Pythonの歩き方

リリース ***0.1.2***

Noboru Yamamoto [1],
J-PARC/KEK, Tsukuba, Japapn

2021 年 **08** 月 **23** 日

[1] mail-to:noboru.yamamoto at kek.jp

目 次

ProjectInfo Python の歩き方, version 0.1.2 by Noboru Yamamoto

Author Noboru Yamamoto

Version 0.1.2

第1章　Pythonの歩き方 (Beginner’s guidebook on Python3)

1.1 旅の準備

これから Python を学ぶ旅に出る前に、 Python の概要を掴んでおきましょう。初めて聴く言葉もあるかと思いますが、それらの詳しい意味は徐々に説明していきます。

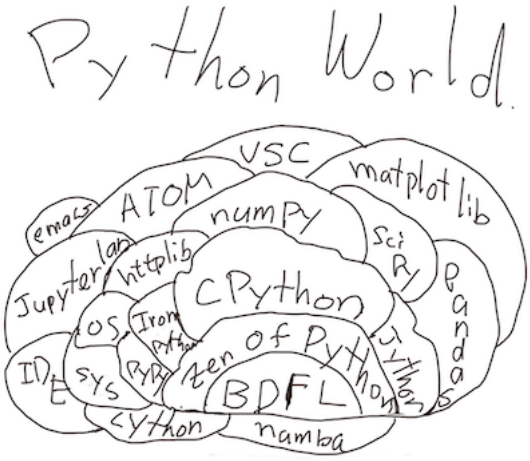


図-1.1: Python Map Image

1.1.1 python とは

まずは目的地である Python の概要を説明します。

Python はプログラミング言語の**規格**であり、またそのインタプリター形の*実行系の名前でもあります。(この実行系は `CPython` と呼ばれることもあります。)

2021 年現在において、Python はさまざまな局面で使われています。何よりも、Python はあなたが電子計算機を使う上でとても有用なツールになるでしょう。

ここでは、Python をあなたの PC で使い始めるに当たって、ちょっと知っておくと役に立つであろう事を紹介しておきます。ここで紹介した事を全てマスタする**必要はありません**が、Python プログラム開発を始めた時、思い出していただければ役にたつこともあるのではないかと期待しています。

BDFL : Benevolent Dictator For Life 慈悲深き終身の独裁官, Python の著者 Guido van Rossum のこと。彼はオランダ出身。

1.1.2 Python の歴史

図-1.2: Python_Logo_Image

---

1. Python は Guido van Rossum(BDFL) 氏がクリスマス休暇のプロジェクトとして作り出した。
2. Python ロゴの蛇は O’ Reilly の表紙から。
3. Python の名前は TV 番組のタイトルから

Mark Lutz 氏の「Programming Python」（第一版）の序文によると、 Python は Guido van Rossum 氏が 1989 年のクリスマス休暇中の「趣味」のプログラミングプロジェクトとして始めたプログラミング言語です。　van Rossum 氏は大ファンであった「Monty Python’s Flying Circus」に因んで、この言語を「深く意味も考えずに」　Python と名付けたとのことです。つまり、Python の始まりは蛇の Python とは (直接には）関係が無かったということです。

Python Logo

このことから、Python のアイコンとして当初は、
なども使われていました。 しかし、Mark Lutz 氏の「Programming Python」が蛇本と呼ばれることがあるように、大蛇 (Python) がその表紙に描かれていたことから（？）、Python のアイ

コンは蛇のアイコンへと変化していきました。

現在の Python の公式ロゴは　　です。

改めて考えると、このロゴは Python2 の時代に定められました。Python3 が主流になった今日、ロゴも三匹の蛇 (Python) に変わるのかもしれません。

蛇足ですが、迷惑メールを spam メールと呼ぶのは、「Monty Python’s Flying Circus」の一つのエピソードにちなんでいるということは良く知られています。

PEP (Python Enhancement Proposal)

