

# Python 講座

本当に何とでもなる



Google Colaboratory



 PyTorch



# 目次

-はじめに-	4
基本編	5
1 Python ってなんだ(飛ばしてもいいよ)	5
1.1 Python	5
1.2 オブジェクト指向って何！！	6
2 開発環境(なんなら一番面倒くさい)	7
2.1 開発環境とは	7
2.2 なにがあって、なにがいいの？	8
2.3 インストール方法 & 使い方(Python + VScode)	9
2.4 インストール方法 & 使い方(Anaconda)	16
2.5 (おすすめ)使い方(GoogleColaboratory)	18
3 モジュール？ライブラリ？	21
3.1 言葉の説明	21
3.2 インストール方法	21
3.3 使い方(import? from?)	23
4 型の話	25
4.1 変数について	25
4.2 色んな型	26
4.3 Python における型の扱い	27
4.4 四則演算(+, -, *, /, その他)	28
5 ごめん、結構ごっそりまとめる	30
5.1 Python のインデント(Tab キー)	30
5.2 リスト！	32
5.3 辞書も便利	34
5.4 関数(ちょっとだけ)	35
5.5 条件(真 と 偽)	36
5.6 分岐処理(if / else if / else)	37
5.7 ループ処理(for / while)	39
5.8 <発展>ループ処理(enumerate)	42
6 関数作ろー！	43
6.1 関数の構造	43
6.2 実際に作ってみよー	43

6.3	軽く練習問題.....	46
7	クラスって、？メソッドと関数、？？.....	50
7.1	クラス・インスタンスという概念(わたあめ).....	50
7.2	ひとまずクラスを作ってインスタンス化.....	51
7.3	__init__なにそれおいしいの？.....	52
7.4	クラス変数、インスタンス変数.....	54
7.5	メソッドと関数って何が違う.....	55
7.6	プライベート変数.....	58
7.7	<発展>デコレータ(@staticmethod, @property, etc.).....	59
7.8	<発展>継承.....	63
7.9	<発展>続・継承(super().__init__).....	65
7.10	<発展>抽象クラス/抽象メソッド.....	67
7.11	<発展>Protected 変数.....	69
8	アノテーション.....	71
8.1	アノテーションとは.....	71
8.2	<発展>typing.....	72
8.3	<発展>Any, Union, Optional.....	73
8.4	<発展>Generics.....	74
9	例外処理って便利なんだよ.....	76
9.1	try の使い方.....	76
9.2	実際に使ってみる(AttributeError 回避).....	77
応用編	.....	79
おまけ	.....	80
●	エラーコード早引き.....	80
●	エスケープシーケンス.....	81
●	よく使うライブラリ(詳細は応用編).....	82
☆	numpy.....	82
☆	matplotlib.....	82
☆	pandas.....	82
●	バージョン管理.....	83
1.	バージョン管理の考え方.....	83
2.	Git とは.....	83
3.	ブランチ.....	83
4.	リモートとローカル.....	84

5.	変更をインデックスにステージング、コミット .....	85
6.	マージとリベース .....	86
7.	プルリクエスト .....	87
8.	コンフリクト .....	87
9.	フェッチ .....	87
10.	(おまけのおまけ)Sourcetree.....	88
-あとがき-	.....	89

## -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。はじめにっで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。まあ暇だからが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きます。さすがに小学生ががつつり 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)

### 1.1 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 ってなににさって言われるとなんだらうな。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。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。

## 1.2 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ていただきます。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、まあ“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。

## 2 開発環境(なんなら一番面倒くさい)

---

### 2.1 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。めんどろすぎで、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よ。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、。



## 2.2 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。

名称	説明	無料or…
IDLE	Python標準の開発環境。 必要最低限すぎる、きつい。	無料
Visual Studio Code	これさえあればなんでも書ける。 CでもjavaでもPythonでも。	無料
PyCharm	Python専用ってわけじゃないけど、 凡庸型ってわけでもないんだなも。 Pythonはめっちゃめっちゃ強い、使いやすすぎる。	無料？ (特殊ライセンスあり)
Anaconda (Jupyter Notebook)	ほぼPython専用。ノートブック型の開発環境。 ライブラリが結構入ってるので使いやすい	無料
Google Colaboratory	Python専用。ノートブック型の開発環境。 機械学習だめっちゃめっちゃ使いやすい。 現行一押しであるが、弱点もあり。	無料 (有料で強化版)

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、。まあ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 は違うけどね……)

あ、Python2 ユーザーは知りません、ごめんなさい。

ここから環境の説明になりますが、2.3～2.5の好きなやつを選んで読んでください。どれか使えればそれでいいので。

## 2.3 インストール方法 & 使い方(Python + VScode)

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。

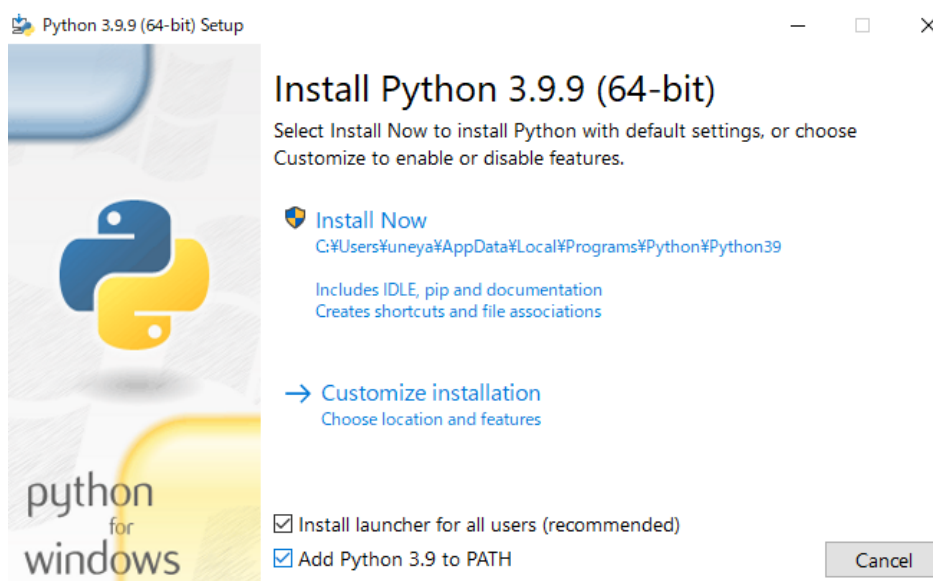
“Python Download: <https://www.python.org/downloads/>”

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)

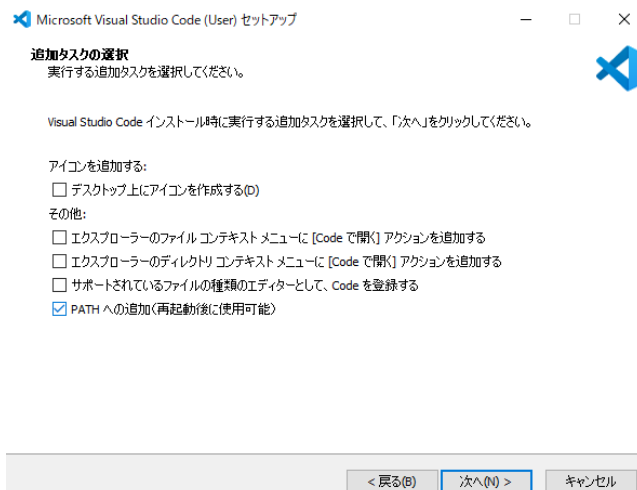
とりあえず、これで Python はインストール完了です。ここで一回パソコンを再起動しといてください。(念のため)

