*ロゴ

自動的に生成された説明黒い背景に白い文字がある

中程度の精度で自動的に生成された説明*

Python講座

本当に何とでもなる

文責：　うね@雪

**目次**

[-はじめに- 4](#_Toc90478883)

[基本編 5](#_Toc90478884)

[1 Pythonってなんだ(飛ばしてもいいよ) 5](#_Toc90478885)

[1.1 Python 5](#_Toc90478886)

[1.2 オブジェクト指向って何！！ 6](#_Toc90478887)

[2 開発環境(なんなら一番面倒くさい) 7](#_Toc90478888)

[2.1 開発環境とは 7](#_Toc90478889)

[2.2 なにがあって,なにがいいの？ 7](#_Toc90478890)

[2.3 インストール方法 & 使い方(Anaconda) 7](#_Toc90478891)

[2.4 インストール方法 & 使い方(Python + VScode) 7](#_Toc90478892)

[2.5 (おすすめ)使い方(GoogleColaboratory) 7](#_Toc90478893)

[3 モジュール？ライブラリ？ 8](#_Toc90478894)

[3.1 言葉の説明 8](#_Toc90478895)

[3.2 インストール方法 8](#_Toc90478896)

[3.3 使い方(import? from?) 10](#_Toc90478897)

[4 型の話 12](#_Toc90478898)

[4.1 変数について 12](#_Toc90478899)

[4.2 色んな型 13](#_Toc90478900)

[4.3 Pythonにおける型の扱い 14](#_Toc90478901)

[4.4 四則演算(+, -, \*, /, その他) 15](#_Toc90478902)

[5 ごめん、結構ごっそりまとめる 17](#_Toc90478903)

[5.1 Pythonのインデント(Tabキー) 17](#_Toc90478904)

[5.2 リスト！ 19](#_Toc90478905)

[5.3 辞書も便利 21](#_Toc90478906)

[5.4 関数(ちょっとだけ) 22](#_Toc90478907)

[5.5 条件(真 と 偽) 23](#_Toc90478908)

[5.6 分岐処理(if / else if / else) 24](#_Toc90478909)

[5.7 ループ処理(for / while) 26](#_Toc90478910)

[5.8 <発展>ループ処理(enumerate) 29](#_Toc90478911)

[6 関数作ろー！ 30](#_Toc90478912)

[6.1 関数の構造 30](#_Toc90478913)

[6.2 実際に作ってみよー 30](#_Toc90478914)

[6.3 軽－く練習問題 33](#_Toc90478915)

[7 クラスって、、？メソッドと関数、、？？ 37](#_Toc90478916)

[7.1 クラス・インスタンスという概念(わたあめ) 37](#_Toc90478917)

[7.2 ひとまずクラスを作ってインスタンス化 38](#_Toc90478918)

[7.3 \_\_init\_\_なにそれおいしいの？ 39](#_Toc90478919)

[7.4 クラス変数、インスタンス変数 41](#_Toc90478920)

[7.5 メソッドと関数って何が違う 42](#_Toc90478921)

[7.6 プライベート変数 45](#_Toc90478922)

[7.7 <発展>デコレータ(@staticmethod, @property, etc.) 46](#_Toc90478923)

[7.8 <発展>継承 50](#_Toc90478924)

[7.9 <発展>続・継承(super().\_\_init\_\_) 52](#_Toc90478925)

[7.10 <発展>抽象クラス/抽象メソッド 54](#_Toc90478926)

[7.11 <発展>Protected変数 56](#_Toc90478927)

[8 アノテーション 58](#_Toc90478928)

[8.1 アノテーションとは 58](#_Toc90478929)

[8.2 <発展>typing 59](#_Toc90478930)

[8.3 <発展>Any, Union, Optional 60](#_Toc90478931)

[8.4 <発展>Generics 61](#_Toc90478932)

[9 例外処理って便利なんだよ 63](#_Toc90478933)

[9.1 tryの使い方 63](#_Toc90478934)

[9.2 実際に使ってみる(AttributeError回避) 64](#_Toc90478935)

[応用編 66](#_Toc90478936)

[おまけ 67](#_Toc90478937)

[ エラーコード早引き 67](#_Toc90478938)

[ こわざ(ショートカットキーetc.) 67](#_Toc90478939)

[ よく使うメソッド 67](#_Toc90478940)

[ リストに対するメソッド 67](#_Toc90478941)

[ 文字列に対するメソッド 67](#_Toc90478942)

[ よく使うライブラリ 67](#_Toc90478943)

[ numpy 67](#_Toc90478944)

[ matplotlib 67](#_Toc90478945)

[ pandas 67](#_Toc90478946)

[ バージョン管理 68](#_Toc90478947)

[1. バージョン管理の考え方 68](#_Toc90478948)

[2. Gitとは 68](#_Toc90478949)

[3. ブランチ 68](#_Toc90478950)

[4. リモートとローカル 68](#_Toc90478951)

-はじめに-

この文書が僕の知りえないところまで広まってないことを少しだけ祈っておきます。(ハズいから)

改めまして一応本名は伏せておきます。はじめにっても書くことが特にありません。「この物語はフィクションです。実在の人物や団体などとは関係ありません。」って言っといたら間違えても怒られないかな。めっちゃ実在する言語だけど。

さて、なぜこんなものを作るに至ったか。まぁ暇だからが8割なんですけど、残りの2割は色々です。現時点でPython全く知らんって人は今後Pythonに触れる機会はないと思いますが、意外と知っておいて損はない気がします。便利なんだよ、プログラミングって。一方、Pythonやり始めたけど、いまいち何するか分からんって人や、結構もう知ってるよって人。そんな人たちにもなんかの手助けになれないかなーって思って書いてます。なんと初心者から中上級者まで、全年齢対象でございます。Gって付けても怒られない気がします。とはいえ一応、大学生推奨って感じです。でもR18って付けたら怒られそうです。

この時代、小学生でもプログラミングに触れるようになっています。さすがに小学生ががっつりjava, Pythonを扱うのは珍しいですが、概念だけでも幼いころにやっていれば、中学生高校生ではもう十二分にコードが書けるようになります。ぶっちゃけ恐ろしい世代です。そんな化け物どもに対抗するためにはこちらも武装しておかねばなりません。今現在、ほとんどの企業で、ほとんどの部署で、ほとんどの業界で、コンピュータは使われています。ってことは、どこででもプログラミングってできるわけですね。どれだけプログラミングから離れてても導入される危険性があるわけです。そんな万が一に備えるためにも、一応触れてみてもいいんじゃないだろうかと思うわけです。

そうなって勉強しようってなった時、世の中の参考書を否定するわけじゃないですけど、どれも難しいんすよ。いやマジで。怖いもん。なので、この資料は、できーるだけ簡単に、できーるだけ気を抜いて読んでもらえるよう書いてるつもりです。分かんない箇所があれば、個別対応もできるので、ぜひぜひ一読していただけると嬉しいです。そんな感じでよろしくお願いします。

基本編

1. Pythonってなんだ(飛ばしてもいいよ)
   1. Python

Pythonはオブジェクト指向型プログラミング言語の代表的な言語です！……と言われて”はい、そうね”って流してもらうのが一番平和だと思います。オブジェクト指向、これは後々さっくり説明しますが、別に分からなくても特に問題ないです。その割に色んな入門書では、さも重要そうに書くんです。いや入門で書くなよって思う。

他にPythonってなにさって言われるとなんだろうな。僕の勝手な感覚ではjavaとCを足して2で割った上に角を丸くした、、って感じですね。ベースはjavaみたい、でも計算はC言語を用いてるっぽいです。なんかそんなもんです。

じゃあ、そもそもプログラミング言語ってなんなんだ。元々コンピュータってのは0, 1しか受け付けない人なんです。普通に考えてスワヒリ語圏の人に日本語で喋っても伝わらないよね。僕らの言葉とコンピュータとの仲介(翻訳)をしてくれるのがプログラミング言語です。ほなjavaだとかCだとかPythonだとかってなんやねんと。ざっくり言っちゃえば英語と日本語とスペイン語みたいな感じ。言ってることは一緒だけど文法が違うの。だから、専門的な拘りが無ければ基本的にどれを使っても大丈夫なんです。一応、それぞれ得意不得意があったり翻訳方法が違ったりします。

さて、有名な話ですが、AIといえばPythonという風潮があります。なぜかと言いますと、Pythonでは豊富な機械学習ライブラリ(後述)が公開されているからです。だから僕はPython。ってか、単純に文法が簡単なんだよね。javaを勉強したことがある人は分かると思うけど、異常にオブジェクト指向なの。だから結構しっかり理解して覚えて書かないといけないし、読むのも難しい。けど、Pythonって割とてきとーに書いても動いてくれるんです。……ほんとだよ？

そんな愛すべきPythonのお話、始まり始まりー。

* 1. オブジェクト指向って何！！

実は、僕もオブジェクト指向っていうの最近までよく分かんなかったんですよねー。実務でようやくなんか理解した感じ。

よく参考書にある記述がこれです。「オブジェクト指向とは、すべてをオブジェクトとして扱うこと」……は？(ガチギレ)　説明ってかもはや言葉そのままやないか！　発酵を「発酵させること」って書いてるのと一緒。発酵ってなんやねん。

そんな感じでずーっと分かんなかったんですよね。さて、では僕なりの説明をしていきます。厳密には間違ってるかも、ごめんね。

まず、オブジェクト。これはなんだ。言葉の意味そのままだと“物”ですね。その訳はいったん置いといてください。ね。今からオブジェクトとは、“操作できるもの”だと思ってください。例えば、スマホ。僕らはスマホで、文字を打ったり、動画を検索したり、ゲームをしたりできます。僕たちは、スマホを操作できるわけです。現実世界のスマホはオブジェクトにあたるんですね。もう少し考えてみましょう。じゃあねー、テレビ！　テレビも、電源を入れる、チャンネルを変える、音量を上げる、などなど操作ができます。つまり、テレビもオブジェクトです。

オブジェクトの概念が何となくでも分かりました？　現実世界ではスマホやテレビなど、主に機械がオブジェクトです。じゃあ、オブジェクト指向ってなんだと。ここで最初の言葉“(プログラミングにおける要素)すべてをオブジェクトとして扱う”の意味がちょっと分かったんじゃないでしょうか。

プログラミングにおける要素ってのは、要は変数のことですね。変数ってのは、まぁ“なんか入ってる箱”ぐらいでいいです。Pythonを例にして、a =‘test’と変数を設定します。今、aには文字列(test)が入っています。オブジェクト指向では、aを操作できるものとして扱います。例えば、この文字を大文字にする(TEST)、一番左の文字を消す(est)、特定の文字を消す(es(tを削除))、などの操作ができるよう言語を作ってるよってのがオブジェクト指向です。

オブジェクト指向、要は言語の設計図ってことですね。

1. 開発環境(なんなら一番面倒くさい)
   1. 開発環境とは

これを作るにあたって僕が一番面倒くさいと感じるのがこの章です。めんどうすぎて、一番最後に書いてます。

1.1で少し触れましたが、プログラミング言語ってのは翻訳機です。となると、翻訳機に僕らの言語を渡してあげなくてはいけません。その橋渡しをしてくれるのが開発環境ってやつです。要はコードを書くところです。ただ、何が面倒くさいって、なんか一番不具合が多いんですよねー……。しかもこの機種はエラー出ないのにこっちの機種ではエラーが出るとか、環境依存してたりもします。えぐい。

一応、僕の環境におけるインストール画面を踏まえつつ説明していきます。基本的にこの通りにやれば大丈夫だと思います、、。

* 1. なにがあって,なにがいいの？

開発環境には、有名どころが何個かあります。軽く名前と説明の一覧を載せますね。



ざっとこんな感じでしょう。この中から、凡庸性の高いVisualStudioCode(VScode), 自前の統合開発環境を作りやすいAnaconda, 接続以外では最強のGoogleColaboratory(GoogleColab)の説明をしていきたいと思います。