Python 本体の開発は、PEP（Python Enhancement Proposal）を提出して新しい機能を提案することから始まります。試験実装と議論を通じて PEP は改定されてゆき、合意が得られたところで、公式な言語仕様として採用されます。インターネットの仕様における RFC(Request For Comments) に似た仕組みです。

これまでに提案された PEP は、PEP　インデックス (https://www.python.org/dev/peps/) で知ることができます。

1.1.3 Python の地図

モジュール (Python のライブラリ）

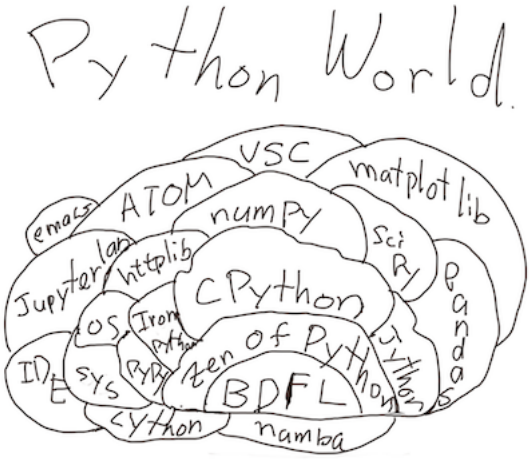


図-1.3: Python Map Image

Python の特徴のひとつは豊富なライブラリ（モジュール）です。

Python 自体は汎用のプログラム言語ですので、電子計算機で実現できることは、Python 言語を使ってできるといってもいいのですが、豊富に用意されたライブラリ／アプリケーションを利用することで、目的を手早く、簡単に実現できるでしょう。

Python の配布パッケージには *Python* 標準ライブラリ (Python ドキュメント ページ (https://docs.python.org/ja/3/library/index.html) を参照) が含まれています。 Python 標準ライブラリには、

1. Python がもつ基本的なデータ型を効率よく利用するためのライブラリ（`string`,`re`,`datetime`,…)、
2. 数学関数 (`math`, `statistics`)、
3. OS のもつ様々な機能（`os`)、
4. GUI 作成用ライブラリ (`tkinter`)

などが用意されています。

標準ライブラリ以外にもサードパーティから豊富なライブラリが提供されています。これらの拡張ライブラリは PyPI (https://pypi.org/) あるいは GitHub (https://github.com/) などで公開されています。PyPI(Python Package Index) は python ライブラリを配布する de facto スタンダードとなっており、`pip` コマンド あるいは `pip` モジュールを使うことで、簡単にモジュールを入手し、インストールすることができます。

よく使われるサードパーティモジュールには、`matplotlib`, `scipy`, `numpy`,`pandas` などがあります。 J-PARC の制御システムでは、`ca`,`ROOT`, `cVXI` などのモジュールが使われています。 (`matplotlib` をベースにした `seaborn` など世の中には様々なモジュールが存在しています。全てをカバーすることは難しいこと、また長期にわたってそれらのモジュールが利用できるかどうかは不明であることなどから、ここでは評価がある程度確立したモジュールだけをご紹介します。)

Python のモジュールはそれ自体だけが Python で書かれたモジュールと、C/C++などのコンパイル言語を使って書かれたモジュールが存在します。 後者のモジュールは、python のプログラムで実行速度が特に問題になるような場合に、実行速度を改善するために使われることがあります。 `Cython`, `f2py`, `numba` などのツールを使うことで、python モジュールを高速化する手法も存在します。

OS などが提供する C/C++などで開発されたライブラリを python から呼び出す方法も存在します (`ctypes`)。この手法は強力ではありますが、OS 間の移植性が失われますし、最悪の場合システムのシャットダウンを引き起こす可能性すらあります。これらの手法は**最後の武器** と考えておくのが良いでしょう。

いろいろな Python 処理系 (実行系）

多くの場合、python.org が配布している Python 処理系 (CPython) が使われますが、目的に応じてそれ以外の Python 処理系 (https://www.python.org/download/alternatives/) が有用である場合があります。

