電気情報工学セミナーⅡ

実験のためのPython＆Git入門（機械学習による関数の近似実験）

～　第2回　Python入門　～

池原研究室

イントロダクション

雑談：なぜ黒画面のプログラムを書くのか

プログラミング勉強して
作れるようになりそうなやつ

プログラミングの勉強で
作らされるやつ

GUI（Graphical User Interface）

CUIプログラミングの延長

【入力】	【処理】	【出力】
処理と紐付け 必須の要素 （例）ボタンを押す	決まったことをする 並行処理を意識	処理した結果 表示のデザイン

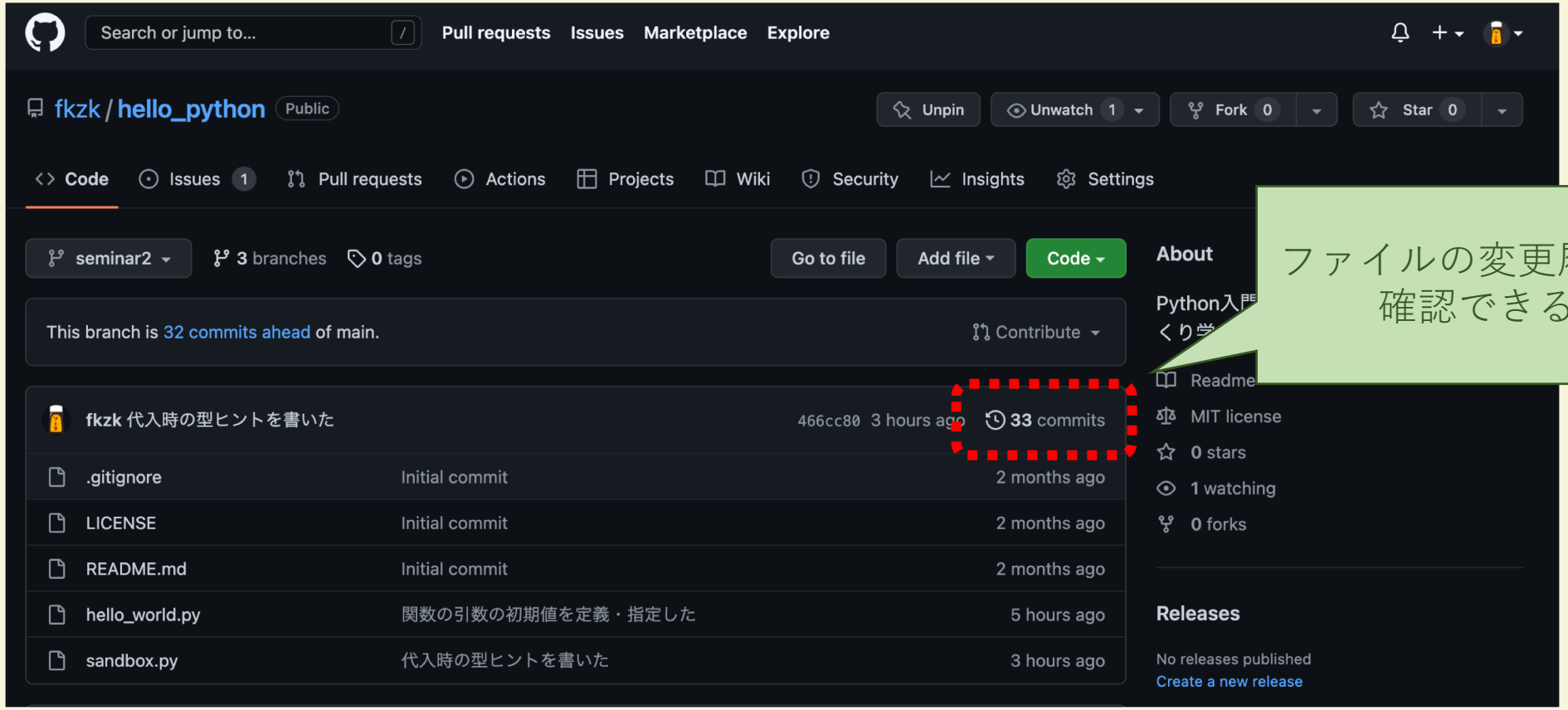
CUI（Character User Interface）

最小限必要なことから学ぶ

【入力】	【処理】	【出力】
~~処理と紐付け~~ ~~必須の要素~~ 文字を打つ一択	決まったことをする ~~並行処理を意識~~	処理した結果 ~~表示のデザイン~~