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。

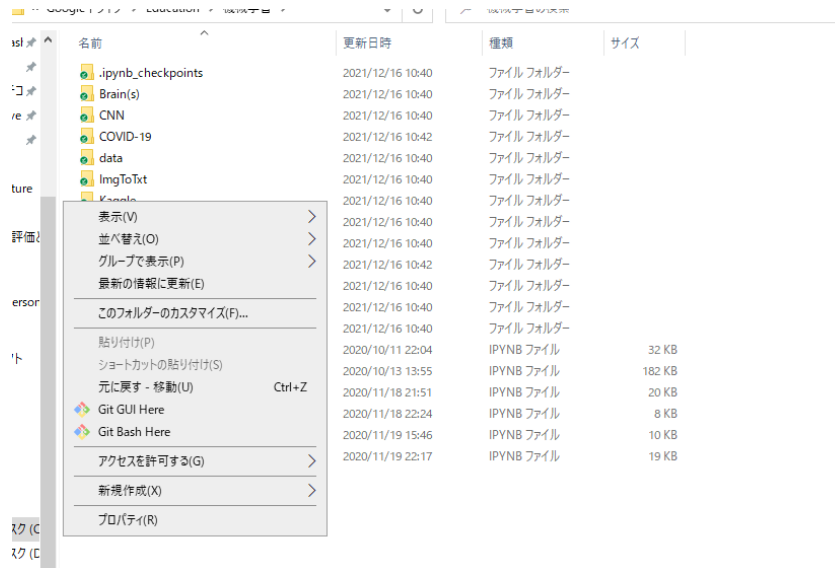
“VScode: <https://azure.microsoft.com/ja-jp/products/visual-studio-code/>”



ここから自分に合うものをダウンロードするだけです。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]アクションはまあ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。



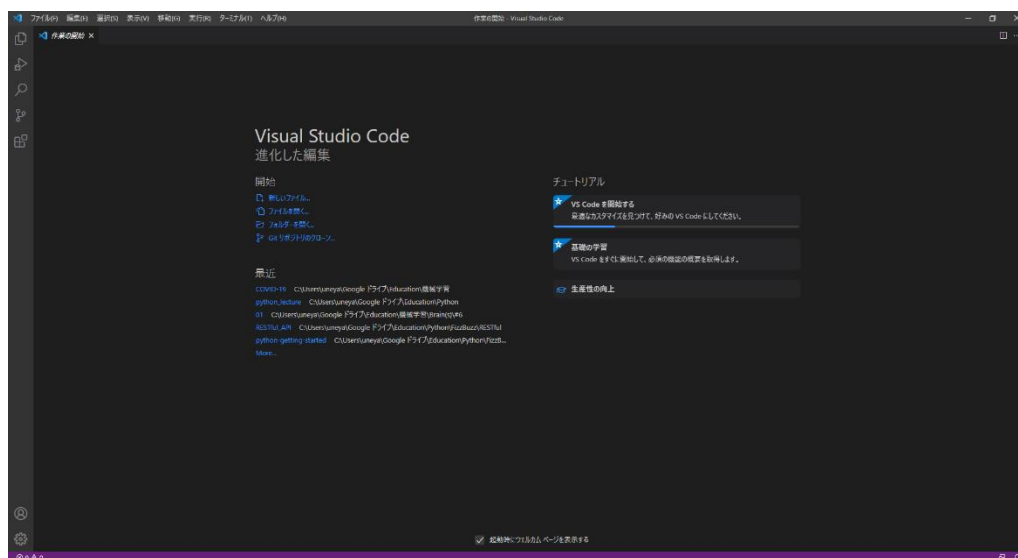
VSCode ほぼ使わないし邪魔なので僕は入れてないです笑

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。

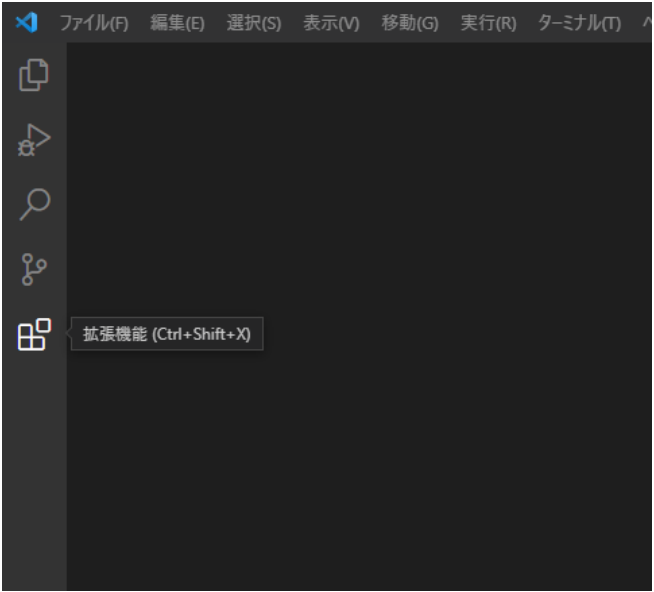
さて、これで VSCode 本体と Python が用意できました！ ここから VSCode で快適に Python が書けるように拡張機能を入れていきます。VSCode には企業やユーザーがたくさんの拡張機能をアップしています。それを使うことで使いやすくなっていくんですね。すごい。

