Python学習資料

この資料はPythonをコンピュータの一つのインターフェイスとして使えるようになるための学習資料である。

習得レベル別に次の三編で構成されている。

- 初級編
- 中級編
- 上級編

初級編では文法や構文といったプログラミング言語としての*Python*を学ぶ。
中級編では自身の作業を効率化できるように一つのシステムとしての*Python*を学ぶ。
上級編では他人の作業も効率化できるように*Python*によるアプリケーション・ライブラリの作成方法を学ぶ。

Pythonとは

PythonはPython Software Foundationというコミュニティが開発している動的型付けインタプリタ言語・システムである。

比較対象として、静的型付けコンパイラ言語である*C言語*と比べてみよう。

動的型付けと静的型付け

*C言語*では変数や引数に必ず"型"を付けなければならない。

```
int x = 1
double y = 1.0

int add(int a, int b) {
    return a + b
}
```

上のコードの`x`と`y`はどちらも"１"ではあるが、型が`int`と`double`で異なるので、コード上では全くの別物である。

これは計算機の仕様でいうと、メモリが関係している。`int`型を宣言するということは、「32bit=4byte分のメモリ領域を確保しろ」という命令を与えることを意味する。`double`型の宣言は、64bit=8byte分のメモリ領域を確保することを意味する。

一方で、*Python*ではコード上で型を意識する必要はない。

```
x = 1
y = 1.0
z = x + y

def add(a, b):
    return a + b
```

厳密に型を指定してコードを書く必要がないので、ふわっとした仕様のままコーディング・実行できるというメリットがある。

動的型付けは90年代後半(JavaScript, Ruby, etc...)のブームであったが、2000年代後半では型による厳密性やメンテナンス性が再び評価され、静的型付け言語(Go [2009, Google])や動的型付け言語に型を付ける(TypeScript [2012, Microsoft], Julia [2012, MIT])のが主流になっている。

Pythonもその流れに乗り、`Python3.6`からは「型アノテーション」がサポートされ、コードの可読性や保守性を向上させることができる。

```
x: int = 1
y: float = 1.0
z: float = x + y

Real = int | float
def add(a: Real, b: Real) -> Real:
    return a + b
```

型アノテーションはただのアノテーションに過ぎないため、実行時には何も効果はない。 上の関数`add`の引数`a`に文字列`"hello"`を入れても、エラーを吐くことはなく実行できてしまう。 これを実行前にチェックするのにmypyというツールを使うのだが、詳しくは上級編で解説する。

私の意見だが、他人に見せるコードには型アノテーションを付けることを推奨する。 型の有無で、読むときと使うときの効率が段違いであるからだ。 一か月後の自分も実質"他人"であるため、書き捨てのスクリプト以外には型アノテーションを付けておいたほうが良いというのが私の経験則である。

インタプリタ言語とコンパイラ言語

プログラムは最終的に0と1で構成されるビット列に変換されてから、電気信号として演算器で処理される。 このビット列はマシン語と呼ばれている。 *C言語*ではソースファイルからマシン語のファイルを生成しなければプログラムを実行することはできない。 このマシン語のファイルを生成する操作をコンパイルという。

マシン語はOSやCPUアーキテクチャによって異なるため、実行ファイルを他者から譲り受けても同じ環境でなければ実行することはできない。 ソースファイルを他者から譲り受けても、実行するには適切なライブラリを用意してコンパイルする必要がある。

*Python*はこうした環境の違いを吸収してくれるので、*Python*さえ動く環境を用意すれば、同じソースファイルを即座に実行できる。

*Python*はソースコードを一行ずつ読み取り、マシン語に変換しながらプログラムを実行している。 このようなソフトウェアをインタプリタという。 インタプリタは実行時に逐次マシン語に翻訳しているので、コンパイル言語と比べると実行が遅い、メモリ使用量が多いなどのデメリットがある。 遅いといっても人間と比べればずっと速いため、無人で何時間も動かす数値シミュレーションやサーバ用アプリケーションでない限り遅さを感じることはない。 高速化するテクニックや手法はいくつか存在するが、ここでは触れないことにする。

良いコードを書くために

プログラムを書くにあたり、一か月後の自分が後悔しないためのノウハウを少しばかり紹介したいと思う。

変数名は省略しない

変数名はできるだけ省略せず、誰もが一意に読める英単語が望ましい。

例えば、「結果」を表す変数を`res`と書いてしまうと、全く何も知らない人が読むと、`result`、`response`、`restore`など様々な推測が浮かんでコードを読むのを阻害する。 はっきりと`result`と書けば、誰がどうみてもそれが「結果」を表すオブジェクトであるとわかる。