今回のコード

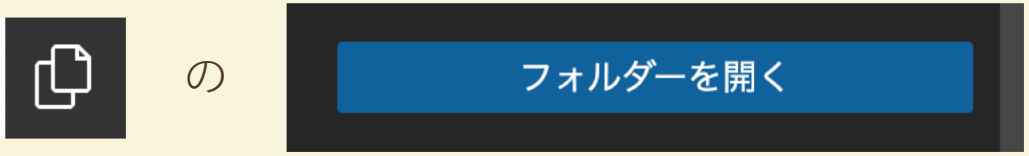
https://github.com/fkzk/hello_python/tree/seminar2



準備：今回の作業ディレクトリを作成して開く

```
$ mkdir $SEMINAR2/hello_python
```

GUIでディレクトリを開く



の

から$SEMINAR2/hello_pythonのフォルダを開く

コマンドでディレクトリを開く

-rは--reuse-windowの略

```
$ code -r $SEMINAR2/hello_python
```

はじめてのPythonプログラム

hello_world.py

```
print('Hello, World!')
```

```
$ python hello_world.py
```

で実行

<参考> hello_world.c

```
#include <stdio.h>
int main(void)
{
    printf("Hello, World!¥n");
    return 0;
}
```

【Hello, World!でわかるCとの違い】

1. #include <stdio.h>のような宣言なしでも標準出力が可能（コンソールへの出力）
2. 文末の;が不要（基本的に1行が1文）
3. int main(void)のようなmain関数不要
4. 改行コードを自動で末尾に追加するprint関数
5. コンパイル→実行ではなくコンパイル不要で即実行

コードをGitで管理するための準備（初期化）

initialize（初期化する）

--initial-branch=mainの略

```
$ git init -b main
```

git initを実行したディレクトリ
（今回は$SEMINAR2/hello_python）
の中のファイルはGitで管理可能に

新しくプロジェクトを始めるときは
git initする（or GitHub上で作成）

※デフォルトでは初期化時に「master」という名前のブランチが作成される
※「master」は奴隷制度を想起させるらしく「main」が多く使われるようになった

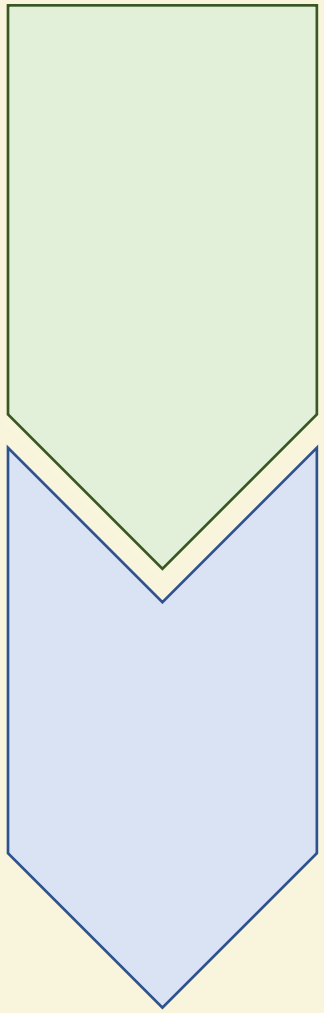
```
$ git config --global init.defaultBranch main
```

しておけば、以降は

```
$ git init
```

だけでOK（v2.28.0以降）

Gitでコードをセーブ（コミット）



①「今回変更した」「今からセーブしたいファイル」を宣言（**ステージング**）

```
$ git add hello_world.py
```

保存したいファイルが複数ある場合は全部 `git add` *ファイル名* する

②ステージングしたファイルをセーブ（**コミット**）

コミットメッセージで
変更内容を記録

```
$ git commit -m '"Hello, World!"と表示するPython
  プログラムを作った'
```

今日覚えるGit操作はこれだけ！

基本文法

代入

hello_world.py

文字列を代入してから表示した

```
message = 'Hello, World!'
print('message')
print(message)
```