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、。

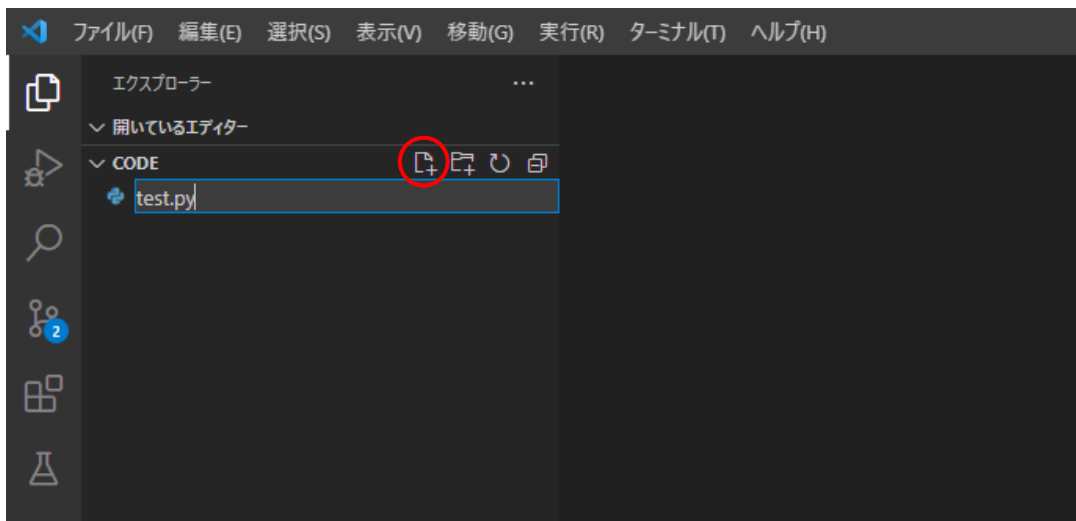


すると、検索バーがあるので、そこで検索して、とりあえず下の一覧ぐらいはインストールしましょう。

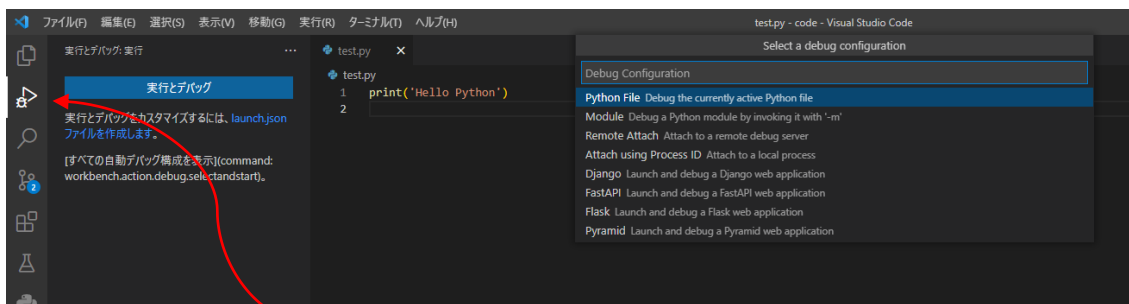
名前	機能のざっくりとした説明
Japanese Language Pack for Visual Studio Code	日本語にしてくれる (Vscode再起動で適用)
Bracket Pair Colorizer	括弧を良い感じに色分け
Output Colorizer	なんか良い感じに色分け
Python Extension Pack	Pythonを書くときに便利になる色んな拡張機能が入ってる

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とか便利そうだなーって思った。ちなみに、各拡張機能の[紹介ページには動画っぽいものがある](#)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」ってところから見れるよ)

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。開いたら、Python ファイルを作しましょう。

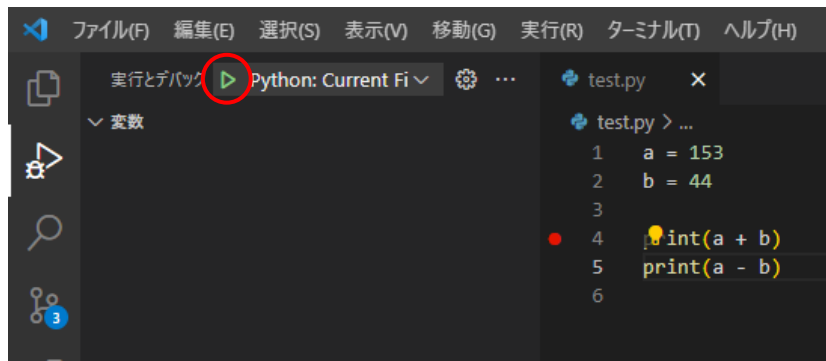


赤丸のところから新規ファイルを作って、てきとうに名前をつけてください。「.py」を最後につけてね。



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。

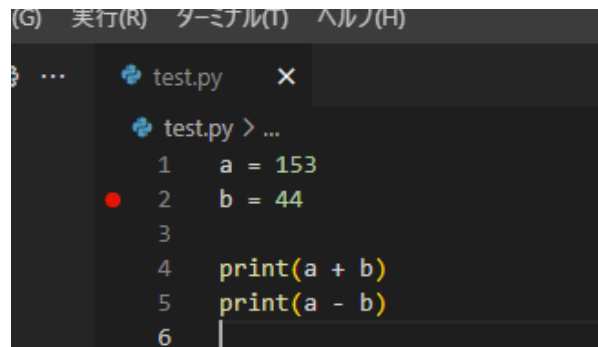
launch.json の詳細は説明しません。めんどいですし、ほぼ使わね。launch.json は×でそっと閉じてください。そっ閉じ。



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**があるのわかりますか。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。

アイコン	名前	機能
	続行	次のブレークポイントまで実行
	ステップオーバー	1行ずつ実行
	ステップイン	関数の中へ移動
	ステップアウト	関数の外へ移動
	再起動	デバッグやり直し
	停止	デバッグ終了

まあ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～。



## 2.4 インストール方法 & 使い方(Anaconda)

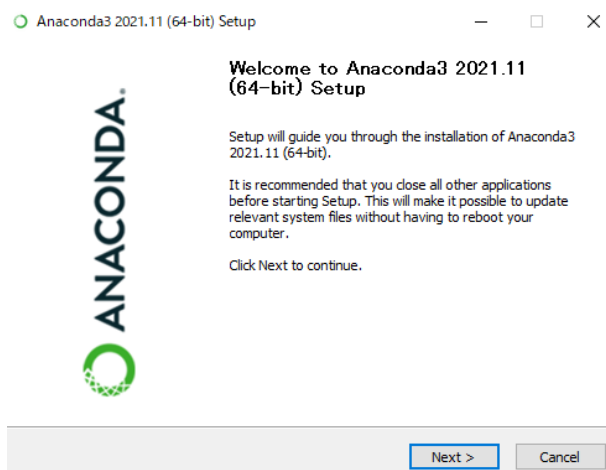
こちらは統合開発環境 **Anaconda** でございます。統合開発環境とはなんだと、“**ぜーんぶ入ったソフト**” ですね。Anaconda は Python と開発環境、さらにライブラリも全部入ってるため、**インストールするだけで Python が使えます**。デバッグ機能が付いた Spyder とノートブック形式の JupyterNotebook どちらも使えるのもいいですね。

まあ、さくっとインストールしていきましょう！

“**Anaconda**: <https://www.anaconda.com/products/individual>”

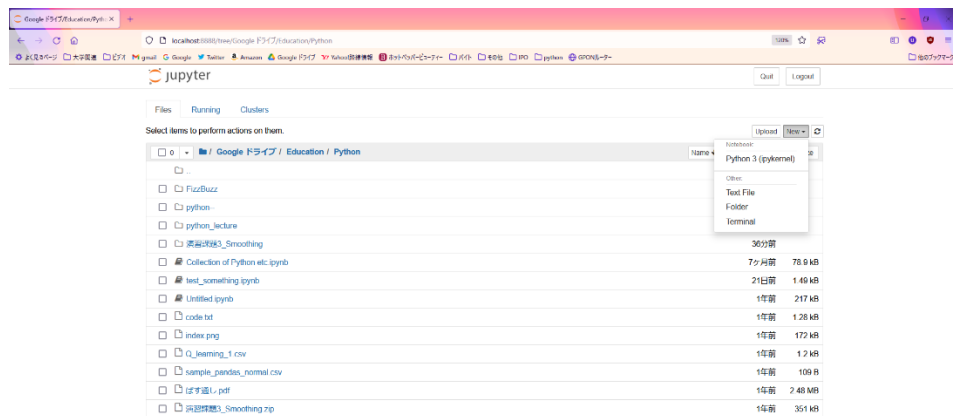
ここから自分の OS のインストーラーをダウンロードしてください。  
(最近の PC はだいたい 64bit だと思います)