が！ 僕はWindowsユーザーでMacが一台も手元にありません……。おそらくMacでも導入方法は変わらないと思いますが、んー……どうなんだろう。Homebrew+pyenvの方がいいって話もあるし、、。まぁその辺がうまくできる人は勝手にやってください。とりあえず、ここではPythonを直接インストールします。(脳筋)

そしてもう一つ。この文書はPython3.9を元に書かれています。もしPython3.10以降仕様変更が起こった場合、対応できません。Python4が出たら一から書き直しレベル笑　なので、ほんとに初めてだよって方はPython3.9をインストールするのがいいかもしれません。(パッと見、Python3.9とPython3.10は違いないけどね……)

あ、Python2ユーザーは知りません、ごめんなさい。

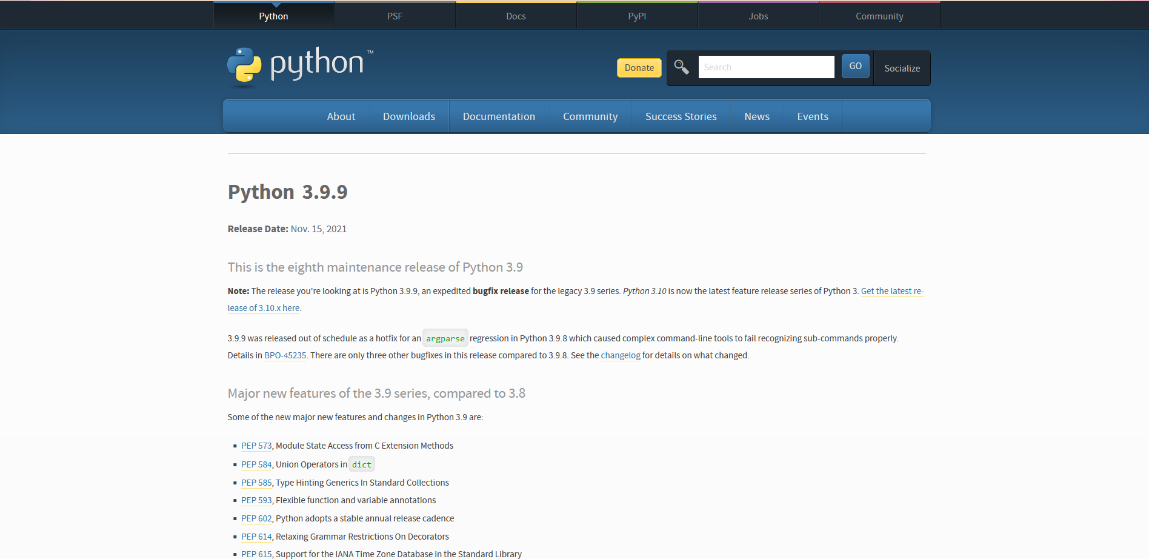
* 1. インストール方法 & 使い方(Python + VScode)

正直、あんまりおすすめはしません。一応ってかんじです。

まずはVScodeから。VScodeにはPython自体は入ってません。VScodeが用意するのはあくまで“プログラムを書く場所”だけです。なので、まずはPythonからダウンロードしていきます。

“Python Download: <https://www.python.org/downloads/>”

ここから好きなバージョンをダウンロードしていきます。目的のバージョンの“Download”を押すと、こんな画面に連れていかれます。



ここの一番下に、WindowだったりMacだったりのインストーラーがあるのでパソコンに合うものをダウンロードしてください。(今のパソコンは大体64bitだと思う)

インストーラーを起動したらこんな画面が出るはず。(Windows)

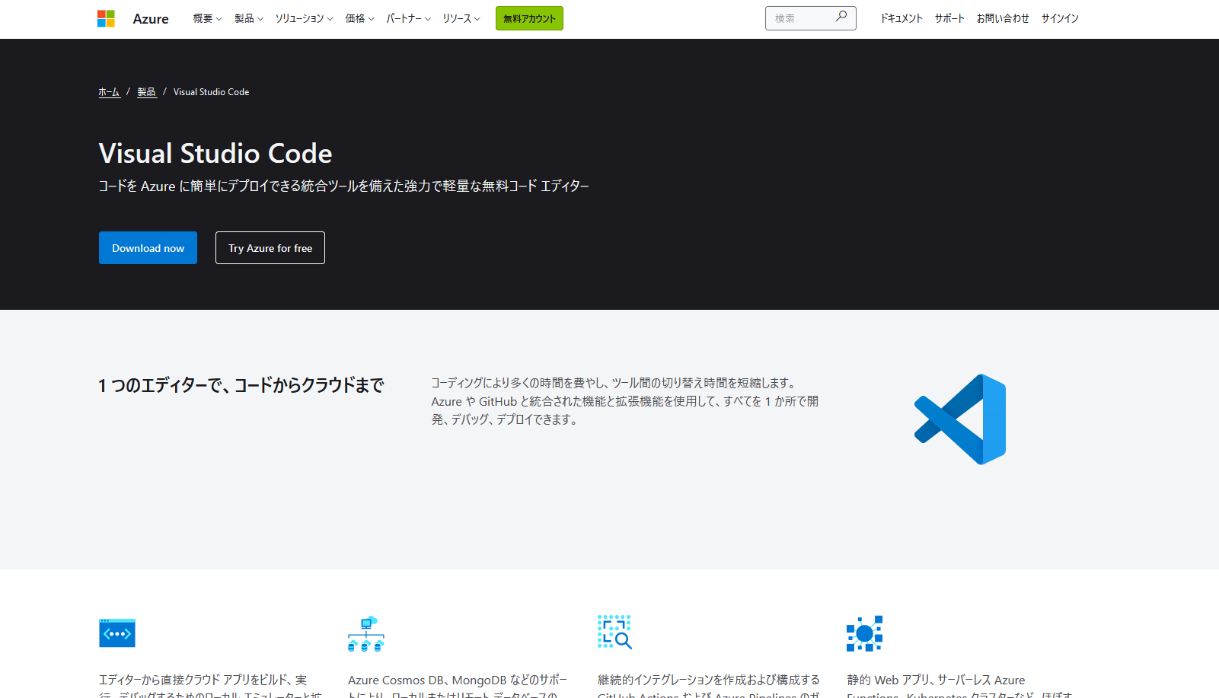


“Add Python 3.9 to PATH”にチェックを入れてInstall Nowを押してください。(バージョンを切替するならチェックは入れない)

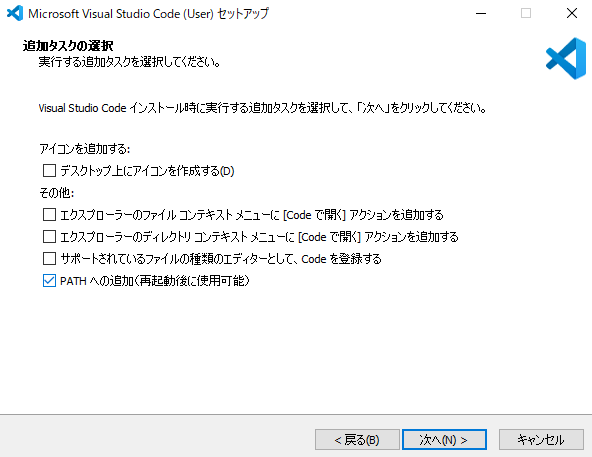
とりあえず、これでPythonはインストール完了です。ここで一回パソコンを再起動しといてください。(念のため)

次にVScodeをダウンロードします。こちらは大したことないです。

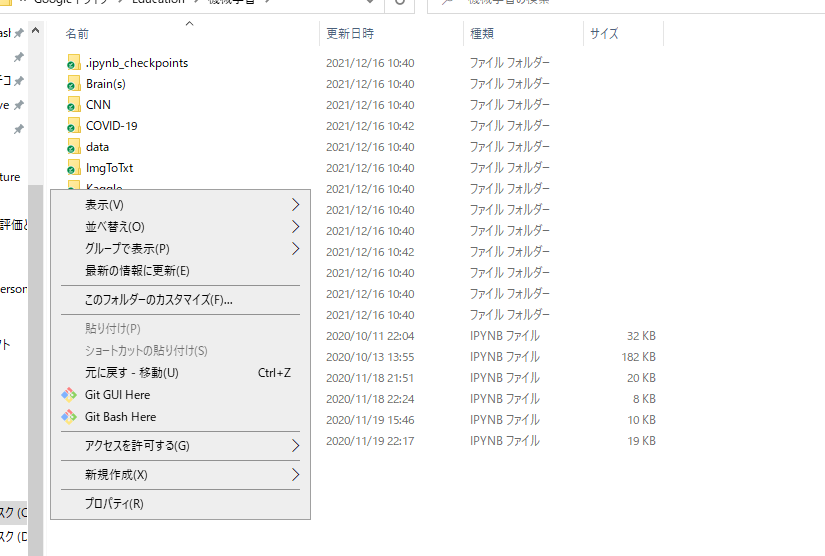
“VScode: <https://azure.microsoft.com/ja-jp/products/visual-studio-code/>”



ここから自分に合うものをダウンロードするだけですね。インストーラーをダウンロードして起動すると、同意させられて、ここまできます。



PATHへの追加はチェックしといてください。(環境関わらず)　あとは自由ですが、[Codeで開く]アクションはまぁ便利だったりします。あの、、、フォルダで右クリックすると出てくる、このメニューに「VisualStudioCodeで開く」が追加されます。



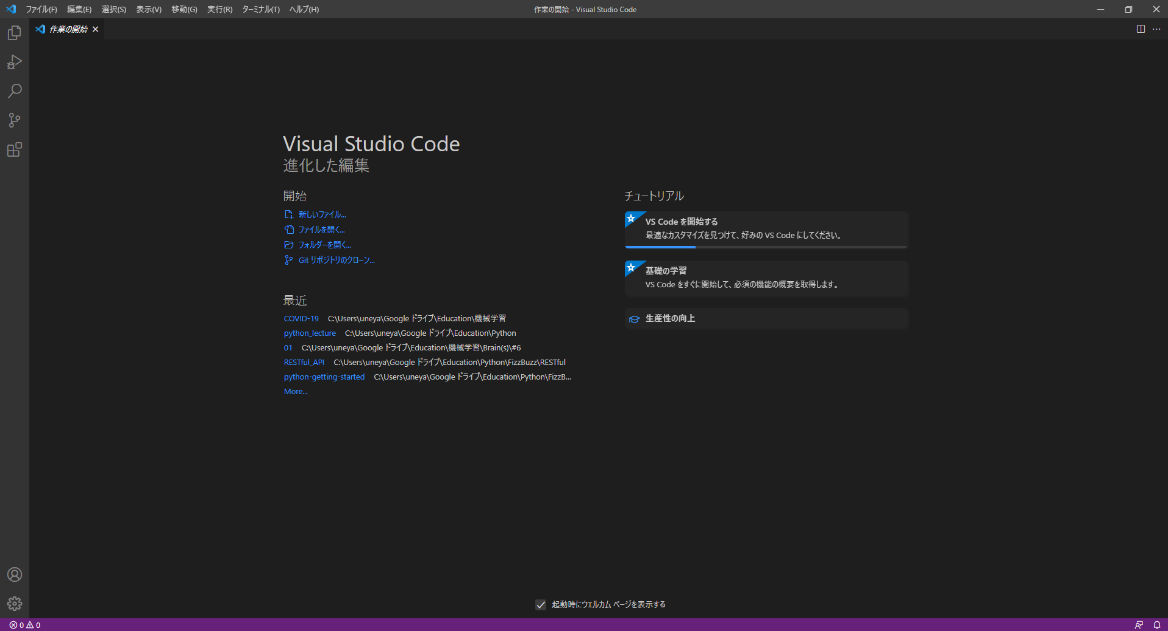
VScodeほぼ使わないし邪魔なので僕は入れてないです笑

インストールが終わったら再起動してくださいね。どんなソフトでもインストール後には再起動しておくのをおすすめします。

さて、これでVScode本体とPythonが用意できました！　ここからVScodeで快適にPythonが書けるように拡張機能を入れていきます。VScodeには企業やユーザーがたくさんの拡張機能をアップしています。それを使うことで使いやすくなっていくんですね。すごい。