1. **変数名 = 代入する値**
2. Cのような型宣言は不要

f文字列を用いた変数の文字列への埋め込み

hello_world.py

- f文字列を使って変数を文字列に埋め込んだ

```
message = 'Hello, World!'
print('message')
print(message)
print(f'{message}')
print(f'メッセージ: {message}')
print(f'{message = }')
```

1. f'　'やf"　"の中で{**変数名**}とすると**値**が展開される
2. {**変数名** = }とすると**変数名** = **値**が展開される

for文による繰り返し

hello_world.py

for文を用いてリストの中身に対する繰り返し処理を書いた

半角80字のライン

```
message = 'Hello, World!'
message_list = ['message', message, f'{message}', f'メッセージ：{message}', f'{message = }']
for item in message_list:
    print(item)
print('繰り返し終了')
```

1. for **リストの中身を表す変数** in **リスト名**:でループ開始
2. **リストの中身を表す変数**に**リスト**の中身が順々に展開
3. ループ内では半角スペース4つぶん字下げ
4. 字下げがもとに戻った時点でループ終了

コード内の文を途中で改行

hello_world.py

リストの中身を改行して見やすくした

```
message = 'Hello, World!'
message_list = [
    'message',
    message,
    f'{message}',
    f'メッセージ：{message}',
    f'{message = }'
]
for item in message_list:
    print(item)
print('繰り返し終了')
```

1. (　)[　]{　}の中は改行OK
2. 1行80字以内に収まるように書く（公式が推奨）
3. コードや変更履歴が見やすくなる

コードにコメントをつける

hello_world.py

コードにコメントをつけた

```
message = 'Hello, World!'
message_list = [
    'message',
    message,
    f'{message}',
    f'メッセージ：{message}',
    f'{message = }', # 実は最後の要素に,をつけてもOK
]
# スライドではここより上の部分を次から省略
for item in message_list:
    print(f'{i_loop}回目のループ: {item}')
```

1. その行の中で# 以降はコードとして解釈されない
2. コードを読む上で役に立つ情報をコメントとして書ける

何回目のループかをカウント

hello_world.py

enumerateを使い、何回目のループかカウントした

```
for i_loop, item in enumerate(message_list):
    print(f'{i_loop}回目のループ: {item}')
```

1. for カウント用, 中身 in enumerate(リスト名):の形式
2. 基本的にはカウント用の変数をループの外で用意しない
3. カウントは0から始まる

if文による条件分岐

hello_world.py

if文を使い、条件に当てはまるときだけの処理を書いた

```
for i_loop, item in enumerate(message_list):
    if i_loop == 0: # == は"等しいかどうか"を判定
        print('最初のループ')
    print(f'{i_loop}回目のループ: {item}')
```

1. if *True_or_False*:の形式でブロックを始める
2. forブロックと同様半角スペース4つで字下げ
3. True_or_FalseがTrueのときだけブロックの中身を実行
4. 条件式（例ではi_loop == 0）はTrueかFalseを返す
   → ifの直後が条件式とは限らない

条件にあてはまらないときに実行されるelseブロック

hello_world.py

elseを使い、条件に当てはまらないときに実行される処理を書いた

```
for i_loop, item in enumerate(message_list):
    if i_loop == 0: # == は"等しいかどうか"を判定
        print('最初のループ')
    else:
        print('早く終わってほしい')
    print(f'{i_loop}回目のループ: {item}')
```

1. if *True_or_False*:ブロックの直後にelse:で始める
2. 半角スペース4つで字下げ
3. *True_or_False*がFalseならelse:の中身を実行
4. 必須ではないので必要なときだけ書けばOK

elifを用いた「～ではない」ときの追加の条件分岐

hello_world.py

- elifを使い、最初の条件に当てはまらないが追加の条件を満たす場合の処理を書いた

```
for i_loop, item in enumerate(message_list):
    if i_loop == 0: # == は"等しいかどうか"を判定
        print('最初のループ')
    elif i_loop < 3:
        print('もう少しがんばろう') # 0回目では実行されない
    print(f'{i_loop}回目のループ: {item}')
```

1. ifブロックの直後にelif *True_or_False*:で始める
2. 半角スペース4つで字下げ
3. ifの条件がFalseならelifを判定してTrueなら中身を実行
4. 必須ではない（複数利用やelseブロックとの併用も可）

