▶ 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

--	--
- ▶ 定数の名前

--	--

- ▶ 読取のみ可能な `__` インスタメソ内で `self.hoge` このうえで、クラス定義直下で `@property``↓` `def hoge(self):``↓` `return 値`
- ▶ 書込のみ可能な `__` インスタメソ内で `self.hoge = 値` このうえで、クラス定義直下で `@hoge.setter``↓` `def hoge(self, hoge):``↓` `self.__hoge = hoge`
- ▶ クラスメソッド `@classmethod``↓` `def HogeHoge(cls, p1, ...):``↓` `...`
- ▶ 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``↓` `def HogeHoge(p1, ...):``↓` `...`
- ▶ ※ `__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 `HOGES = 値1``↓` `FUGA = 値2``↓` `...`
- ▶ `__` する際に値を連番に `from enum import auto``↓` `...↓` `HOGES = auto()``↓` `...`
- ▶ 定数の名前 `Foo.HOGES.name` ※この場合、文字列 `HOGES` が返る

- ▶ 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。