Python の処理系としては、次のようなものがよく知られています。

1. `CPython`： 公式の Python 処理系. C/C++による実装
2. `Jython`: java による実装。 Java object が python object に自動的にマップされます。同じ作者による、.Net 向けの `Iron Python` もあります。
3. `micropython`: マイクロコントローラ向けの小型の実装
4. `pypy`: RPython(制限された python,Python のサブセット）で作られた python 実装系。JIT(Just in Time) コンパイラを内蔵している。処理系自体は RPython から C/C++などに変換された後、処理系が作成される。
5. Brython: (https://brython.info/index.html) Javascript による Python の実装。web ページに Python プログラムを埋め込んで、実行できる。

1.2 Python のインストール確認

Python プログラムを実行するためには、Python プログラムを実行するための処理系 (アプリケーション) が必要です。幸いなことには、お使いの計算機には既に python3 がインストールされているかもしれません。　確かめてみましょう

コマンドラインツールで

```
python3 --version
```

あるいは

```
python --version

py --version
```

のいずれかを実行してみましょう。　Linux/macOS などのシステムでは前者 2 つのどちらか、Windows の環境では最後のコマンド (py --version) が使えるかもしれません。
以下に実行例を示します。

```
%%sh
python --version # on most of Unixen including macOS
#py --version # windows only
```

```
Python 2.7.16
```

```
%%bash
python3 --version
```

```
Python 3.8.2
```

```
import sys
print (sys.version)
```

```
3.9.6 (v3.9.6:db3ff76da1, Jun 28 2021, 11:49:53)
[Clang 6.0 (clang-600.0.57)]
```

これらのコマンドを実行した結果が

```
Python 3.9.6
```

であれば python3 のバージョン 3.9.6 がインストールされています。(2021.7.21 現在の python3 最新版は、3.9.6 です。)

注) 実行例の欄にある%%bash は jupyterlab でこの入力を bash で実行することを指示する、directive です。

1.2.1 python がインストールされていなかったら？

不幸にして、

- これらのコマンドが全く動作しない、
- あるいはインストールされた python がまだ python2 に止まっている

と言った場合には、Python　ダウンロードサイト (https://www.python.org/downloads) からご使用の環境にあったインストーラーをダウンロードして、python3 をインストールしましょう。

1.3 最新情報を見逃すな

1.3.1 Python ドキュメント

公式の Python3 ドキュメント (日本語訳) は Python ドキュメント サイト (https://docs.python.org/ja/3/) から閲覧可能です。 チュートリアルも用意されています。最新版以外のドキュメントも選択して表示させることができます。

1.3.2 PyPI サイト

問題を解決するのに必要なライブラリが見つからない時は、 PyPI Site (https://pypi.org/) を探してみましょう。

(`pip` コマンドの `search` サブコマンドはこの文書執筆時の状況では**使えない**状態です。 PyPI ホームページで検索しましょう。)

1.3.3 The Hitchhiker's Guide to Python!

Python ドキュメントには公式の チュートリアルも掲載されています。此他にも色々な Python 関連の資料が Internet から入手可能です。

この文書とよく似た名前の、The Hitchhiker's Guide to Python! (https://docs.python-guide.org/) は分かりやすい online の Python 入門書です。 github (https://github.com/realpython/python-guide) で最新版が提供されています。

日本語版 (https://python-guideja.readthedocs.io/ja/latest/) もありますが、残念なことに最新版への追随が遅れている様です。

この文書を書き始めた後に、このサイトを **再発見** しました。以前にサイトを訪れたことはあったものの、すっかりその存在を忘れていました。 とはいえ、どこかで影響を受けているかもしれません。