条件分岐のまとめ

```
if True_or_False1:
    ... # True_or_False1がTrueなら
elif True_or_False2:
    ... # True_or_False1がFalseかつTrue_or_False2がTrueなら
elif True_or_False3:
    ... # True_or_False1と2がFalseかつTrue_or_False3がTrueなら
else:
    ... # True_or_False1～3がすべてFalseなら
```

各ブロック（字下げは半角スペース４つ）は
上から順に判定したときに
最初に条件を満たすものだけ実行

elif　必須：×　複数：○

else　必須：×　複数：×

関数を定義する

hello_world.py

関数を定義した

```
def is_small(num): # define（定義する）のdef
    return num < 3

for i_loop, item in enumerate(message_list):
    if i_loop == 0:
        print('最初のループ') # これまで使ってきたprintも関数
    elif is_small(i_loop): # 小さいかどうかを判定していることがわかりやすい
        print('もう少しがんばろう')
    print(f'{i_loop}回目のループ: {item}')
```

1. def **関数名(引数):**で関数定義ブロックを始める
2. returnしたもの（戻り値）が実行された場所に展開される
3. return文のない関数も定義可能（Noneという定数が返る）

引数が複数の関数を定義する

hello_world.py

is_small関数の引数を増やした

```
def is_small(num, th):
    return num < th

for i_loop, item in enumerate(message_list):
    if i_loop == 0:
        print('最初のループ')
    elif is_small(i_loop, 3):
        print('もう少しがんばろう')
    print(f'{i_loop}回目のループ: {item}')
```

1. def **関数名(引数)**:の引数は,区切りで複数設定可能
2. 引数は0個でもOK
3. 呼び出し時は定義した順番通り引数を指定

引数の初期値を設定する

hello_world.py

関数の引数の初期値を定義・指定した

```
def is_small(num, th=3):
    return num < th

for i_loop, item in enumerate(message_list):
    if is_small(i_loop, th=1):
        print('最初のループ', end=' / ') # print関数にもキーワード引数がある
    elif is_small(i_loop): # thは省略可能（3として扱う）
        print('もう少しがんばろう', end=' / ') # endの初期値は改行コード
    print(f'{i_loop}回目のループ: {item}')
```

1. **引数名=値** で引数の初期値を設定可能（キーワード引数）
2. キーワード引数は呼び出し時に指定しなくてもOK

Pythonとクラス（型）

main関数つきPythonファイルの作成

sandbox.py

main関数のあるファイルを作成した

```
def main():
    ... # 基本はこの部分だけいじる

# 以降のスライドではこれより下を省略
def second_last_fn():
    ...
# ...
def first_fn():
    ...

if __name__ == '__main__': # sandbox.pyを直接実行したら
    main()
```

過去にメインとしての
役割を果たした関数

変数のクラスを確認

sandbox.py

- tupleにいろいろなものを列挙した
- [演習] tupleの中身の型と値を調べた

```
def main():
    # ( )の中を,区切りで列挙したものをtupleという
    # hello_world.pyで定義していたのは[ ]で囲ったlist
    item_tuple = (
        1, 1.0, "1.0", True, False, None, print,
    )
    print(f'{type(item_tuple) = }')
    print(f'{item_tuple = }')
    print('')
    # [演習]以下にtupleの中身を上と同様に型・値を表示するコードを書く
```

1. type(**変数名**) でクラス（型）がわかる
2. 型宣言がないだけでクラスがないわけではない

listとtupleを使い分けよう

sandbox.py

- listとtupleの使い分けを学んだ
- [演習] fruitsの中身を展開して番号つきで表示した

```
def main():
    fruits = [ # a list of tuples
        ('リンゴ', 'apple', 479),
        ('みかん', 'orange', 339),
        ('いちご', 'strawberry', 2064),
        ('バナナ', 'banana', 185),
    ]
    # [演習] fruitsの中身を展開して「No. i: fruit」の形式でprint
```

1. listは同じ種類のものを入れる
2. tupleは違う種類でもOK
3. listは中身を追加・削除・順序入れ替えしても違和感ない

listクラス固有の関数を使おう

sandbox.py

listクラス固有のappend関数でfruitsにももを追加した

```
def main():
    fruits = [ # 次のスライドから中身を省略
        ('リンゴ', 'apple', 479),
        ('みかん', 'orange', 339),
        ('いちご', 'strawberry', 2064),
        ('バナナ', 'banana', 185),
    ]
    peach = ('もも', 'peach', 837)
    fruits.append(peach) # peach.append('山梨県')は失敗
```