ここでは基本的なものだけ紹介します。

VScodeを起動するとこんな画面ですね。いっしょかな。(再インストールしたのに「最近」が残ってるの怖すぎなんだけど、、)



左のバーのメニューから「拡張機能」を選択してください。あ、初期状態だと「Extensions」ですね、。

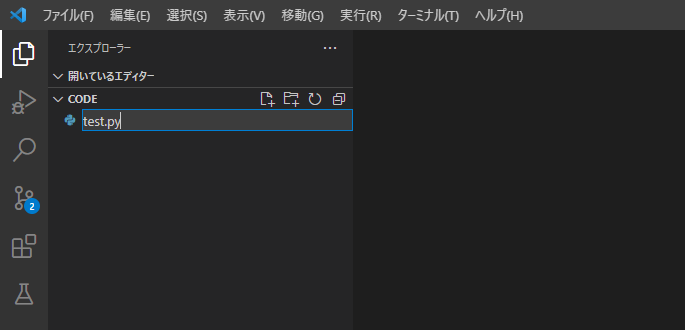


すると、検索バーがあるので、そこで検索して、とりあえず下の一覧ぐらいはインストールしときましょう。

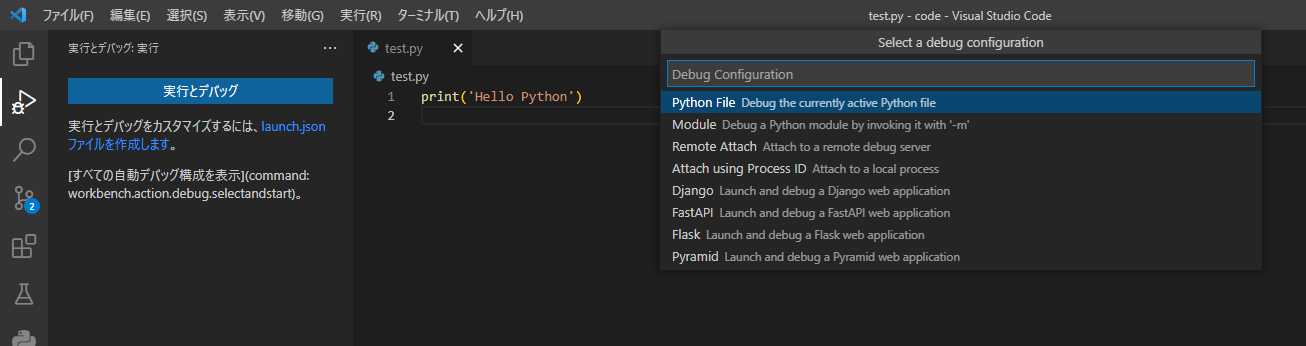


他にも色んな拡張機能があります。久しぶりに見たけど、Python PathとかPython Previewとか便利そうだな―って思った。ちなみに、各拡張機能の紹介ページには動画っぽいのがあることが多いので、英語読まなくてもなんとなく使い方は分かります笑　(Python Extension Packの中身は「詳細」の「拡張機能パック」ってとこから見れるよ)

これにてVScodeの環境構築は完成！　では実際に実行してみましょう。「ファイル」→「フォルダーを開く」でファイルを作りたい場所を選択してください。開けたら、Pythonファイルを作りましょう。



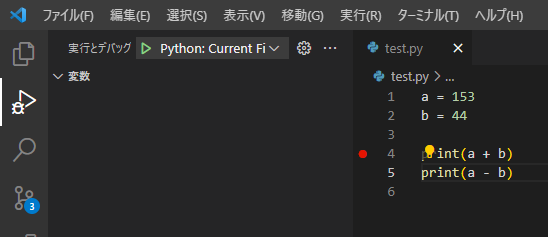
赤丸のところから新規ファイルを作って、てきとうに名前をつけてください。「.py」を最後につけてね。



左メニューバーの「実行とデバッグ」を押して、「launch.jsonファイルを作成します」を押すと、なにやら選ばされますが、何も考えずに「Python File」を選んでください。脳死で。

launch.jsonの詳細は説明しません。めんどうです。launch.jsonは

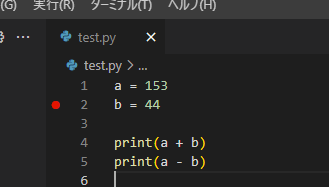
×でそっと閉じてください。そっ閉じ。



　そしたら、なにやら実行できそうな感じですね。てきとうにコードを書いて、緑矢印を押すとデバッグが始まります。

　最後にデバッグの説明だけ……。デバッグって地味に意味が違ったりするんですけど、「デバッグ機能」のことだと思ってください。

　VScodeに限らず、コーディングソフトにはデバッグ機能がついてるものがあります。この機能の説明をしていきますね。



まず、コードの左側、数字がついてます。これは言うまでもなく行数ですね笑　コードってのは基本的にこの行数通りに進行していきます。その左側に赤丸があるの分かりますか。ここら辺をクリックしたら出てきたり消えたりします。これをブレークポイントと言います。デバッグを行うと、この行で実行をストップしてくれるんです。

するとその時点での変数の状態などが分かります。エラーが出た行の直前でストップして何がおかしいのか見るのが多いですね。

これでデバッグを開始すると、変なメニューバーが出てきます。



　左から順に一覧にしますね。便利な機能達です。



まぁ使ってみてください、結構重宝しますので。

ってな感じでVScodeの説明は以上です！　お疲れ様でした～。

* 1. インストール方法 & 使い方(Anaconda)

まぁ使ってみてください、結構重宝しますので。ってな感じでVScodeの説明は以上です！　お疲れ様でした～。

* 1. (おすすめ)使い方(GoogleColaboratory)

めちゃめちゃ便利。環境構築不要の統合開発環境です。

まず、てきとうなGoogleアカウントを用意します。メインでもサブでも、なるべく容量ある方がいいかもです。

“GoogleColab: <https://colab.research.google.com/notebooks/welcome.ipynb?hl=ja>”



一番最初はこの画面です。ここから「ファイル」→「ノートブックを新規作成」で白紙のノートブックを作れます。

1. モジュール？ライブラリ？
   1. 言葉の説明

さて、2.2でちょっと触れた、ライブラリの話ですね。Pythonには多種多様なライブラリが存在します。ライブラリってのは、 “モジュールが格納されているもの”です。……モジュールってなんやねん。モジュールってのは、“便利な部品”ですね。なんか良い例があるといいんですけど……。なんかもう分かんないから、書庫と思ってください。ライブラリってのが書庫で、モジュールが本です。書庫には本が格納されてますよね？　そんな感じ。書庫ってなんか地味だけど、後で説明がすっごく楽になるんです。

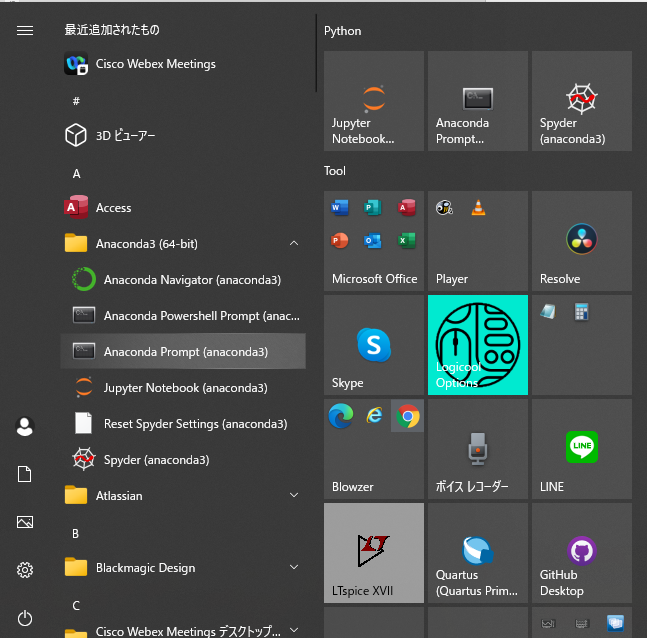
* 1. インストール方法

先ほど説明した、ライブラリ。元々Pythonに同封されているやつはいいんですけど、基本的に入ってないのが多いです。そのため、ライブラリを開発環境にインストールしてあげないと使えないんです。

ライブラリのインストール方法ですが、環境によってちょっと違うので2パターン記しておきます。あ、GoogleColabはほぼインストールされてるので特に書いてません。

<Anaconda>(Windows, Mac同じ)

Anacondaを使う場合は、まず、“Anaconda Prompt”というアプリを探してください。Windowsだと下図のようにAnaconda3の中にあると思います。最悪検索バーで検索してください。



起動すると、



こーんな画面が出るはずです。そこで、

|  |
| --- |
| conda install ライブラリ名 |

と打って、Enterキーを押すとインストールが始まります。とはいえ、Anacondaには元々よく使うライブラリはインストールされてるので、あんまり追加で入れる必要はないのかなって思います。

<Python + VScode>(Windows, Macちょっと違う)

こっちも似たような感じ。AnacondaPromptではなく、OS標準のプロンプトを使うだけです。各OSのプロンプトはこんな感じ。

Windows → コマンドプロンプト or WindowsPowerShell

Mac → ターミナル

そいつらを起動したら、

|  |
| --- |
| pip install ライブラリ名 |

と打ち込んでやれば、ライブラリのインストールが始まるはずです……が！あくまで僕の環境の場合です。

なんやかんやでエラーが出ることもあるみたいです、、。ちょっとそこまで書き切れないので、なんとなくの対処法のみ箇条書きにしておきます。

・pipを再インストールする。(検索したらやり方出てきます)

・pythonを再インストールする。(普通にアンインストール)

……これくらいかな。意外と無いですが、まぁ大丈夫かと。

* 1. 使い方(import? from?)