1.3.4 `help` 関数

python インタプリタには `help` 関数が用意されています。`help` の引数には関数名、クラス名、モジュール名などを与えることができます。関数を定義する際に `help` のための文字列も定義しておくことができます。

```
def foo(arg:int):
    """
    docstring と呼ばれる関数定義のこの部分が
    ヘルプメッセージとして使われます。
    """
    return arg+2

help(foo)
foo(100)
```

```
Help on function foo in module __main__:

foo(arg: int)
    docstring と呼ばれる関数定義のこの部分が
    ヘルプメッセージとして使われます。
```

```
102
```

1.4 ライブラリ／モジュールの入手方法

python の強みの一つは公開されている豊富なモジュール（ライブラリ）です。あなたの問題を解決するのに役立つモジュールをまず探してみましょう。

PyPI:The Python Package Index (https://pypi.org/) は Pyton のライブラリを探し始めるのに一番役に立つでしょう。 もちろん、Google Seach も頼りになるツールです。

1.4.1 pip/PyPI

PIP : PyPI = yum(dnf) : RedHat/CentOS repository = apt : Ubuntu

---

pip と PyPI の関係は yum/dnf と RedHat/CentOS などのリポジトリとのに関係に似ています。

PyPI サイト (https://pypi.org/) に登録されたさまざまな Python のライブラリを pip コマンドを使って手元の環境にインストールすることができます。

pip の動作確認

pip は Python 3.4 以降には、標準で付属しています。 pip が見つからなければ、python3 のバージョンを 3.4 以上に更新した方が良いでしょう。

pip がインストールされているかどうかを確認するために、コマンドラインで、

```
python3 -m pip --version
```

を実行してみましょう。

```
pip 21.0.1 from /Library/Frameworks/Python.framework/Versions/3.9/lib/
↪python3.9/site-packages/pip (python 3.9)
```

などと端末に出力されれば、pip はインストール済みです。

```
%%bash
python3 -m pip --version
```

```
pip 21.2.2 from /Users/noboru/Library/Python/3.8/lib/python/site-
↪packages/pip (python 3.8)
```

python3 -m pip <subcommand> <parameters>

以前は、コマンド pip3 が使われていましたが、次の例の様にこの形式は将来的に使えなくなります。python3 -m pip の形式を使いましょう。

```
%%bash
pip3 --version
```

```
pip 21.2.2 from /Users/noboru/Library/Python/3.8/lib/python/site-
↪packages/pip (python 3.8)
```

```
WARNING: pip is being invoked by an old script wrapper. This will fail␣
↪in a future version of pip.
Please see https://github.com/pypa/pip/issues/5599 for advice on␣
↪fixing the underlying issue.
To avoid this problem you can invoke Python with '-m pip' instead of␣
↪running pip directly.
```

pyhton3 -m <module name>は python モジュールをメインプログラム (スクリプト) として実行することを指示します。

pip のインストール

不幸にして pip がインストールされていない場合は、上に述べたように python3.4 以降の python3 をインストールすることを検討しましょう。

どうしても python3.4 以降の python3 をインストールできない場合には、

```
https://bootstrap.pypa.io/get-pip.py
```

から get-pip.py プログラムをダウンロードした上、

```
python3 get-pip.py
```

あるいは windows では、

```
py   get-pip.py
```

を実行します。 pip のインストールの詳細は pip のインストール (https://pip.pypa.io/en/stable/installation/) に詳しく紹介されています。

pip 本体の更新

pip は python3 の標準配布の中に含まれていますが、python3 本体の更新とは独立にバージョンアップされます。必要に応じて

```
python3 -m pip install -U pip
```

を実行して、インストールされている pip のバージョンを更新しておきましょう。

pip の使い方　(-h/--help)

```
python3 -m pip --help
```

サブコマンドの詳細は、サブコマンド (以下の例では install) の後に-h または--help オプションを指定します。

```
python3 -m pip install --help
```

モジュールのインストール　(install サブコマンド)

```
python3 -m pip install <module 名>
```

search サブコマンドは現状使えないので、PyPI の web ページでモジュールを検索しましょう。

モジュールの更新 (-U/--upgrade)

```
python3 -m pip install -U <module 名>
```

ユーザースペースへのインストール (--user オプション)

```
python3 -m pip install --user  <module 名>
```