インストーラーを起動したらこんな感じですかねー。えっと、特に何もいじらずホイホイ進んで大丈夫です笑

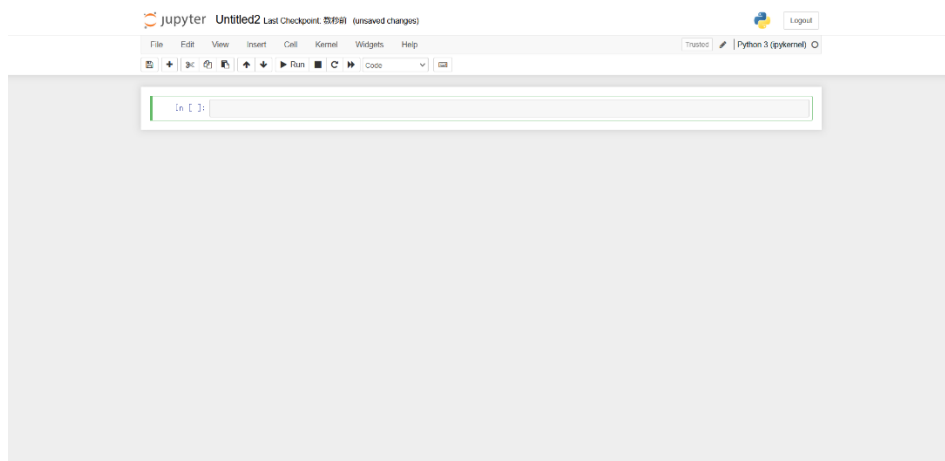


インストール出来たら起動してみましょう。ただ、他は使わないのでここでは **JupyterNotebook** のみ説明しますね。

JupyterNotebook を起動するとブラウザが開きます。こいつ実はブラウザで動作するソフトなんです。なんか変な感じ。てきとーなフォルダでここから新規作成します。



空白のノートブックができました。



さて、ノートブックの説明をちょっとだけしますね。ノートブック形式とは、コードを書く場所が「セル」というブロックに分かれていて、**メモとかが簡単に書けます**。また、**セルごとに実行ができる**ので、必要ない部分をわざわざ実行する必要なしってのも良いですね。これで Anaconda の環境構築は以上です！ たぶん簡単です。

## 2.5 (おすすめ)使い方(GoogleColaboratory)

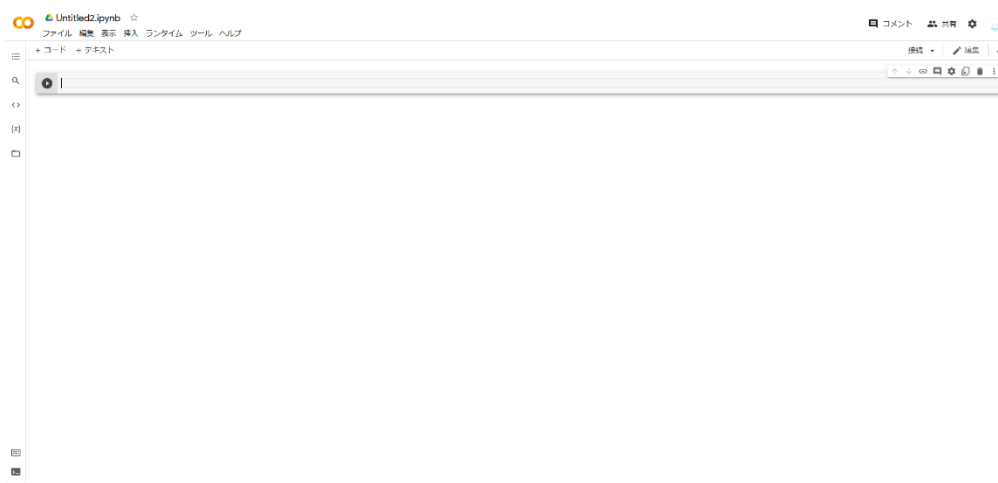
めちゃめちゃ便利。環境構築不要の統合開発環境です。

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。

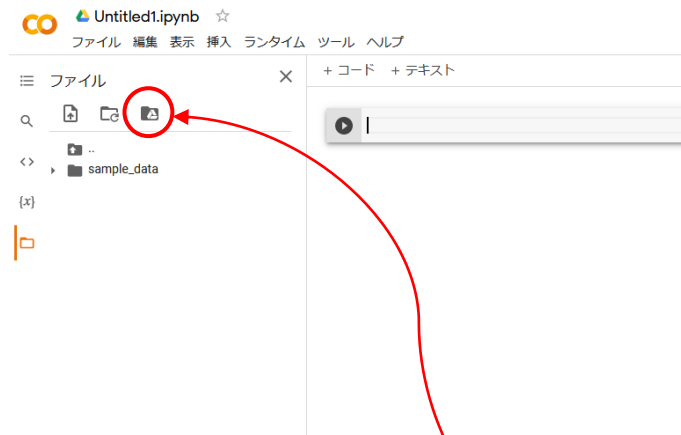


あとはここにコードを書いていくだけです。デフォルトの保存場所は GoogleDrive 直下の「Colab Notebooks」です。邪魔くさいので俺は結構てきとーに移動させてます。

これで Colab は終わり！！ 簡単ですねー。

……と、言いたいところですが、最後に 1 つだけ厄介なことが。

GoogleColab は基本的に drive に入っているファイルしか扱えません。しかし drive のファイルすら扱うのも結構厄介なんです。



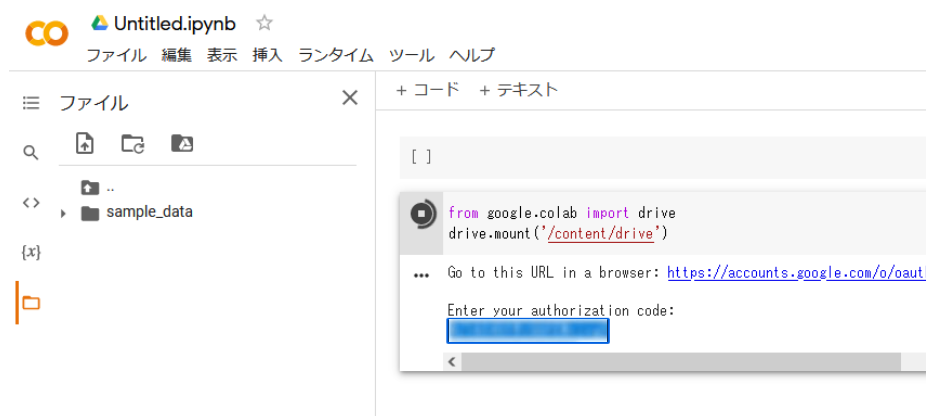
まず、初期状態だと、ファイルのアイコン押しても drive のファイルはなく sample\_data しかありません。そこで、ここのボタンを押して、自動入力されたコードを実行します。