「ライブラリをコード内で使いたい！」

とあるコード内でライブラリを使いたいのですが、すべてのライブラリを導入してしまうとメモリ使いすぎて死にます。と、いうことで、特定のライブラリだけをコードにインポートしてあげたいわけです。めっちゃ簡単です。

|  |
| --- |
| import ライブラリ名 |

こんだけ。よゆーっすね。ちなみにasを後ろに付けると、自分で名前を付けることができます。慣例的に付ける名前は決まってたりします。numpy→np, matplotlib→plt, pandas→pdだったりね。numpyを使うときはこんな感じ。

|  |
| --- |
| import numpy as np  my\_array = np.array([0, 1]) |

ね、ちょっと短くなるでしょ。そんだけです笑　この例だと、numpyがライブラリ、arrayがモジュールです。

問題は、fromの使い方。先ほど、“ライブラリは書庫、モジュールは本だ！”って言ったわけですが、書庫って本棚がありますよね。同じようにライブラリにもモジュールを格納している場所というものがあります。その場所に行かないと本が取れないわけです。それを何回も書くのは面倒なので、最初に指定しているのがfromです。

|  |
| --- |
| from 場所 import モジュール名 |

あんまり気にしなくていいんですけど、importの後がライブラリ名からモジュール名に変わってます。まぁ、そんなに気にしないでください。ちょっと例を見つつ、「こんなもんかぁ」って感じで。

下の二つは実質的に同じことをやってます。(コードの内容はスルーの方向で)

|  |
| --- |
| from torch.ulils.data import DataLoader  test\_loader = DataLoader(TensorDataset(data, test\_label)) |

|  |
| --- |
| import torch  test\_loader = torch.utils.data.DataLoader(TensorDataset(data, test\_label)) |

コーンな感じ。別に一回ぐらいなら下の量書いてもいいんだけど、これ最低2回は書くんだよね、めんどくさいし、汚い。ってことで、上のようにfromを使って場所を指定してあげるんですね。

正直、自らの意志を以てfromを使うなんてそうそうないです。参考(コピペ)したコードでfrom使ってたら、「そーなのかー」ぐらいでいいと思います。ってか俺も覚えてない……だってtorchってマジで多いんだもん……。まぁ、使いたいやつがあれば「(ライブラリ名)　使い方」って検索したらたいてい出てきます。

結論：

　検索したものをそのまま使えばいい！

1. 型の話
   1. 変数について

一番最初に変数の話が出た時、“なんか入ってる箱”と言いましたが、それ以上の説明がありません……。特にPythonの場合、本当に“何か(数字だったり文字だったり)が入ってる”としか言いようがないんです。

さて、そんな変数。最初に定義してあげないと使えません。定義の方法はめっちゃ簡単です。

|  |
| --- |
| 変数名 = 値 |

こんだけ。“=”は代入演算子と言われます。まぁ、要は左に右のやつをぶち込んでるだけです。もし定義を行わず、変数を使おうとすると以下のようになります。

|  |
| --- |
| print(a) |
| Traceback (most recent call last):  File "Main.py", line 1, in <module>  print(a)  NameError: name 'a' is not defined |

見事にエラー(NameError)が出ました。この部分にエラーの詳細が記述されています。“name 'a' is not defined”、つまりaは定義されてませんってことですね。エラーメッセージも理解しておくと結構便利です。プログラミングってエラーとの闘争ですので。

* 1. 色んな型

これはPythonに限った話ではなく、基本的にプログラミング言語ってのは“型”を持っています。型ってのは、変数の形です。「こいつはこんな値やで」って決めるものですね。

とりあえず、Pythonでよく使う標準の型を下の一覧でまとめておきます。(詳しいことは後々)



　こんなもんですかね。とりあえず今は「ふーん」って思っといてください。なぜ型が存在するかというと、処理方法が違うってだけ。何となく整数と文字列を同じ扱いしたらヤバそうですよね。ごっちゃごちゃだ。

　例えば、整数同士は足し算ができますが、整数と文字列は足し算できません。型が違うと、演算ができないんですね。

* 1. Pythonにおける型の扱い

じゃあ、Pythonと他の言語に違いはないのか。そんなこともないです。Pythonってのは動的型付け言語なので、型の宣言が要らないんです。C言語とPythonを比べてみましょう。

＜C言語＞

|  |
| --- |
| int a = 0;  int b = 4;  int c;  c = a + b; |

＜Python＞

|  |
| --- |
| a = 0  b = 4  c = a + b |

結果、cはおんなじ値4(int)になってます。C言語だと変数を作る時にいちいちintって型を宣言しないといけないです。一方、Pythonは=を付けるだけで、勝手に型を設定してくれるのです。めっちゃ便利。

「実はfloatの方が良かった、、」って場合も大丈夫です。

|  |
| --- |
| a = 0  a = float(a) |

float()ってしてやれば、ちゃんとaはfloat型(0.0)になってます。逆はint()でおっけー。intにする時は、確か四捨五入処理になるはずです、たぶん。(3.6→4, 3.4→3)

他にも、文字列を結合させたいとき。

|  |
| --- |
| a = 21  b = “点”  c = str(a) + b |

これでcは“21点”になります。“c = a + b”ってするとエラー(TypeError)になるので注意です。

忘れてましたが、””で囲むとstr型(文字列)になります。上のbとかそうですね。数字でも文字列であることがあるので注意。

* 1. 四則演算(+, -, \*, /, その他)

ここからは基本の計算のお話です。まずはザクっとまとめちゃいますね。具体例は後ほど。



こんなもんでしょうか。掛け算の記号はアスタリスクだったり、スターだったり呼ばれています。日本語配列キーボードであれば、“:”のキーにありますね。では、型を交えて演算していきましょう！

＜int/float型＞

|  |
| --- |
| a, b = 8, 4.3  print(a+b)  print(a-b)  print(a\*b)  print(a/b) |
| 12.3  3.7  34.4  1.8604651162790697 |

普通に演算できます。intとfloatはほとんど区別することなく使えます。ただし、余り(%)の時はint同士じゃないと使えません。

|  |
| --- |
| a, b = 8, 3  print(a%b) |
| 2 |

＜str型＞

|  |
| --- |
| a, b = “そら”, “あおい”  print(a+b) |
| そらあおい |

str型でも正しく演算できるものもあります。足し算(+)だけです。

引き算(-)もいけそうな気がしますが、ダメでした。(経験済み)　文字列同士を引き算するときは、もうちょっと別の方法を考えないといけないですねー。

1. ごめん、結構ごっそりまとめる
   1. Pythonのインデント(Tabキー)

ちょっとずつコードの話に入っていきます。C言語では“;”や豊富な括弧((){})により、どれだけぐちゃぐちゃに書いてもエラーが出ません。やべー言語よな。

逆にPythonではインデント(Tabキー)による空白で、段落分けされており、正しくインデントしていないとおかしな挙動になります。

簡単な例を見ていきます。

|  |
| --- |
| a = 1  b = 2 |

aとbは明らかにズレていますね。これはエラー(IndentationError)が出ます。ってか見た目悪いしね。映えは大事。

次に、エラーは出ないけど、おかしくなるコード。print()とfor文を使うので「print/forとか初耳！」って方は、今、先取りで 5.5 / 5.8の＜指定回数ループさせるとき＞ を読んできてください。

|  |
| --- |
| a = 0  b = 1  for i in range(2):  print(a)  print(b) |
| 0  1  0  1 |

|  |
| --- |
| a = 0  b = 1  for i in range(2):  print(a)  print(b) |
| 0  0  1 |

上のコードではprint(a)とprint(b)が一段インデントされています。よって２つともfor文の中に入っているので、a, b２回ずつ出力されてますねー。

一方、下のコードではprint(b)はインデントされてないです。なので、for文に入ってるのはaだけとなり、bはforが終わった後に一回だけ出力されています。

こーんな感じで、機能性と見栄えを両立させたのがPythonです。ちなみに先に紹介したJupyterNotebook, VScode, GoogleColabなどのエディタは基本的に、「:」を打って改行すると自動で一段インデントしてくれます。便利ですね。

* 1. リスト！

どんどん便利ツールを増やしていこうね。リストのお話。他言語では配列って言ったりもします。ちょっと訳が違う気もします。んん。

さっき変数の話をした時、変数名に対して、値は１個でした。つまり、値1つに対して変数名が1つ要るということ。めんどい。ってな時！！　便利なのがリストなのです。

リストはいくつかの値をまとめて1つの変数名にしてくれます。

作り方は簡単。

|  |
| --- |
| a = [0, 1, 2, 3] |

こんな風に、“,”で区切って“[]”で囲むだけ！

便利すぎるリスト、ちょっとだけその機能を見ていきましょう。

３

２

１

０

|  |
| --- |
| a = [0, 1, 2, 3]  インデックス  b = a[2]  print(b) |
| 2 |

リストの要素を取り出すには、“リスト名[インデックス]”で取り出せます。インデックスとは“何番目”って意味です。pythonでは0番目から始まります。なので上の例で“インデックス2”ってのは三つ目の要素を表しており、“3”が取り出されます。ちょっとややこしい。

では、こんなリストをちょっとだけ操作していきましょう。

|  |
| --- |
| a = [0, 1, 2, 3]  a.append(8)  print(a) |
| [0, 1, 2, 3, 8] |

上のように“リスト名.append(追加要素)”で要素を追加できます。

|  |
| --- |
| a = [0, 1, 2, 3]  a.clear()  print(a) |
| [] |

こんなん使うんかって感じですが、“リスト名.clear()”でリストの中身を空にすることができます。

|  |
| --- |
| a = [0, 1, 2, 3]  a.pop(1)  print(a) |
| [0, 2, 3] |

“リスト名.pop(インデックス番号)”で何番目の要素を消すことができます。上の例めちゃめちゃ分かりづらいな。

|  |
| --- |
| a = [“わん”, “ツー”, “さん”, “し”]  a.pop(1)  インデックス1はここ  print(a) |
| [“わん”, “さん”, “し”] |

最後にリストの長さを出しましょう。

|  |
| --- |
| a = [“わん”, “ツー”, “さん”, “し”]  print(len(a)) |
| 4 |

len(リスト)でリストの要素の数を出してくれます。ほんとによーお世話になります。rangeと組み合わせるのが多いかな。

そんな感じで扱っていくのがリストになります。他にも、並び替えしたり、最大値を出したりと、ある程度の機能はありますが最低限って感じがします。

* 1. 辞書も便利

さて、リストは取り出し方がインデックス(0,1,2…)でした。番号にそれほど意味がないなら良いけど、“この数字に対して、この要素が良い！”とかだと困ります。そこで使うのが辞書型です。

|  |
| --- |
| 辞書名 = {キー: 要素} |

定義の仕方はこんな感じ。リストと違い、“{}”になっています。また、キーと要素が入っていますね。どのように使っていくのでしょうか。

|  |
| --- |
| a = {“春”: “4月”, “夏”: “8月”,  “秋”: “10月”, “冬”: “2月”}  ここの要素を取り出す  print(a[“春”]) |
| 4月 |

最近は四季があいまいまいですけどね……。さて、先ほどのリストと同じように取り出しているんですけど、[]の中がインデックスではなくキーになっています。ね、すごく便利じゃない？

ただし、辞書に順番はありません。なんか勝手にごちゃごちゃになってたりします。それでもキーを渡せば、要素を返してくれるので結構重宝しています。

ちなみに辞書に追加するのはめっちゃ簡単。

|  |
| --- |
| a[“晩秋”] = “11月”  print(a) |
| {“春”: “4月”, “夏”: “8月”,  “秋”: “10月”, “冬”: “2月”, “晩秋”: “11月”} |

上のように、“辞書[追加するキー] = 追加する要素”とするだけです。appendとか要らんのね。

辞書にも色んな機能がありますが、ひとまずこれで十分なので必要に応じて検索してみてください。(丸投げ)

* 1. 関数(ちょっとだけ)

後のセクションでのんびり話すんですけど、下でちょっと使わなきゃいけないので、軽く言っておきます。

関数って、“貰ったものに対して何か処理をする機械”です。例えば、一番使う関数、print()さん。もらったものを出力部分に出してくれます。逆に、print()を使わないと出力部分には何も出てくれません。これ以外ほぼないんだわ。displayとかあるけど知らん知らん。

|  |
| --- |
| a = 555120  b = “残高(希望)”  print(a)  print(b) |
| 555120  残高(希望) |

int型もstr型も変わらずそのまま出力してくれます。C言語だとそうもいかないので、やっぱPython神。

|  |
| --- |
| a = 555120  print(type(a)) |
| <class 'int'> |

ついでにこの関数type()もご紹介。貰った変数の型を返してくれます。今aはint型なので、class 'int'と出力されてますね。

さて、関数はこんなもんですね。一方、Pythonにはメソッドという概念もあったりします。“似てるけど、どこか違う、だけど同じ匂い……”まさにそんな感じです。その辺は、また後々説明するので、どうか耐えてください。皆さんのスルースキルが試されます。

* 1. 条件(真 と 偽)