また、`result1`、`result2`のような最後に数字を付ける変数命名は自分が後悔することになるので絶対に使わないこと。

変数を定義した場所と使う場所を離さない

変数は処理直前に定義するほうが、コードを読むときに余分な記憶領域を使わずに済む。

悪い例

```
def foo():
    tmp = bar()
    ...    # 10行のコード
    result = xxx(tmp)
```

良い例

```
def foo():
    ...    # 10行のコード
    tmp = bar()
    result = xxx(tmp)
```

定数は大文字で書く

これは*Python*の慣例の一つある。

```
PI = 3.14159
```

*C言語*の`const`とは違うため値を変更することができるが、静的解析ツールを使用することで実行前にそれを発見することができる。

他にも*Python*にはコーディングの慣例があり、PEP8としてまとめられている。 PEP8を遵守しなくてもプログラムは実行できるが、Python開発メンバーによる保守性や堅牢性のためのベストプラクティスであるため、「良いコードとはどんなものか」を学ぶためにも一度目を通しておくと良いかもしれない。 特に読んでほしいPEP項目を以下に挙げておく。

- Style Guide for Python Code
- The Zen of Python
- The Theory of Type Hints

大きいスクリプトは関数化する

スクリプトとして書いた*Python*は一行一行読み込みながら実行しているため遅いが、関数化することでマシン語に翻訳してから実行できるため多少の速度改善が望める。

```
import time

n = 100000000

# 関数化しない場合
start = time.time()
for i in range(n):
    _ = i ** 2
print("before:", time.time() - start, "[s]")


def func() -> None:
    for i in range(n):
        _ = i ** 2


# 関数化した場合
start = time.time()
func()
print("after:", time.time() - start, "[s]")
```

```
before: 1.8978188037872314 [s]
after: 1.6133537292480469 [s]
```

型アノテーションをつける

最低でも関数の引数と戻り値には型アノテーションは絶対つけるようにする。
それをしない場合、自作関数を使うたびに関数全体を読み直すことになる。

再利用する頻度が高い場合は、docstringを付けておくのも望ましい。

```
import os

def load(filename: str | os.PathLike[str]) -> list[list[str]]:
    """
    Load data from file

    Parameters
    ----------
    filename : str | os.PathLike[str]
        Path to a file to load.

    Returns
    list[list[str]]
        The loaded data.
    """
```

docstringはNumpyスタイルとGoogleスタイルがメジャーである。

とにかく書く

最初から完璧なプログラムを書ける人は存在しない。 とにかく、まず書いてみることが重要である。

とりあえず書いた雑なコードを何度もリファクタリングすることによって、コーディングスキルとコードは洗練されていく。

一つの言語を習得すれば、「*Python*でこういう処理を、この言語で実装するにはどうすれば良いですか？」という質問ができるようになるので、他言語の習得速度が必ず上がる。

*Python*はエラーも比較的わかりやすいから、とりあえず書いて試行錯誤してみよう。

初級編の概要

初級編ではプログラミング言語として*Python*を学ぶ。

内容は大きく分けて次の３つである。

- プログラムの実行方法
- 文法・構文
- 組み込み関数

文法・構文は最低限のものをまとめておいた。 詳しくは公式チュートリアルを参照してほしい。

各オブジェクトのより詳しいメソッドについても、公式ドキュメントを参照するかGoogleすると良い。

組み込み関数は、頻繁に使用するものを挙げておいた。全ての組み込み関数はここで確認することができる。

中級編の概要

中級編では*Python*を使った自身の作業の効率化の方法を学ぶ。

内容は大きく分けて次の4つである。

- ライブラリの使用方法
- 標準ライブラリの紹介
- サードパーティライブラリの紹介
- 効率化アプリケーションの例

*Python*は標準ライブラリが充実しているので、一度目を通すだけでソースコードの量が減ったり、できることが増えたりする。 私は学習初期に標準ライブラリの気になった項目を１日１つ読んでいた。

サードパーティライブラリについては、私の知る限りで頻繁に使用するライブラリを挙げておいた。 さらに*Python*でできることを増やしたい場合は、PyPIやGitHub検索すると良い。

アプリケーション例のコードは、`app_examples`ディレクトリにまとめられている。

上級編の概要

上級編ではライブラリ・アプリケーションの作成方法を学ぶ。