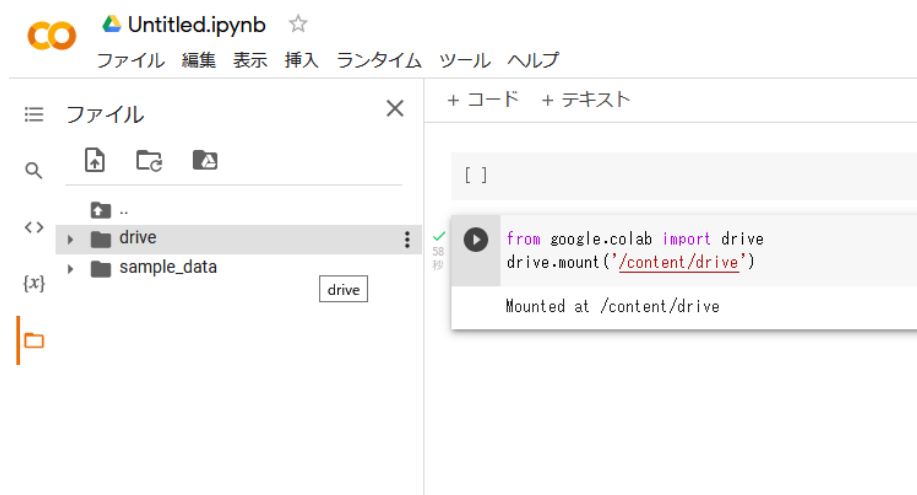
指示されるがまま、url をクリックするとアカウントを選択する画面が出てくるので使いたい drive のアカウントを選んで、ログインを押します。



このコードをコピーして、



さっきのこの箱にペーストします。そして、実行。



すると、こんな感じで drive が出てきます。あとは欲しいファイルの右「…」を押して、「パスをコピー」してやると、`pd.read_csv`などの関数にぶち込んで使えるってわけですね。

まあこれだけ面倒くさいです、GoogleColab。

## 3 モジュール？ライブラリ？

### 3.1 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。

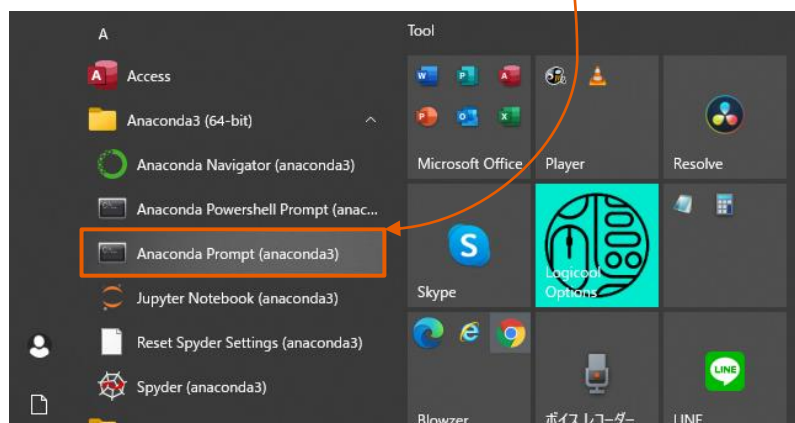
### 3.2 インストール方法

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、基本的に入っていないのが多いです。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、まあ大丈夫かと。

### 3.3 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ね。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。まあ、そんなに気にしないでください。ちょっと例を見つつ、「こんなもんかあ」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.utils.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……。まあ、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！**

## 4 型の話

### 4.1 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。まあ、要は**左に右のやつをぶち込んで**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。

## 4.2 色々な型

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)

型の名前	説明
int	整数
float	浮動小数
str	文字列
list	リスト
dict	辞書
bool	真偽値

こんなもんですかね。とりあえず今は「ふーん」って思っといってください。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。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。

## 4.3 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ません。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。

## 4.4 四則演算(+, -, \*, /, その他)

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

演算	記号
足し算	+
引き算	-
掛け算	*
割り算	/
余り	%

こんなものでしょうか。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)  
print(a*2)
```

```
そらあおい  
そらそら
```

str 型でも正しく演算できるものもあります。足し算(+)と掛け算(\*)です。掛け算は整数(int 型)を掛けると整数回繰り返されるんですね。

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。

## 5 ごめん、結構ごっそりまとめる

### 5.1 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, b 2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、機能性で見栄えを両立させたのが Python です。ちなみに先に紹介した Jupyter Notebook, VScode, Google Colab 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。

## 5.2 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。

インデックス 0 1 2 3

```
a = [0, 1, 2, 3]
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。上の例めっちゃめっちゃ分かりづらいな。

```
a = [ “わん” , “ツー” , “さん” , “し” ]
a.pop(1)
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。

## 5.3 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” }  
print(a[ “春” ]) この要素を取り出す
```

4 月

最近は四季があいまいですけどね……。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)

## 5.4 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。

## 5.5 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。

条件式	真になる条件
a == b	aとbが同じ値
a != b	aとbが違う値
a < b	aがbより小さい(a=bを除く)
a <= b	aがb以下(a=bを含む)
a > b	aがbより大きい(a=bを除く)
a >= b	aがb以上(a=bを含む)

## 5.6 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、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。

## 5.7 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
```

とまあ、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。まあ、変数一個だし、キーか要素だよなって話。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

これはまあそこそこ使うかな、`.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、これがいっつつちばん使う。`.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。

## 5.8 <発展>ループ処理(enumerate)

発展としていますが、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すが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。

## 6 関数作ろー！

### 6.1 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）

### 6.2 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。まあ、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ではないです。

### 6.3 軽ーく練習問題

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、頑張ってください！

<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`に入れられています。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()`)して行っていますね。

`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])



もう一問ぐらいいいときますか！ 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。

まあ、そこそこ色々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

## 7 クラスって、、？メソッドと関数、、？？

### 7.1 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、一番つまづく……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。

## 7.2 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。

### 7.3 \_\_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？ まあいいか。

`__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

## 7.4 クラス変数、インスタンス変数

さて、ここまできると、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？ まあ何となくで、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。

## 7.5 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。結局一緒なのにな。

前座は置いて、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。関数と一緒に“**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)

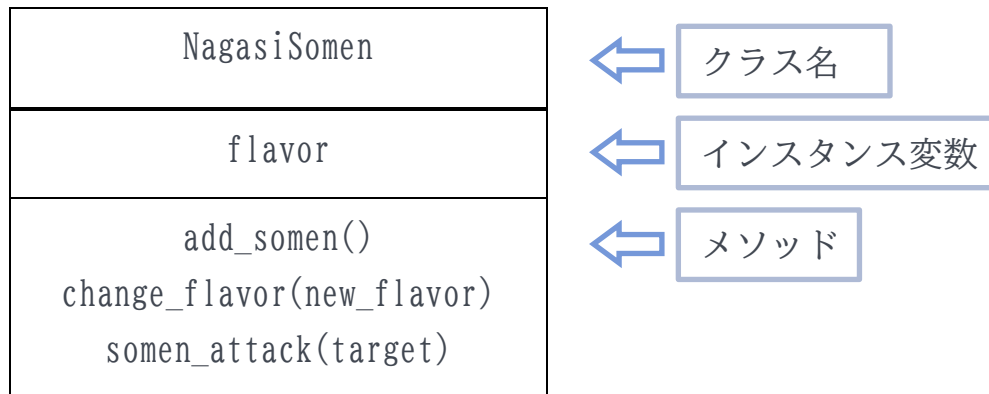
**出力の一行目**は、`candy_1` を定義した直後なので、`flavor` は“イチゴ”となります。その後、`candy_1.change_flavor` でクラス内に定義された `change_flavor` が実行されることによって、`flavor` が“ぶどう”に変化します。ってなわけで出力の二行目は“ぶどう”になるわけで



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。



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られています。

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！

<p>&lt;要件(add_somen())&gt;</p> <p>処理：現在の味のそうめんを流すよう出力してください。</p> <p>“(flavor)のそうめんを流したよ”</p> <p>(例) “よもぎのそうめんを流したよ”</p>
<p>&lt;要件(change_flavor(new_flavor))&gt;</p> <p>引数：新しい味</p> <p>処理：self.flavor を new_flavor に変えてください。</p>
<p>&lt;要件(somen_attack(target))&gt;</p> <p>処理：現在の味のそうめんで、target に攻撃してください。</p> <p>“(flavor)のそうめんで(target)に攻撃”</p> <p>(例) “サクラのそうめんで市長に攻撃”</p>