条件のお話です。数学っぽくて嫌いです。4.1の表にいらっしゃるbool型のことです。bool型は、TrueかFalseの2つしかありません。何がTrueで何がFalseなのか。のんびり見ていきましょう。

|  |
| --- |
| a = 1  b = 2 |

とした時、aとbは違う値です。なので、

|  |
| --- |
| print(bool(a == b)) |
| False |

と出力されます。bool()は()内の条件が真ならTrue、偽ならFalseを返してくれます。“a == b”というのが条件にあたります。a=1, b=2でa == bとはならないので、Falseなんですね。“==”の親戚はいっぱいいるので、一覧にしておきます。“>=”を“=>”と書くとエラー(invalid syntax)が出るので気をつけてください。



* 1. 分岐処理(if / else if / else)

いわゆるif文ってやつですね。

|  |
| --- |
| if 条件1:  条件1が真なら実行  elif 条件2:  条件2が真なら実行  else:  どの条件にも当てはまらなかったら実行 |

こんな感じ。上の例では条件が2つだけでしたが、elifを続けていくことで、いくらでも条件は増やせます。もちろん、elifやelseは省いても大丈夫です。

てきとーにこんな例を。

|  |
| --- |
| a = 846  if (a % 2 == 0) and (a % 3 == 0):  print(“aは2の倍数かつ3の倍数”)  elif (a % 2 == 0) or (a % 3 == 0):  print(“2か3で割り切れる”)  else:  print(“それ以外”) |

aが2で割り切れるか、3で割り切れるかの話です。and, orは日本語で“かつ”と“または”にあたります。aの値を変えてどの条件に当てはまるのか試してみてください。(俺はメンドイ)

僕がよく使う条件式は「リストの要素が無い」ですが、これはこんな感じで書けます。

|  |
| --- |
| a = []  if not a:  a.append(“test”)  print(a)  else:  print(a) |
| ['test'] |

リストのところでは説明しませんでしたが、リストは“要素があると真(True)”、“要素が無いと偽(False)”を出してくれるんですね。よって、「要素が無い」というのを条件にするには、notを付けてやればうまく判定できます。リストが空だと困ることが多いので、結構よく使います。

* 1. ループ処理(for / while)

偉大な偉大なループ処理。繰り返して行いたい処理を実行するときに多いに役立ちます。そんなループ処理、書き方自体は簡単です。

|  |
| --- |
| for 変数 in イテレータ:  ループさせたい処理 |

とまぁ、書いてみたもののなかなかややこしいところがありますね、、。イテレータってのは“要素を持つもの”って考えるのが一番簡単です。リストもイテレータの一種なので、イテレータ＝リストって捉えてもいいと思います。

for文はイテレータの最初の要素から順に、変数に代入してくれます。for文の中でその変数を用いて処理をすることが多いですね。

実例を用いつつ説明していきます。

<リストをループさせるとき>

|  |
| --- |
| my\_list = [“ワン”, “ツー”, “さん”, “し”]  for i in my\_list:  print(i) |
| ワン  ツー  さん  し |

リストは簡単ですね。要素を最初からそのままiに代入してくれます。

<指定回数ループさせるとき>

|  |
| --- |
| for i in range(4):  print(i) |
| 0  1  2  3 |

上の例は4回ループさせるときの使い方です。rangeって関数に回数を入れてあげるだけです。「for i in 4:」ってダメなの？ってなりますが、ダメなんです。4はint型なのでイテレータじゃないんです。range関数は4(int)に対して[0, 1, 2, 3]っていうイテレータを作ってくれるんです。なので、出力が“0, 1, 2, 3”になってますね。ちなみに0～3の4個であって、4は出力されません。

<辞書をループさせるとき>

|  |
| --- |
| my\_dict = {1: “ワン”, 2: “ツー”, 3: “さん”, 4: “し”}  for i in my\_dict:  print(i) |
| 1  2  3  4 |

これが辞書を単純にループさせたときの挙動になります。……うん、なにこれ。そうなんです。辞書をそのままforに突っ込むとキーだけが出てくるんです。まぁ、変数一個だし、キーか要素だよねって話。じゃあ、次！！

<辞書をループさせるとき②>

|  |
| --- |
| my\_dict = {1: “ワン”, 2: “ツー”, 3: “さん”, 4: “し”}  for i in my\_dict.keys():  print(i) |
| 1  2  3  4 |

無意味！！！

はい、.keys()メソッドでキーが取り出せるんですけど……for文で使うなら要らないっすね。

<辞書をループさせるとき③>

|  |
| --- |
| my\_dict = {1: “ワン”, 2: “ツー”, 3: “さん”, 4: “し”}  for i in my\_dict.values():  print(i) |
| ワン  ツー  さん  し |

これはまぁそこそこ使うかな、.values()メソッドで辞書の要素を取り出すことができます。よく“s”を忘れて怒られます、、。

<辞書をループさせるとき④>

|  |
| --- |
| my\_dict = {1: “ワン”, 2: “ツー”, 3: “さん”, 4: “し”}  for key, value in my\_dict.items():  print(key, value) |
| 1 ワン  2 ツー  3 さん  4 し |

最後に、これがいっっっちばん使う。.items()メソッドを使うと、2変数(key, value)に分解して、キーと要素を取り出してくれます。

(whileを使う)

ごめんなさい、完全に忘れてました。基本的にforを使えば何とかなるからあんまり使わないですが、whileを使う方がいい時もあります。while文の書き方も簡単です。

|  |
| --- |
| while 条件式:  ループさせたい処理 |

条件式が真であればループ、偽であればループしない。シンプルですね。変数がない分、楽といえば楽です。

* 1. <発展>ループ処理(enumerate)

発展としているけど、めっちゃ使います。enumerateです。普段は予測変換で出しているのでスペル間違っていないかビビっております。

さて、どうやって使うかだけ書きますね。

|  |
| --- |
| my\_list = [“ワン”, “ツー”, “さん”, “し”]  for index, value in enumerate(my\_list):  print(index, value) |
| 0 ワン  1 ツー  3 さん  4 し |

こんな感じ。for文に使うんですけど、なにやら変数が増えております。indexってやつですね。enumerateはリストに対して、インデックスと要素を出してくれるんです。便利なんだこれが。

そんなこんなで、優秀なfor君のお話は終わり！次からは関数の話へと移っていきます。

1. 関数作ろー！
   1. 関数の構造

関数の話第２弾！　シリーズものは大体2作目が駄作と言われますね。5.8で関数自体の説明は終わりで、今回は関数をワクワクさんの如く実際に作っていきたいと思います。

関数ってのは、引数と戻り値(返り値)を持っています。引数をもらって、戻り値を返してくれるんです。イメージはこんな感じ。

……分かった？？　いや……うん、自分でも絶望してます。もう説明とか難しいから、実際に作ってみましょう！(やけくそ)

* 1. 実際に作ってみよー

作り方自体はめっちゃ簡単。

|  |
| --- |
| def 関数の名前(引数):  ～なんかの処理～  return 戻り値 |

引数や戻り値は１個とは限りません。何個かある時もあるし、１個もない時もある、割とめっちゃ自由です。

さて、まず手始めに、2つの値の合計を返す関数を作ってみましょう。2つの値だから、引数は2つですね。で、戻り値は1つ、と。

|  |
| --- |
| def sum(a, b):  c = a + b  return c  if \_\_name\_\_ == “\_\_main\_\_”:  sum\_value = sum(2, 3)  print(sum\_value) |
| 5 |

“if \_\_name\_\_ == “\_\_main\_\_”:”は定義部分と実行部分を分かりやすくするためのものです。気にしないでください、おまじないです。なくても大丈夫です、こういう書き方もあるよってだけ。

薄い黄色の部分で関数の定義(def～return)をして、薄い青を実行しています。今さらですが、プログラミングには定義するところと実行するところがあります。基本的に実行部分を順に処理していくのですが、そこで関数などが出てきたとき、定義するところに戻って実行するという流れになります。まぁ、なんとなくで。

コードを読むときは、まず実行部分を読んでいって関数があれば定義部分を読んでください。ちなみに、絶対実行部分よりも前に定義部分はあります。(変数と同じですね)

話を戻しまして、“def”が関数を作るよっていう定義。“sum”は関数名(自分で決める)、a,bは引数ですね。最後に“return”で戻り値を返しています。出力されるのはcになります。なんとなーくわかりますかね？

もう少し例を出しましょうか。リストを引数にして、中身を全部2乗したリストを返す関数を作ってみましょう。

|  |
| --- |
| def square(input\_list):  square\_list = []  for value in input\_list:  square\_value = value \*\* 2  square\_list.append(square\_value)  return square\_list  if \_\_name\_\_ == “\_\_main\_\_”:  my\_list = [0, 2, 5, 6]  new\_list = square(my\_list)  print(new\_list) |
| [0, 4, 25, 36] |

結構長くなりましたね。まず、my\_listが定義されています。それが、squareって関数に入れられてnew\_listが出力されてますね。じゃあsquareってなんだって話。これは定義部分を見るとinput\_listが処理されているのが分かります。

for文の使い方、覚えてますか？　リストはそのまま取り出してくれるんですよね。input\_listの0から順に2乗(\*\*2)されて、square\_listに追加(.append())されています。のんびり読めば大したことはないです。

* 1. 軽－く練習問題

さて、ちょっと慣れてもらうために、下の要件を満たす関数を作ってもらいます。ちなみに解説する気はあんまりないです。めんどい。

次のページに僕のコードを載せるので、頑張って！

|  |
| --- |
| ＜要件＞  関数名: create\_list  引数: intのリスト, 基準値(int)  戻り値: intのリスト  処理: 渡されたリストの要素に対して、基準値よりも小さければ2倍した値を、それ以外なら元の値を入れたリストを返してください。 |

|  |
| --- |
| def create\_list(input\_list, criteria):  new\_list = []  for value in input\_list:  if value < criteria:  new\_list.append(value \* 2)  else:  new\_list.append(value)  return new\_list  if \_\_name\_\_ == “\_\_main\_\_”:  list\_1 = [0, 5, 2, 8]  list\_2 = [10, 2, 7. 21]  print(create\_list(list\_1, 3))  print(create\_list(list\_2, 9)) |
| [0, 5, 4, 8]  [10, 4, 14. 21] |

コードは実行部分から読みます。

まず、list\_1とlist\_2が定義されています。一つ目のprintでは、list\_1に対して、基準値を3として関数create\_listに入れられてます。そこでcreate\_listの定義を見てみましょう。

create\_listはinput\_list, criteriaを引数として処理を行っていますね。戻り値とするnew\_listを作って、要件を満たすように値を追加していきます。

for文でinput\_listから値を取り出し、if文で「criteriaより小さければ2倍した値、それ以外で元の値」をnew\_listに追加(.append())していってますね。

for文が全て実行出来たら、new\_listの完成です！　returnで返してあげましょう。返ってきたリストはprintで出力されています。([0, 5, 4, 8])

2つ目のprintは同じ処理をlist\_2, 基準値9で行っているだけです。ちゃんと想定通りの出力が出てますね。([10, 4, 14. 21])

もう一問ぐらいいっときますか！　いけるっしょ！　さっきと同じように下の要件を満たすような関数を作ってください。ちょっとだけレベルアップした問題です。

|  |
| --- |
| ＜要件＞  関数名: fizz\_buzz  引数: 辞書(Dict[int: str]), 値(int)  戻り値: 文字列(str)  処理: 入力される値に対して、辞書のキーで割りきることができる時の辞書の要素をつなげて文字列を返してください。どのキーでも割り切れない時は“割り切れない”を返してください。 |