1. list変数名.append(**新しい要素**)で要素を1つ追加できる
2. **変数名.関数名**でその変数のクラス固有の関数が使える

listやtupleの要素にアクセスする

sandbox.py

listやtupleの要素にインデックスでアクセスした

```
def main():
    fruits = [...]
    peach = ('もも', 'peach', 837)
    fruits.append(peach)
    print(fruits[2]) # 番号は0からスタートするため3番目
    print(fruits[-1]) # 負の数は後ろから
    for fruit in fruits:
        print(f'{fruit[0]}({fruit[1]}): {fruit[2]}円')
```

1. list/tuple**変数名[番号]**で要素にアクセスできる
2. **番号**（インデックス）は０から始まる

クラスを作る

sandbox.py

Fruitクラスを作成した

```
class Fruit:
    def __init__(self, jp, en, price): # 初期化用，selfは作成される実例
        self.jp = jp # peachという変数ならpeach.jp = jpに相当
        self.en = en
        self.price = price
def main():
    fruits = [
        Fruit('リンゴ', 'apple', 837), # __init__が実行される。selfは省略
        ...
    ]
    for fruit in fruits:
        # 名前でアクセスできるため可読性◯
        print(f'{fruit.jp}({fruit.en}): {fruit.price}円')
```

クラス専用の関数を作成

sandbox.py

- Fruitクラスに情報をprintする関数を定義した
- [演習] 任意にFruitの税込価格を表示できるようにした

```
class Fruit:
    def __init__(self, jp, en, price):
        self.jp = jp
        self.en = en
        self.price = price

    def print_info(self):
        # [演習] オプションで消費税8%を加えた表示にできるよう書き換え
        print(f'{self.jp}({self.en}): {self.price}円')

def main():
    fruits = [ ... ]
    for fruit in fruits:
        fruit.print_info()
```

足し算: a + b
かけ算: a * b

サブクラスを作る

sandbox.py

Fruitクラスのサブクラス PeelableFruitを実装した

```
class Fruit: ...

class PeelableFruit(Fruit): # Fruitクラスをベースに機能を拡張
    def peel(self):
        print(f'{self.jp}の皮を手で剥いた')

def main():
    fruits = [ ... ]
    peach = PeelableFruit('もも', 'peach', 837)
    for fruit in fruits:
        if isinstance(fruit, PeelableFruit): # PeelableFruitのfruitなら
            fruit.peel()
        fruit.print_info() # PeelableFruitもprint_infoが使える！
```

クラス周辺で覚えておきたいこと

【利用編】
1. 各変数は何かしらのクラスの実例（インスタンス）
2. クラスはまとめて扱いたい変数とそれを扱う関数からなる
3. **実例名.変数名** でその実例がもつ変数にアクセスできる
4. **実例名.関数名** でその実例のクラスの関数が使える
5. **クラス名(引数)** で新しい実例を作れる

【作成編】
1. 関数の第1引数は実例代わりとなる `self`
2. `__init__` という名前の関数に初期化の処理を書く

その他

モジュールのimport

sandbox.py

mathモジュールをimportして利用した

```
import math # mathはPythonの標準モジュールの1つ

def main():
    root2 = math.sqrt(2)
    pi = math.pi
    print(f'{root2 = }')
    print(f'{pi = }')
```

1. import **モジュール名**で"モジュール"を利用可能に
2. **モジュール名.変数や関数やクラス** で機能にアクセス

辞書型 dict

sandbox.py

dictクラスを使ってみた

```
def main():
    fruits = { # 'key': value の形式で列挙
        'apple': Fruit('リンゴ', 'apple', 479),
        'orange': PeelableFruit('みかん', 'orange', 339),
    }
    fruits['peach'] = PeelableFruit('もも', 'peach', 837) # 要素の追加
    for key, value in fruits.items(): # .keys()や.values()もある
        print(f'{key = }')
        value.print_info(with_tax = True)
```