```
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。

## 7.6 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。誰だっで見られたくない1つや2つあることでしょ、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。

## 7.7 <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。言葉で言うのがめっ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！ パカ°パカ°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のでしょうか。

```
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よう。

```
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**という**関数がデコレータの実態**なんですね。wrapper 関数の返り値は func が実行されてから出力されます。まあ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。まず、`wrapper` の `print` が実行されて、`test` が実行されて、戻り値の `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。

## 7.8 <発展>継承

説明がクソめんどくさいなあと思っている**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、まあ違うゲームです。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” )
```



とまあこんな感じです。“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。

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。

## 7.9 <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:
    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。まあ、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

カッコいいですね、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。

## 7.10 <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。まあ、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っ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、**形だけ決まっています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** が設定されています。**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。まあ、 `意識高い系クラス` ってことで……。

## 7.11 <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 変数自体は軽いものです。Private 変数はプライベート変数って書いてもいいけど、プロテクトド変数って…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。

	クラス外部	継承クラス	同一クラス内
Private変数	×	×	○
Protected変数	×	○	○

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。ということだ。

```
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う。んんん、どうしたらいいんだー！

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話でした笑(@property 使おうね)

## 8 アノテーション

### 8.1 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
```

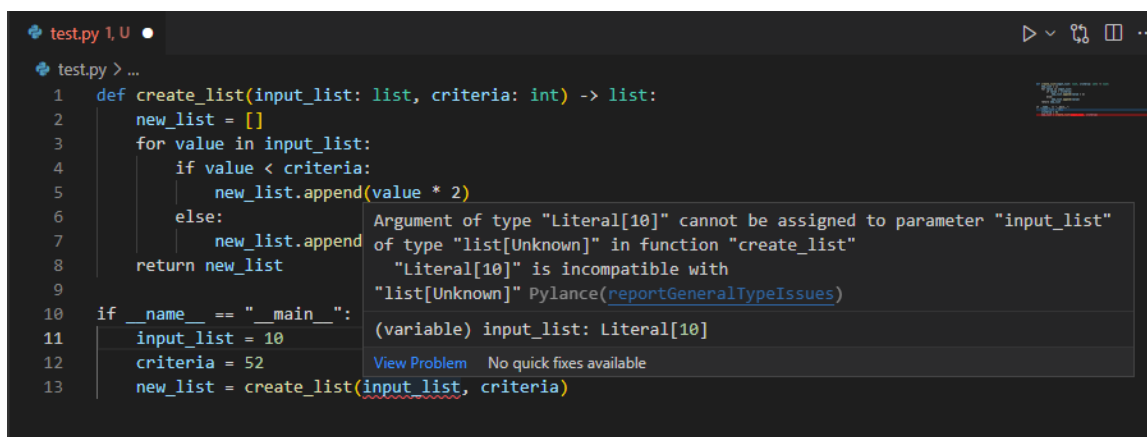
さて、6章の `create_list` を借りてきました。ここにアノテーションを実装していきましょう！

```
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。



## 8.2 <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])
    ~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。

## 8.3 <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。

モジュール	説明	大丈夫な例	ダメな例
Any	なんでもオッケー	a: Any = 562 a: Any = 45.99	無し
Union	複数の型を許容します	a: Union[int, str] = 5 a: Union[int, str] = 'natu'	a: Union[int, str] = 2.22 a: Union[int, str] = None
Optional	Noneも許容します	a: Optional[str] = 'test' a: Optional[str] = None	a: Optional[str] = 15

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。まあ  
@property みたいなもんだと思ってもらえれば。

## 8.4 <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ろなので、、、。

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** っていうのは **“T” 型** っていうことですね。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** っていう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。

```
int_adder: Adder[int] = Adder()
str_adder: Adder[str] = Adder()
print(int_adder.add(6, 5))
print(str_adder.add(15.5, 'int'))
```

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。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……。そこだけなんやなマジで。

## 9 例外処理って便利なんだよ

### 9.1 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**。

まあ、てきとーに例を書いてみましょう。

## 9.2 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:
    print(“リストじゃないよ!”)
else:
    print(new_a)
```

← エラー出る時

← エラー出ない時

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!”)`が実行されていますね。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:
    print(“リストじゃないよ!”)
else:
    print(new_a)
```

← エラー出る時

← エラー出ない時

[0, 5, 87, 7, 5]

とまあ、最初の状態と一緒にですね。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

---

えー、作ろうと思ったのですが、予想以上に基本編に時間がかかったので、分けます笑

予定としては、クラスをしっかり使う練習問題、機械学習に触れてみるなどなどまた時間がかかりそうなものになってます。なんかねー、もう皆さん結構がっつり Python 使ってるみたいだから別に要らねーなーとも思ってます、どうしようかな。まあモチベ次第ってことで。



# おまけ

## ✚ エラーコード早引き

基本的なエラーコードです。アルファベット順になってます。

もちろんこれ以外にもエラーは存在します。それはさておき、。

エラーメッセージ	説明	原因 or 対処法
Attribute Error	メソッドが存在しません。	そのクラスに対するメソッドを確認 ほんとにそれは実装されてる？
File Not Found Error	ファイルが存在しません。	ファイルのパスが間違っている
Import Error	インポートできません。	存在しないモジュールをインポートしている (ex)from numpy import takoyaki
Indentation Error	インデントがおかしいです。	変なインデントが入っている
Index Error	そのインデックスはありません。	リストの長さを確認(len関数)
Invalid Syntax	文法が間違っています。	defの「:」がないかも リストや辞書で「,」がおかしい
Key Error	そのキーはありません。	辞書のキーを確認(.keys())
Module Not Found Error	モジュールが見つかりません。	モジュールをインストールしていない 自作モジュールの場所が間違っている
Name Error	変数が定義されていません。	変数定義して！
Type Error	型が間違っています。	型の確認(type関数)
Value Error	想定される値と違うものが入っています。 (型は合っている模様)	関数の引数に入れる変数がおかしい
Zero Division Error	ゼロで割っています。	ゼロで割っています

## ✚ エスケープシーケンス

ほんとは章として作っても良かったかなあと思う、**エスケープシーケンス**。なにかと言うと、**特殊文字**に関することですね。

Pythonに限らずプログラミング言語には、「**この記号はこういう意味！**」と決められた**記号が存在**します。例えば、“**\n**”は文字列の中で改行したいときに使います。こいつを含んだ文字列はprintしたとき、**nは表示されず改行として扱われる**わけです。環境によっては、“\”ではなく“¥”かも。どっちも同じです。