FizzBuzzってやつです。

例えば、{2: “Fizz”, 3: “Buzz”}と12を入力したとき、12は2でも3でも割り切れるので、“FizzBuzz”を返してほしいです。

|  |
| --- |
| def fizz\_buss(dictionary, num):  string = “”  for key, value in dictionary.items():  if num % key == 0:  string += value  if string == “”:  string = “割り切れない”  return string |

こんな感じでしょうか。まず、戻り値用のstringは空白(“”)にしておきます。

そこから、dictionaryのキーと要素をfor文で回します。もし、値(num)がキーで割り切れるのであれば、stringに要素(value)を足してあげます。

for文が終わった時、どのキーでも割り切れなかったらstringは“”のままです。その時は、stringに“割り切れない”を入れて返してあげます。

まぁ、そこそこ色んな要素が入ってるのかなと思います。僕が作った問題じゃないですけど笑

1. クラスって、、？メソッドと関数、、？？
   1. クラス・インスタンスという概念(わたあめ)

Pythonを学ぶにあたって、一番つまずく……というか一番使い方が分からない概念が“クラス”です。ぶっちゃけ、これが一発で理解できるなら、もうさっさとどっか入社してください、即戦力っす。

　まず、クラスとは何なのか。んー……“情報を持った設計図”？かな。これも現実世界を例にとって、スマホ。みんなのスマホには皆さんそれぞれ異なった個人情報が入っています。でもGoogle検索やYoutubeなど基本的な機能は同じです。持ってる情報は違うけど、機能は同じように設計されてます。これがクラスの基本的な概念です。

　そして、ややこしい言葉がもう一つ。インスタンスってやつです。一言で言うなら、“クラスが入った変数”です。んにゃー分かりづらい。さっきの例でいくと、スマホの設計図がクラス、皆さんそれぞれのスマホ本体がインスタンスですね。

もっと単純にわたあめに例えてみようと思います。

　わたあめって、わたあめ機の中では何も見えないじゃないですか？　でも、棒を入れてクルクルするとわたあめが出来てくる。

こんな感じです。わたあめ機がクラス、わたあめがインスタンスです。わたあめの質・味・色を決めるわたあめ機を使って、わたあめを作る。クラスってのは要は設計図なんですね。

* 1. ひとまずクラスを作ってインスタンス化

さて、ある程度の理解で進んでください。実際にクラスを作ってみないと何とも言えないっすよね。クラスの作り方はいたって簡単。

|  |
| --- |
| class クラス名: |

これで終わり。簡単でしょ。

では、実例を見ていきましょう。せっかくなので、さっきのわたあめ(CottonCandy)クラスとそれをインスタンス化したものを作ろうと思います。

|  |
| --- |
| class CottonCandy:  name = “わたあめ”  flavor = “イチゴ”  color = “赤”  if \_\_name\_\_ == “\_\_main\_\_”:  candy = CottonCandy() |

なんか関数の使い方と似てますね。クラスとインスタンスの違いは分かるでしょうか。クラスはCottonCandy、インスタンスはcandyです。インスタンス化されたcandyはフレーバーや色などのわたあめの色々な情報を持っています。

それを取り出すには以下のように“.～”ってしてやればおけです。

|  |
| --- |
| print(candy.name)  print(candy.flavor) |
| わたあめ  イチゴ |

candyの名前(name)とフレーバー(flavor)が取り出せました！

暇なら、色(color)も取り出してみてください。

* 1. \_\_init\_\_なにそれおいしいの？

しかしここで問題が。先ほどのCottonCandyクラスを使ってもっとインスタンスを作ってみましょう。

|  |
| --- |
| class CottonCandy:  name = “わたあめ”  flavor = “イチゴ”  color = “赤”  if \_\_name\_\_ == “\_\_main\_\_”:  candy\_1 = CottonCandy()  candy\_2 = CottonCandy()  print(candy\_1.flavor)  print(candy\_2.flavor) |
| イチゴ  イチゴ |

なんと、candy\_1もcandy\_2も同じ味。こんな店やだ。

つまりインスタンスごとに値を変えたいのです。それを解決してくれるのが\_\_init\_\_関数です。（めんどいのでflavorだけ）

|  |
| --- |
| class CottonCandy:  def \_\_init\_\_(self, flavor):  self.flavor = flavor  if \_\_name\_\_ == “\_\_main\_\_”:  candy\_1 = CottonCandy(“イチゴ”)  candy\_2 = CottonCandy(“メロン”)  print(candy\_1.flavor)  print(candy\_2.flavor) |
| イチゴ  メロン |

……かき氷屋じゃね？　まぁいいか。

\_\_init\_\_関数(これは名前変えないでね)は、インスタンスが作られる時に実行される関数です。その際、引数をもらうこともできます。先の例では、CottonCandy(“イチゴ”)と引数が渡されていて、self.flavorに“イチゴ”が代入されてます。

ここからがクラスの厄介なところ。\_\_init\_\_関数を見てみると、“def \_\_init\_\_(self, flavor):”と2つの引数で定義されています。なのに、使うときは“CottonCandy(“イチゴ”)”と1つしか引数がありません。selfどこいった？？

このself、実はインスタンス自身なのです。なんかよく分からんよね。例えば、さっきの例のdef \_\_init\_\_内部にあるself.flavor、これはcandy\_1の時はcandy\_1.flavor、candy\_2の時はcandy\_2.flavorを表しています。selfってのは、“インスタンスそのもの”なんです。

難しく考えないでください笑

* 1. クラス変数、インスタンス変数

さて、ここまでくると、説明しなくても何となくわかるかもですね。まだまだこき使うCottonCandy。

|  |
| --- |
| class CottonCandy:  name = “わたあめ”  def \_\_init\_\_(self, flavor, color):  self.flavor = flavor  self.color = color  if \_\_name\_\_ == “\_\_main\_\_”:  candy\_1 = CottonCandy(“イチゴ”, “赤”)  candy\_2 = CottonCandy(“メロン”, “緑”) |

　name, flavor, colorそれぞれ「クラス変数」と「インスタンス変数」どっちっすかね？　まぁ何となくで、nameがクラス変数、flavor, colorがインスタンス変数です。

　クラス変数 : どのインスタンスでも一緒。

　　インスタンス変数 : インスタンスごとに違う。

　先の例でいくと、candy\_1.nameとcandy\_2.nameは同じ“わたあめ”ですが、candy\_1.flavorとcandy\_2.flavorは違う値になります。

　たまーに使い分けることがあります。……考えてみたらよくありました。失敬。

* 1. メソッドと関数って何が違う

さらに新要素が増えていきます。メソッドという概念です。メソッドを一言でいうと、“クラス内に定義する関数”です。しかし、Pythonの教科書やwebサイトでは、メソッドと関数はちょっとだけ区別されています。結局一緒なのにね。

前座は置いといて、さっそく作ってみましょう。わたあめの味変に挑戦してみようと思います。

|  |
| --- |
| class CottonCandy:  def \_\_init\_\_(self, flavor, color):  self.flavor = flavor  self.color = color  def change\_flavor(self, new\_flavor):  self.flavor = new\_flavor  if \_\_name\_\_ == “\_\_main\_\_”:  candy\_1 = CottonCandy(“イチゴ”, “赤”)  print(candy\_1.flavor)  candy\_1.change\_flavor(“ぶどう”)  print(candy\_1.flavor) |
| イチゴ  ぶどう |

メソッドの作り方はこれまたとっても簡単。関数と一緒で“def 関数名”とするだけです。ただし、\_\_init\_\_と同じで、引数にselfを入れるのを忘れないようにしてください。(selfを使わない場合は、static methodと呼ばれます)

出力の一行目は、candy\_1を定義した直後なので、flavorは“イチゴ”となります。その後、candy\_1.change\_flavorでクラス内に定義されたchange\_flavorが実行されることによって、flavorが“ぶどう”に変化します。ってなわけで出力の二行目は“ぶどう”になるわけです。  
　ん、なんとなくでいけるっしょ。いけるいける！

……とはいえ、ちょっと練習してみようか。どんなクラス作ろうかな、、。じゃあ下の図のクラスを作っていきましょう。

メソッド

インスタンス変数

クラス名

|  |
| --- |
| NagasiSomen |
| flavor |
| add\_somen()  change\_flavor(new\_flavor)  somen\_attack(target) |

これはクラス図と呼ばれるものです。実務ではこれが何個もあって、継承したり抽象化したりとそれぞれ繋がれています。

夏が懐かしいので流しそうめんでもしましょうか。クラス名をNagasiSomenとして、インスタンス変数にflavor(味)、メソッドにadd\_somen(), change\_flavor(new\_flavor)……somen\_attack(target)を以下の要件を満たすように作っていきましょう！

|  |
| --- |
| ＜要件(add\_somen())＞  処理: 現在の味のそうめんを流すよう出力してください。  　　　“(flavor)のそうめんを流したよ”  　　　(例)“よもぎのそうめんを流したよ” |
| ＜要件(change\_flavor(new\_flavor))＞  引数: 新しい味  処理: self.flavorをnew\_flavorに変えてください。 |
| ＜要件(somen\_attack(target))＞  処理: 現在の味のそうめんで、targetに攻撃してください。  　　　“(flavor)のそうめんで(target)に攻撃”  　　　(例)“サクラのそうめんで市長に攻撃” |

|  |
| --- |
| class NagasiSomen:  def \_\_init\_\_(self, flavor):  self.flavor = flavor  def add\_somen(self):  print(self.flavor +“のそうめんを流したよ”)  def change\_flavor(self, new\_flavor):  self.flavor = new\_flavor  def somen\_attack(self, target)  print(self.flavor +“のそうめんで”+ target +“に攻撃”) |

物騒な流しそうめんっすね。クラス自体はこれでいいでしょう。selfをうまく使ってメソッドを実装してください。

ではでは実際に流してみますか。

|  |
| --- |
| somen = NagasiSomen(“普通”)  somen.add\_somen() |
| 普通のそうめんを流したよ |

うんうん、普通のそうめんが流れたみたいです。\_\_init\_\_の引数にflavorがあるので、インスタンスを作る時の引数が最初の味になります。もうちょっと遊んでみる。(上の続きね)

|  |
| --- |
| somen.change\_flavor(“ゴマ”)  somen.somen\_attack(“アザラシ”) |
| ゴマのそうめんでアザラシに攻撃 |

アザラシーーーー！！！　somenを作った時は、flavorは“普通”でしたが、change\_flavorで“ゴマ”に変わっています。最後にsomen\_attackをすると、“ゴマ”アタックになるわけですね。

* 1. プライベート変数

クラス変数、インスタンス変数、メソッド……など様々なクラスの要素を扱ってきました。しかし、中には外部から呼び出されたくないものも存在します。誰だって見られたくない１つや２つあることでしょう、そんな感じのやつがプライベート変数です。

個人情報の塊、スマホをクラスとして例を作ってみましょう。

|  |
| --- |
| class SmartPhone:  model = “iPhone”  \_\_tel\_number = “08022221111” |

クラス変数なのはちょっと変だけど、めんどうなので、、。

機種(model)は他人に見られても大丈夫です。別にいいよね。でも電話番号(tel\_number)は困ります。そんな時に使うのが“\_”(アンダーバー)です！見にくいけど、2本あります。

変数の前に2本アンダーバーを入れると、プライベート変数となり、クラス外からの呼び出しが出来なくなります。試しにやってみましょう。

|  |
| --- |
| my\_smart\_phone = SmartPhone()  print(my\_smart\_phone.model)  print(my\_smart\_phone.\_\_tel\_number) |
| iPhone  AttributeError Traceback (most recent call last)  ～中略～  AttributeError: 'SmartPhone' object has no attribute '\_\_tel\_number' |

ね、電話番号をとろうとすると、AttributeErrorになります。インスタンス変数やメソッドも同様にアンダーバー2本でプライベートにできます。

Pythonでは一応こうやってプライベート変数を実装します。

(※)実際には、無理矢理引き出すことが可能です。気を付けてね。

* 1. <発展>デコレータ(@staticmethod, @property, etc.)