1. keyは見出し語、valueは意味に相当
2. .items() でkeyとvalue両方を使うループが書ける

dictを使った処理の分岐

sandbox.py

dictを使って処理の分岐を実装した

```
def poly(data): print(f'{data}を多項式フィッティングで回帰')
def gp(data): print(f'{data}をガウス過程回帰で回帰')
def nn(data): print(f'{data}をニューラルネットワークで回帰')
def main():
    method = 'poly' # 回帰方法の指定
    regressors = {
        'poly': poly,
        'gp': gp,
        'nn': nn,
    }
    regressor = regressors[method] # regressorはpoly関数になる
    data = 'データ'
    f_x = regressor(data) # poly(data)が呼び出されている
```

```
if method == 'poly': ...
elif method == 'gp': ...
else: ...
      とかしなくていい
```

dictを使った引数の代入

sandbox.py

dictを使って関数の引数を一気に指定した

```
def affine(x, a, b = 0):
    return a * x + b

def main():
    kwargs = dict(a = 1, x = 2, b = 3) # {'a': 1, 'x': 2, 'b': 3}と同じ
    print(f'{affine(**kwargs) = }') # **dict名 で一気に引数を指定
```

数値型 int / float の計算

sandbox.py

さまざまな計算をした

```
def main():
    a = 5
    b = 3
    c = 4
    print(f'{a = }, {b = }, {c = }')
    print(f'{a + b = }') # 足し算
    print(f'{b - c = }') # 引き算
    print(f'{a + b * c = }') # かけ算は足し算より先に計算
    print(f'{(a + b) * c = }') # ( )で優先順位を変えられる
    print(f'{a / b = }') # 普通の割り算（float型になる）
    print(f'{a // b = }') # 整除算（あまりを無視した商）
    print(f'{a % b = }') # aをbで割ったときのあまり
    print(f'{a ** b = }') # aのb乗
```

a + bは
a.__add__(b)ができるか試み、
未実装ならb.__radd__(a)を試みる

→ 自作クラスでも演算を定義可能

真偽値 True / False

sandbox.py

評価式や真偽値の演算を学んだ

```
def main():
    a = 5
    b = 3
    c = 4
    nums = [1, 2, 3]
    print(f'{a = }, {b = }, {c = }, {nums = }')
    print(f'{a == b = }') # 等しいかどうか、!= は等しくないかどうか
    print(f'{b < c = }') # >, <=, >= もある
    print(f'{a == b and b < c = }') # and は両方Trueのときだけ True
    print(f'{a == b or b < c = }') # or は片方でもTrueならTrue
    print(f'{not isinstance(a, int) = }') # not は直後の真偽値を反転させる
    print(f'{a in nums = }') # numsにaが含まれているかどうか
    print(f'{a not in nums = }') # not in はinと逆の真偽値を返す
    print(f'{a is None = }') # aがNoneかどうか、is notで逆の真偽値を返す
```

比較も自作クラスで定義可能

型ヒント・docstring

sandbox.py

- 型ヒントやdocstringのついた関数を定義した

```
def add_complex(
    a: tuple[float, float],
    b: tuple[float, float]
) -> tuple[float, float]:
    """複素数aとbを足す。

    a = p + qi, b = r + siとして、a + b = (p + r) + (q + s)iを計算する。

    Args:
        a (tuple[float, float]): 足される複素数。(p, q) の形式のtuple。
        b (tuple[float, float]): 足す複素数。(r, s) の形式のtuple。

    Returns:
        tuple[float, float]: aとbの和。(p+r, q+s)の形式のtuple。
    """
    return (a[0]+b[0], a[1]+b[1])
```

緑の部分は処理結果に影響を与えないが、
使うときにヒントが表示されて便利

代入時の型ヒント

sandbox.py

代入時の型ヒントを書いた

```
def introduce_dict():
    fruits: dict[str, Fruit] = { # 'key': value の形式で列挙
        'apple': Fruit('リンゴ', 'apple', 479),
        'orange': PeelableFruit('みかん', 'orange', 339),
    }
    fruits['peach'] = PeelableFruit('もも', 'peach', 837) # 要素の追加
    for key, value in fruits.items(): # .keys()や.values()もある
        print(f'{key = }')
        value.print_info(with_tax = True)
```

仕様を詳しく知る方法

公式ドキュメントを読む

https://docs.python.org/ja/3/

自分でコードを書いてみる

今回のsandbox.pyのような
やり方がおすすめ

解説サイトを読む

上２つよりは優先度低め
キーワードが分かったら公式ドキュメントを確認