こういうのを**エスケープシーケンス**って言います。

## ✚ よく使うライブラリ(詳細は応用編)

### ✧ numpy

めっちゃめっちゃお世話になってる numpy。numpy の機能は主に 1 つだけ！ ndarray という行列の概念を追加することです！

numpy はそれだけでここまでメジャーなライブラリになってますね。行列の掛け算や内積、ベクトルの外積もあります。とにかく、座標計算を用いるならリストなんて使ってられません。

応用編を作るモチベがあったら、とにかく詳しく ndarray を扱っていききたいと思います。(作れるかなー)

### ✧ matplotlib

さて、レポートにはグラフがつきもの。でも Excel って所定の書式にするのが面倒すぎる……。そんなとき、Python を使ってさくっとグラフ作っちゃえばいいじゃない。ってのが matplotlib。

実際僕はずっと matplotlib でレポートのグラフを書いています。一度書いたらほぼコピペでグラフが完成するので Excel より楽なんですよねー、。

こいつも説明するところは盛りだくさんです。きついな。

### ✧ pandas

Python 標準の辞書って、まあ便利だけど、どっか使いづらいよねー……。そんな人には pandas。DataFrame というデータ格納コンテナを追加します。

DataFrame はカラム(columns)、インデックス(index)、値(value)という要素を持っています。辞書はキーと値だったので、1 つ要素が増えてますね。辞書を一次元増やしたのが DataFrame なんですよ。

ってか、Excel 思い浮かべて。あれを Python にもってきたのが pandas です笑

マジでよく使うし、matplotlib や numpy との連携も取れるのでほんとうに便利なライブラリになってます。

## バージョン管理

### 1. バージョン管理の考え方

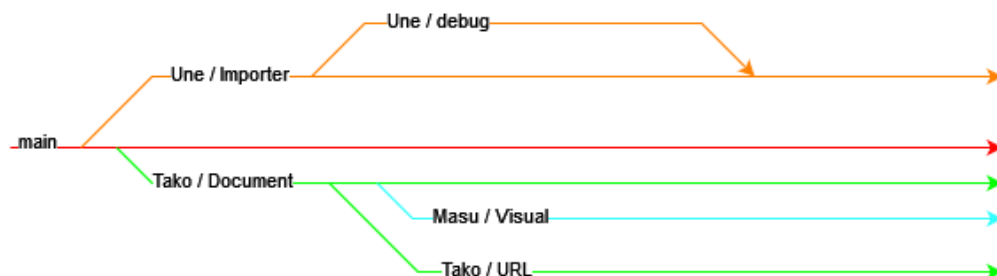
バージョン管理ってのは、“いつ何をしたか”を分かりやすくするものです。例えば、今この資料作る時に、何章まで完成してたっけって分からなくなることが多々あるのですが、それを一発で分かるようにしたり、どっかの作業まで巻き戻したりするのがバージョン管理です。まあまあ便利です。まあまあね笑

### 2. 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くさい。めんどくさいは罪。ってことで Git を利用してもうちよい使いやすく & 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。

### 3. 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)



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。

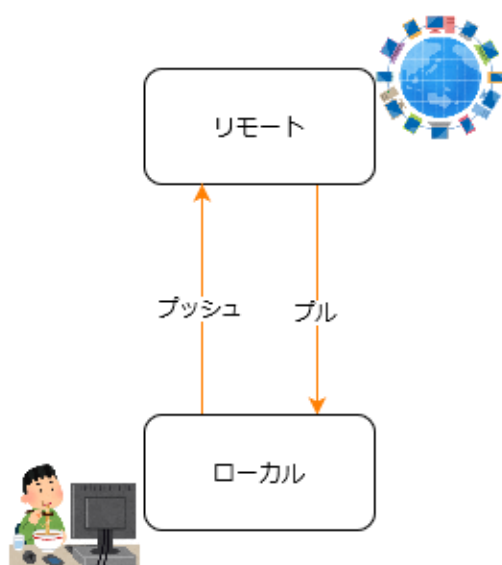
## 4. リモートとローカル

さて、ブランチに分けて作業していますが、現在作業しているのは使っているPCの中の作業です。これをローカルリポジトリ(ローカル)と言います。その状態をネットワークにアップロードした場所、それがリモートリポジトリ(リモート)です。

なにが言いたいのかというと、リモートに全て置いておけば、ネットがあるところではいつでも、ネカフェでも作業ができるわけです！

しかし、ローカルの状況が常にリモートに反映されているわけではありません。それは大変だ。こちらが反映するタイミングを決めてあげます。それがプッシュです。簡単でしょ。

逆にリモートの状況をローカルにコピーするというのがプル。例えば、ネカフェでの作業をプッシュしたら、家でプルすると作業を反映できるわけですね。



プッシュとプルの関係をしっかり理解しておきましょう。まあ逆なだけっすね。ブランチごとにリモートとローカルがあって、プルやらプッシュをしてバージョンを管理していきます。

(※)プルと似た概念として、クローンがありますが、それは一番初めにリモートをローカルにコピーするだけの操作です。一回クローンしたら、ほぼ二度とクローンしません。

## 5. 変更をインデックスにステージング、コミット

意識が高くなってきた。カタカナばかりかよ。

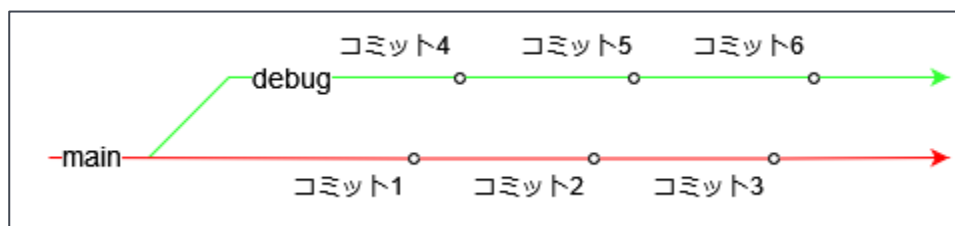
とあるブランチにて、数個のファイルに変更を加えました。しかし、今の状態ではこいつらはまだ**変更前と変更後の間**をさまよってます。ここから、確定させたいファイルをインデックスにステージングします。そして、**コミット**すると**変更が確定**されます。

ええと、、卵を想像してください。君は今ゆで卵が作りたい。冷蔵庫から卵を取り出した。でもまだ冷蔵庫に戻すのは簡単です。これが**変更前と変更後の間**です。さて、ここから選んで水に入れます。これがインデックスにステージングと言う操作。あ、**火を付けちゃいました**。もう後戻りはできません……ってのが**コミット**ですね。

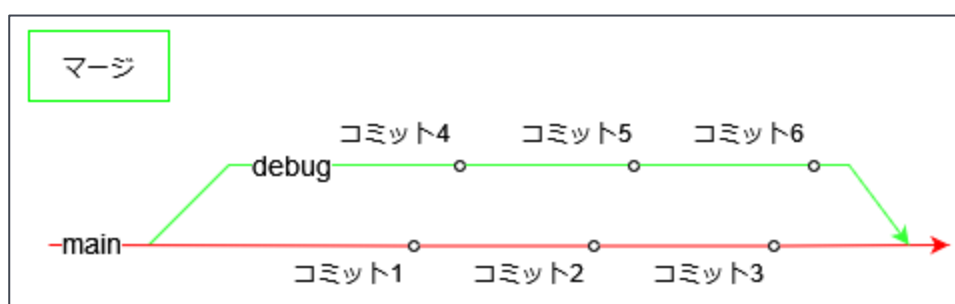
お分かりいただけたでしょうか。ちなみにコミットには、**コミットメッセージ**を付けることができます。さっきの例だと「**ゆでた！**」とかでしょうか。そんな感じで、変更をコミットしていくわけです。