発展と言いつつ基本的なものは割と使いますデコレータ。言葉で言うのがめちゃムズいランキング堂々の１位です。一言で言うなら、、“関数をデコレートしてくれるもの”、、。

はい例にいきましょう。

|  |
| --- |
| class Dentaku:  def add(self, a, b):  return a+b  def sub(self, a, b):  return a-b |

電卓クラスを用意しました。足し算と引き算のメソッドが実装されてますね。いやでも待てよ、と。addもsubも引数にself入れてるけど、self別に使わなくね？ そーなんすよ。self要りません。

でもself消したらダメだしな、、、ってことでデコレータ@staticmethodの出番です。(足し算だけやります、余白の都合上)

|  |
| --- |
| class Dentaku:  @staticmethod  def add(a, b):  return a+b |

見事にselfが消せました！ ﾊﾟﾁﾊﾟﾁﾊﾟﾁｰ

@staticmethodを使うと、インスタンスを作る必要がなくなります。実際にDentakuを使ってみましょう。

|  |
| --- |
| print(Dentaku.add(8, 5)) |
| 13 |

ちょっと簡単になりますね、これは嬉しい。

@staticmethodがないと、dentaku=Dentaku()ってインスタンス作って、dentaku.add(8, 5)になりますかね。１行で書くならDentaku().add(8, 5)かな。ちょっと気持ち悪いけど笑

ここがインスタンスになってるよ

他にはなにがあるかな。@propertyとかでしょうか。先のプライベート変数、クラス外から読み出しできないってこと以外にもう一つ別の役割があって、それが“クラス外から変更させない”ことです。

どういうことでしょうか。例を見てみましょう。

|  |
| --- |
| class SmartPhone:  model = “iPhone” |

さて、クラス変数をmodelだけにしました。今の状況だと、modelに値を入れたらどうなるんでしょうか。

|  |
| --- |
| my\_smart\_phone = SmartPhone()  print(my\_smart\_phone.model)  my\_smart\_phone.model = “Android”  print(my\_smart\_phone.model) |
| iPhone  Android |

なんとAndroidに変わってしまいました。めっちゃ困る。

こんな時に使うのが@propertyデコレータです。クラスを書き直してみましょう。

|  |
| --- |
| class SmartPhone:  \_\_model = “iPhone”  @property  def model(self):  return self.\_\_model |

こんな感じで作っていきます。何が変わったか。.modelで実行される関数はこっちのmodelになります。すると返り値はself.\_\_modelなので“iPhone”が出力されます。ね。簡単でしょ？

じゃあ@propertyはなにをしてくれるんだろう。ここでさっきと同じように“Android”を代入してみましょう。

|  |
| --- |
| my\_smart\_phone.model = “Android” |
| AttributeError Traceback (most recent call last)  ～中略～  AttributeError: can't set attribute |

お、うまいこと機能してますね。@propertyを付けると、その変数に対しての代入を拒否してくれます。こうやって変更できない変数を作るんですね。

ちなみに@staticmethodや@propertyはPython標準のデコレータですが、自分でデコレータを作ることもできます。例えば、「関数実行の前には“-Start\_function-”って出したいなー」とか。いざ実装。

|  |
| --- |
| def start\_function(func):  def wrapper(\*args,\*\*kwargs):  print(“-Start Function-”)  phrase =“-End Function-”  func(\*args, \*\*kwargs)  return phrase  return wrapper |

こいつがデコレータになります。関数を受け取ってwrapper関数を返すっていう関数がデコレータの実態なんですね。wapper関数の返り値はfuncが実行されてから出力されます。まぁ返す意味ないけどね。

これを使ってみましょう。

|  |
| --- |
| @start\_function  def test(function\_name):  print(function\_name +“を実行した！”)  test(“test”) |
| -Start Function-  testを実行した！  -End Function- |

色がさっきのデコレータと対応しています。ます、wrapperのprintが実行されて、testが実行されて、返り値のphraseが出力されてるって感じですね。さすがにデコレータを自作する機会はそうないですね、、、。staticmethodとpropertyが使えれば十分かと。

* 1. <発展>継承

説明がクソめんどくさいなぁと思っている継承。頑張っていきましょう。何かを実装するときには、何個もクラスを作るわけですがそれぞれ全くの無関係ってことはあまりありません。例えばゲームで考えてみましょう。

色んなゲームがあるんですけど、シューティングゲーム・RPG・サバイバルゲームの３つにしようかな。このそれぞれ、まぁ違うゲームです。でも、歩く・ジャンプ・素材を集める、など共通する操作が存在します。分けて実装するとめんどうくさい。ってことで！それらをまとめちゃおう！ってのが継承です。イメージは下の感じー。

実際にコードにするとこんな感じです。

|  |
| --- |
| class Game:  def walk(self):  print(“歩く”)  def jump(self):  print(“ジャンプ”)  class ShootingGame(Game):  def shoot(self, target):  print(target + “を撃つ”)  class RPG(Game):  def talk(self, person):  print(person + “に話しかける”)  class SurvivalGame(Game):  def eat(self, food):  print(food + “を食べる”) |

　とまぁこんな感じです。“クラス名(継承したいクラス)”で継承ができます。さて、これで何が良くなったのか。

　さっきの例のように継承すると、継承したクラスのメソッドが使えるんです。ShootingGameクラスにwalkメソッドはないんですけど、Gameクラスを継承しているのでShootingGame.walk()は使えます。同じようにRPGやSurvivalGameでもwalkやjumpはできますね。一方で、それぞれ特有の動作は各々のクラスで定義しています。ShootingGameでは誰かを撃って、RPGでは誰かに話しかけて、SurvivalGameでは何かを食べて……。

　用語として、継承元を親クラス、継承先を子クラスと言います。ま、イメージのまんまですよね。

　継承。まとめられるようなメソッドがあれば、まとめちゃおうの精神です。めんどうなことに、そこそこよく使います、、。気が向けば、Enumっていう列挙型を継承して使ってみてください。便利です。

* 1. <発展>続・継承(super().\_\_init\_\_)

継承の話をしたとき、先の例では普通のメソッドのみを継承しました。「あれ？じゃあ継承元の\_\_init\_\_はどうなるの？」そう思った方は初学者じゃないです、ダウト。

その通り、ちゃんと\_\_init\_\_も継承できます。例を見ていこ。

|  |
| --- |
| class Game:  こいつが実行される！  def \_\_init\_\_(self, player\_name):  self.player\_name = player\_name  def walk(self):  print(self.player\_name +“が歩く”)  class ShootingGame(Game):  def \_\_init\_\_(self, player\_name, weapon):  super().\_\_init\_\_(player\_name)  self.weapon = weapon  def shoot(self):  print(self.player\_name + “が”+ self.weapon + “で撃つ”) |

“super().メソッド”で親クラスのメソッドを使うことができます。基本的にはsuper().\_\_init\_\_かな、他は使ったことないです笑

ShootingGameはGameを継承していて、\_\_init\_\_でplayer\_nameとweaponを入れています。ただ、self.player\_nameはGameで実装されているので、それを利用してやろうって魂胆です。

んん、、親の財産を子が利用する……。やめときましょう。

ちなみに、super()ってのはsuperクラスのインスタンスが作成されてたりします。不思議なクラスですね。

実際に使うとどうなるか。まぁ、さっきと特に変わりません。

|  |
| --- |
| game = ShootingGame(“いさお”, “R310”)  game.shoot() |
| いさおがR310で撃つ |

かっこいいですね、いさお。R310って怒られるかな……。

さて、gameインスタンスを作る時にplayer\_nameとweaponを入れました。それを使ってshoot()しています。

出力を見ると、ちゃんとself.player\_nameが入っているのが分かります。便利ですね。

クラスを使うと何がいいのか。今さらこんなところで記します。

クラスは情報を格納しておける場所なんです。手続き型プログラム、つまりdefのみを使って書くやり方だと、いちいち引数に情報を渡してあげなければなりません。だるい。それを一度クラスに格納して、selfで引っ張ってくればいいだけなので、単純に引数を減らせますし、工程もグッと抑えることができます。

初頭で述べた、オブジェクト指向プログラミング言語の神髄はクラスにあります。実は“オブジェクト=クラス”です。PythonはObjectってクラスを全てのクラスが継承している設計(Objectはtypeを継承)になっています。全てのクラスはオブジェクトなんですね。ちなみに、オブジェクトに対する操作、これはメソッドのことです笑

なーんか手のひらで踊らされた気分。かれこれ述べましたが、せっかくのオブジェクト指向プログラミング言語なので、ぜひクラスを使ってみては、というお話でございました。

* 1. <発展>抽象クラス/抽象メソッド

抽象クラス！？要らんだろこれは……。さて、先のセクションで継承のイメージは掴んでいただけたでしょうか。それを一段階発展させます。抽象クラス、といっても実際は継承クラスにabstract methodが追加されただけです。まぁ、完全に実務っぽい話になります。

多人数で開発する際、困るのがクラス設計のズレです。各々が勝手にクラスを作って勝手に実装するとめちゃくちゃになります。それをコード側に管理してもらおうってのが抽象クラスですね。人間が機械に管理されます、くろわろた。

abstract methodっていうのは、形だけ決まっていて実際の中身がないメソッドです。抽象クラスを継承して初めてメソッドが実装されます。具体例を作るの難しいんですけど、やってみましょう。

|  |
| --- |
| from abc import ABCMeta, abstractmethod  class NoodleCreator(metaclass=ABCMeta):  @abstractmethod  def create(self, flavor: str):  pass  class UdonCreator(NoodleCreator):  def create(self, flavor: str):  print(flavor+“味のうどんを作ったよ”)  class SobaCreator(NoodleCreator):  def create(self, flavor: str):  print(flavor+“味のソバを作ったよ”) |

さて、分かるでしょうか、、？　これ使いこなせたらすごいよねー…。またまた食物の話になりますが、NoodleCreatorが抽象クラスと呼ばれるクラスです。metaclassにABCMetaが設定されております。metaclassとは何ぞや、さすがに説明したくないんですけど、また後々書き足すかもしれません。今はスルーで。

さて、そして抽象メソッド(@abstractmethod)がcreateって関数ですね。createには引数の型(str)のみ指定されています。そこで継承したクラス(UdonCreator, SobaCreator)を見ると、同じような引数を持っているけどcreateの中身は違うんです。それが抽象メソッドです。

忘れてた。pythonでは使う変数の型を指定できるんです。次章にアノテーションという項目を載せてますので参照までに。

そんな感じで、引数と返り値の型だけ指定して、中身は自由に作っていいよってのが抽象メソッドですね。それが入ってるのが抽象クラスです。まぁ、意識高い系クラスってことで……。

* 1. <発展>Protected変数

ここまで来ましたか。とは言っても、抽象メソッドの概念が重すぎただけでProtected変数自体は軽いもんです。Private変数はプライベート変数って書いてもいいけど、プロテクテッド変数って…ダサい。

さて、Protected変数とはPrivate変数の親戚です。クラス外部からの呼び出しを禁止してくれそうです。んん、何が違うんでしょうか。Private変数と比較してみましょう。



比較するとこんな感じ。違うのは赤の部分ですね。Protected変数では継承クラスに○がついています。どういうことだ。

|  |
| --- |
| class Game:  def \_\_init\_\_(self, player\_name):  self.player\_name = player\_name  def walk(self):  print(self.player\_name +“が歩く”)  class ShootingGame(Game):  def \_\_init\_\_(self, player\_name, weapon):  super().\_\_init\_\_(player\_name)  self.weapon = weapon  def shoot(self):  print(self.player\_name + “が”+ self.weapon + “で撃つ”) |

さて、継承で使ったGameクラスですが、このままだとself.player\_nameに別の名前を代入できます。プライベート変数でやりましたね。

でもそれは困るんだ。と言っても、self.\_\_player\_nameとしてしまうとShootingGameでself.\_\_player\_nameが使えなくなってしまいます。んんんん、どうしたらいいんだーー！

それを解決するのが、Protected変数。使ってみましょう。

|  |
| --- |
| class Game:  def \_\_init\_\_(self, player\_name):  self.\_player\_name = player\_name  def walk(self):  print(self.player\_name +“が歩く”)  class ShootingGame(Game):  def \_\_init\_\_(self, player\_name, weapon):  super().\_\_init\_\_(player\_name)  self.weapon = weapon  def shoot(self):  print(self.\_player\_name +“が”+ self.weapon +“で撃つ”) |

これでよし！ Protected変数ってのは、アンダーバーを1本入れるだけです。これで、self.\_player\_nameは呼び出せなくなるはずです。

実際どうなっているんでしょう。

|  |
| --- |
| game = ShootingGame(“いさお”, “R310”)  game.shoot()  print(game.\_player\_name) |
| いさおがR310で撃つ  いさお |

……あれ？ はい、実はProtected変数はクラス外からも呼び出せてしまいます。“\_を付けたものは外からは呼び出さないでね”っていう注意喚起でしかないのです。なんてこったい。結局、上の表は大嘘だよっていう話でした笑(@property使おうね)

1. アノテーション
   1. アノテーションとは

アノテーション……個人で作業してる時の僕は聞いたこともありませんでした。抽象クラスのところで、“人間がコードに管理される”的なことを言いましたが、まさにそれです。

かれこれ適当にコードを書いておりますと、「こいつの引数と戻り値なんだっけ、、？忘れたわろた」みたいなことが多々……多々！あるわけですね。そこでアノテーションの出番。

|  |
| --- |
| def create\_list(input\_list, criteria):  ～中略～  return new\_list |

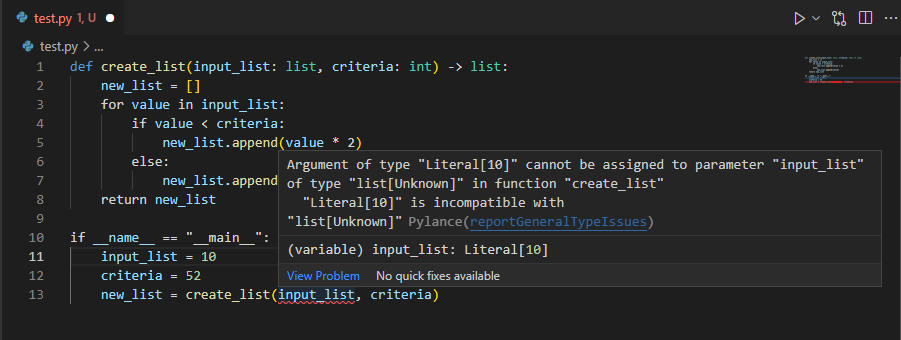
さて、6章のcreate\_listを借りてきました。ここにアノテーションを実装していきましょう！

|  |
| --- |
| def create\_list(input\_list: list, criteria: list) -> list:  ～中略～  return new\_list |

1行目になにやら追加されております。引数の型(list)と戻り値の方(list)ですね。こんな感じでアノテーションを付けることができます。

こうすることで何を入力して、何が出力されるのか分かりやすくなりました。違う型を入れてもエラーにはなりませんが、環境によっては警告されるのでミスにも気づきやすいです。

VScode(pylance)の画面を参照してみましょう。



ね、こんな感じで警告してくれます。ありがてー。

* 1. <発展>typing

さっきの例をもう一度。

|  |
| --- |
| def create\_list(input\_list: list, criteria: list) -> list:  ～中略～  return new\_list |

……listの中身ってなんなん？？　そうなんです、このアノテーションだけではlistの中身の型が分かりません。intなのかstrなのか……。それを解決するのがtypingライブラリです。

Pythonには標準でtypingっていうライブラリが入っています。さくっと使ってみましょう。

|  |
| --- |
| from typing import List  def create\_list(input\_list: List[int], criteria: List[int])  -> List[int]:  ～中略～  return new\_list |

どうでしょうか。Listをインポートして、アノテーションに使っています。[int]と中身がint型であることが一発で分かりますね。

typingは辞書にも使えるんです。また6章からお借りしましょう。

|  |
| --- |
| from typing import Dict  def fizz\_buss(dictionary: Dict[int, str], num: int) -> str:  ～中略～  return string |

分かりやすくていいねー。“Dict[キー, 要素]”ってなってます。

アノテーションをつけることで、AttributeErrorやTypeErrorなどのエラーを回避していくってのも重要だったりします。

しかし、アノテーションはあくまで目安。どうしてもエラーが出ることはあるので、それを回避するのは次章のお話。

* 1. <発展>Any, Union, Optional

もはやおまけです。typingには他にもいろいろあるよーっていう紹介だけさせていただきます。import方法はList, Dictと同じ。



主に使うとしたらこれくらいですかね。Anyとか絶対使わんけど笑

表に入れるのが難しかったので入れてませんが、Finalってやつもいます。Finalは一度定義した変数の再定義(代入)を禁止します。まぁ@propertyみたいなもんだと思ってもらえれば。

* 1. <発展>Generics

ガチの発展です。本当に読み飛ばしてほしい。僕も使いこなせているかビミョーなとこなので、、、。

ジェネリクス自体はPythonというよりjavaの概念ですね、静的型付け言語に使われるものだったりします。ジェネリクスとは、型を動的に決める、つまりプログラム上で型が決まるという状況に対して設定するものです。軽くだけやります。

特にクラスで多用するのですが、例えば、このインスタンスではこの型だけど、違うインスタンスではこっちの型っていう分け方をしたいときに使われます。軽くやってみましょう。

|  |
| --- |
| from typing import Generic, TypeVar  T = TypeVar(“T”)  class Adder(Generic[T]):  @staticmethod  def add(a: T, b: T) -> T:  return a+b |

@staticmethodっていうのはselfが要らないメソッドね。

TypeVarは型に名前を付けることができます。上のTってのは“T”型ってことですね。これを使ってアノテーションしていきます。

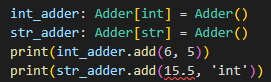
さて、Adderは.addでaとbを足してくれるクラスです。しかし、a, bは一緒の型じゃないと足し算できないですし、でもintってアノテーションするとint型しか足してくれません。これは困った。

ここで“T”型の出番です。実際に使ってみましょう。

|  |
| --- |
| int\_adder: Adder[int] = Adder()  str\_adder: Adder[str] = Adder() |

このようにint\_adderとstr\_adderを用意しました。int\_adderでは“T”型はint型になっています。str\_adderではstr型です。すると、addメソッドの引数はそれぞれint型、str型になるんですね。

実際に使って、VScodeで見てみましょう。



キチンと15.5(float型)にエラーが出ているのが分かります。ちなみにGoogleColabやノーマルのJupyterでは出てくれません。str\_adderでも普通にint型が入ってしまいます。無能。

2章ではやりませんでしたが、VScode+Jupyterっていう環境も作ることができます。すると、Jupyterでもアノテーションを利用できるようになるみたいです。無料なのに意外と便利ですVScode。

(※)GoogleColabでは型ヒントは使えないみたい……。そこだけなんよなマジで。

1. 例外処理って便利なんだよ
   1. tryの使い方

これは意外と便利、tryさんです。

pythonだけでなくプログラミングってのは日夜エラーとの闘いです。なんでかっていうと、関数にぶち込むものがずっと一緒だとは限らないからですね。ある時は数字を入れたり、ある時はリストを入れたり、ある時は辞書を入れたり……自由すぎる。

そんな時に活躍するのがtryさんです。みんなの味方。例の如く、書き方だけサクッとみてみましょう。

|  |
| --- |
| try:  エラーが出るかもしれない処理  except (エラーの内容):  エラーが出たら何するか。  else:  エラーが出なかったら何するか。  finally:  エラーが出ても出なくても行う処理 |

さて、何となく分かるでしょうか。tryの中にエラーが発生するかもしれない処理を書いて、あとはエラーの状況に応じて実行されるものになっています。ちなみに、elifと同じようにexceptは何個も並べることができます。

まぁ、てきとーに例を書いてみましょう。

* 1. 実際に使ってみる(AttributeError回避)

|  |
| --- |
| def tuika(some\_list):  some\_list.append(5)  return some\_list |
| a = [0, 5, 87, 7]  new\_a = tuika(a)  print(new\_a) |
| [0, 5, 87, 7, 5] |

リストに対して、5を追加する関数を用意しました。普通のリスト(a)を入れると、当然5が追加されて出力されます。普通にappendが使われてますね。

では、tuikaに数字(int)が入るとどうなるよ。

|  |
| --- |
| a = 85  new\_a = tuika(a) |
| Traceback (most recent call last):  ～中略～  AttributeError: 'int' object has no attribute 'append' |

しっかりエラーが出てきます。AttributeError、これは“そんなメソッドはないよー”ってエラーです。int型にappendというメソッドは存在しません。整数に何追加するんって話ですね。それがAttributeErrorです、こいつにはよく苦しめられます。

では、こいつを回避してみましょう。

|  |
| --- |
| a = 85  try:  new\_a = tuika(a)  except AttributeError:  エラー出る時  print(“リストじゃないよ！”)  else:  エラー出ない時  print(new\_a) |
| リストじゃないよ！ |

なんとエラーが出ませんでした。AttributeErrorが発生したので、print(“リストじゃないよ！”)が実行されてますね。ほなエラーが発生しない時はどうなるか。

|  |
| --- |
| a = [0, 5, 87, 7]  try:  new\_a = tuika(a)  except AttributeError:  エラー出る時  print(“リストじゃないよ！”)  else:  エラー出ない時  print(new\_a) |
| [0, 5, 87, 7, 5] |

とまぁ、最初の状態と一緒ですね。まじでストレスフリー。ただたんにストレスが無くなるよっていう効果でしたー。

(※)ちゃんと使いどころはあるんだけどねー。

応用編

おまけ

* エラーコード早引き
* こわざ(ショートカットキーetc.)
* よく使うメソッド
* リストに対するメソッド
* 文字列に対するメソッド
* よく使うライブラリ
* numpy
* matplotlib
* pandas
* バージョン管理
  1. バージョン管理の考え方

バージョン管理ってのは、“いつ何をしたか”を分かりやすくするものです。例えば、今この資料作る時に、何章まで完成してたっけって分からなくなることが多々あるのですが、それを一発で分かるようにしたり、どっかの作業まで巻き戻したりするのがバージョン管理です。まあまあ便利です。まあまあね笑

* 1. Gitとは

Git。初めましての方も多いのでは。Gitはバージョン管理システムで今一番使われているといっても過言ではないシステムです。ただ、Git単体ではコマンドを使って管理するので非常にめんどうくさい。めんどうくさいは罪。ってことでGitを利用してもうちょい使いやすく & 共有しやすくしたのがGitHubやBItBucketです。よく企業でも使われていますね。便利。

* 1. ブランチ

バージョン管理において、一番大事な概念、ブランチです。英語で枝って意味ですね。実はそのままの意味で、枝のように作業を分けるのがブランチです。そうすることで、ある作業を誰かにやってもらってそれを最後にまとめていくってことが可能になります。実際、僕の職場の全体構造は下図みたいになっています。(仮名) こんなに分かれてんのほんとアホみたい。

１つのプロジェクトを全員で一斉に作業すると厄災が起きるので、こうやって分けて作業してるんです。

* 1. リモートとローカル