## 6. マージとリベース

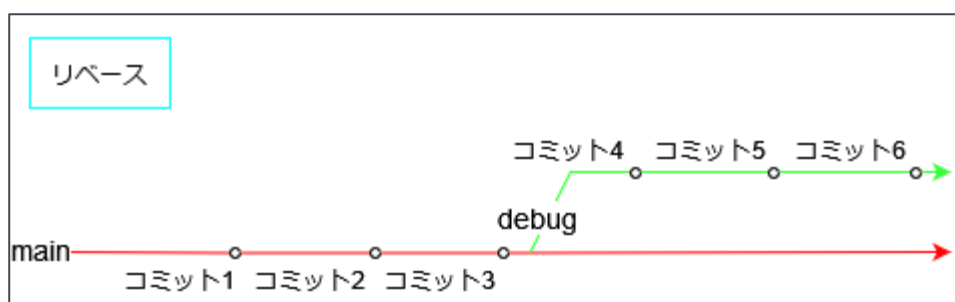
さて、また似たような概念として、**マージ**と**リベース**があります。  
二つのブランチ main と debug を使ってみましょう。



上のように、2つのブランチにはそれぞれ変更(コミット)が加えられてます。こいつらで2つの操作を行ってみましょう。



これが**マージ**。



これが**リベース**。

あれ、図を見ると全然違いますね。マージってのは**対象のブランチに結合**。リベースってのは**対象のブランチの変更を自分に反映**。

割と違う操作だったりします笑

## 7. プルリクエスト

これに関しては名前変えろマジで。ほとんどプルとは関係ありません。マージリクエストです。それで分かるかな。

main ブランチってのは基本的に、完成形なんです。それに対して勝手にマージしていくとエラーとかバグとか地獄になります。だから、「main にマージしていいですか？」って管理者に尋ねるのがプルリクエストですね。なんでプルリクエストなんだろう。ほんとにそれだけは分からない。俺が知らないだけかな。よく分かんないです。

## 8. コンフリクト

同じファイルの同じ場所を、別のブランチと自分のブランチで変更した場合。あれ？ どちらの変更を適用すればいいの、、？ ってなるのがコンフリクトです。これめっちゃめんどくさい。

機能ごとにブランチを分けてるといっても、自分の範囲外のファイルを書き換えるなんて割とざらにあることです。そんな時にコンフリクトは発生します。地道に解消しましょう。

さて、プルリクエストを出したときにコンフリクトが発生すると結構めんどくさいです。管理者は全て把握しているわけではないので、どちらのコードが正しいか判断するのは難しい。なので、プルリクエストを出す前には、main をリベースして自分でコンフリクトを解消しておくのをおすすめします。すると、管理者はプルリクエストのコードを確認してマージするだけなので効率 up です。

## 9. フェッチ

一瞬で説明終わります。リモートの変更を反映する！ 以上です！

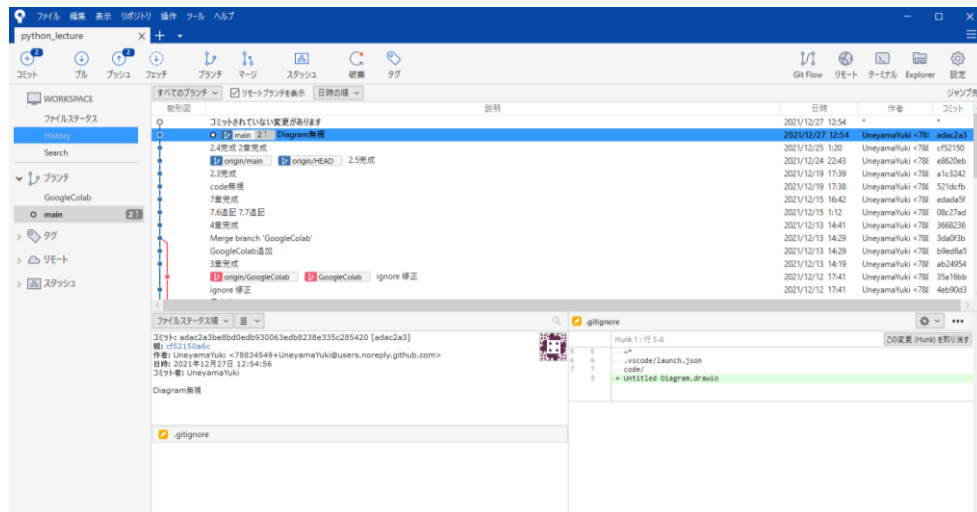
ローカルからリモートの状況って逐次反映されてるわけじゃないんです。リモートの main が進んでも、自分からみたら別になんも変わってないみたいな。そんな時、フェッチするとリモートの状態が把握できます。こまめにフェッチはしておきましょう。



## 10. (おまけのおまけ)Sourcetree

本当におまけでございます。Git って基本的にコードでコミットしたり、プッシュしたり、フェッチしたりするんですけど、めんどくさくね、？ コード覚えるのもめんどいし、。

っていうので、Git を扱いやすくしたのが Sourcetree っていうソフトです。



この文書の実際の Sourcetree です笑

ブランチの状態も分かりやすいし、上の方にプッシュだったり、プルだったりあるよね。こうやってボタンで視覚的に操作できるのはめっちゃ便利です。GitHub, BitBucket と連携できるのでぜひ使ってください。

## -あとがき-

書き終わったーーーー！！！！(終わってない)

最後まで読んでいただき光栄の至りでございます。これ何ページあるん……88 ページ？　こんなん作ったことないっす。初めてすぎた。

いかがでしたでしょうか。気楽に読めたかな。Python って基本さえ押さえればめっちゃ簡単な言語なんです。書く量もすくないしね。その一端を触れていただければ幸いかと。

応用編、作るのかな。それは反響見つつかないかな……。実際に使って作っていくのが応用編なんだけど、問題を考えるのがめっちゃめっちゃ難しくてちょっと挫折してます。メンタル的に良くない。単純にレポート一個増えてるだけやもん。おい。

ちなみに、僕はちゃんと Python 勉強し始めたのが去年の 8 月とか。実はまだ 1 年ぐらいしか経ってないんですねー。なんなら始めた 2 か月後とかに競馬 AI 作ってるしねー。その頃はメソッドとかクラスとかマジで意味不明だった。そんな状態でも機械学習ってできたりします……(小声)

こんな感じに、てきとーに AI が組めるぐらいまでを応用編で扱おうと思います。なんか楽しそうでしょ。コード動かすだけなら楽しいんだ。データを集めたり、精度を上げたりするのが地獄を見るだけ。

ま、そんな感じで締めとします。本当に最後の余談まで読んでいただきありがとうございました。困ったら聞いてみてください、たぶん答えられます、おそらく、きっと、Maybe。それじゃあ、ばいばい。