M1-mac での注意点

---

x86-64 向けしかないバイナリライブラリを含むモジュール (例えば matplotlib) を使う際には、

```
arch -x86_64 python3
```

を使う必要があります。

M1-mac と呼ばれる arm ベースの CPU を持つ macitosh では、バイナリファイルとして arm64 向け、x86_64 向け、それらふたつのバイナリを一つのファイルとしてもつ fat バイナリの三つの形式があります。

残念ながらこのノートの執筆時点 (2021/7/2 現在) では、pip を経由して macos にインストール可能なパッケージでありながら、x86_64 向けのバイナリファイルだけが提供されているものがあります。　例えば、よく使われる matplotlib　がその一つです。このような場合、pip でパッケージを install するばあいに、arch コマンドを用いて、

```
arch -x86_64 python3 -m pip install -U matplotlib
```

などとして、インストールすることができます。パッケージが x86_64 向けのバイナリしか持っていませんので、利用する場合にも、

```
arch -x86_64 python3
```

などとして、python3 本体も x86_64 バイナリを使って起動することが必要です。　いずれ M1 mac 向けの配布パッケージがバイナリを含む場合には、Fat binary 形式が配布されるようになると思われますが、それまで、少しの間このトリックが必要になります。

（Python.org で配布されている macosx 向けの python3 本体の配布も実験的に Fatbinary の Installer が提供されています。あと少しの我慢でしょう。)

Note: matplotlib 本体ではなく、その中に含まれる scipy モジュールが問題となります。scipy をソースコードから arm64 向け（あるいは Fatbinary 向けに）にコンパイルするなどの方法でインストールすれば、 arch コマンド無しでも利用可能になります。（吉本さん、情報提供有難うございました。）

####　macos で Python で SSL を使う note: macos 上の python プログラムで SSL を使う際には次のおまじないが必要です。

```
import os, certifi
os.environ["SSL_CERT_FILE"]=certifi.where()
```

1.4.2 conda/anaconda

conda はデータサイエンス分野でよく使われるパッケージ配布用プログラムです。この conda をベースにデータサイエンスでよく使われさまざまなパッケージを管理／配布しているシステムが anaconda と呼ばれています。

anaconda では Excel の表や RDB のテーブルと親和性のある dataframe というデータ構造を使います。この dataframe は pandas モジュールをインポートすることで、conda/anaconda と離れて利用することも可能です。

尤も、これは順序が逆でデータ解析の Python モジュールをまとめたパッケージを, 蛇のアナコンダに因んで、anaconda と名づけたと思われます。その anaconda のパッケージプログラムを conda と名づけたということと類推します。

それではなぜアナコンダなのかといえば、Python 自体のアイコンに蛇が使われるように、python=大蛇ということで、大蛇の一種であるアナコンダにかけたということでしょう。
　もっともプログラム言語の Python の名前は BBC で放送されたコメディ番組”Monty Python’s Flying Circus“から採ったと作者 Guido van Rossum が述べています。(Foreward for”Programming Python",Mark Lutz, 1996, O’Reilly & Associates, Inc.） [fn_python][]

[fn_python]: 関係ないですが、 Wikipedia（日本語版）の Python の解説に、KEKB control home page (http://www-acc.kek.jp/WWW-ACC-exp/KEKB/control/KEKB-Control-home.html) が脚注で引用されている (リンク切れ）ことに気が付きました。(2021.4.21)

1.4.3 その他のパッケージ配布システム

Linux であれば、dnf(yum) や apt といったパッケージ配布プログラムで python 処理系や様々なモジュールをインストール／更新管理することができます。これらの配布パッケージに含まれる Python やそのライブラリのバージョンは、上記の pip や conda のリポジトリに比べやや古いことがあります。

一方で、利用中のシステムとの互換性が確認されているなど、運用上の安定性には優れています。

macos では homebrew や Fink, mac ports もよく使われるパッケージ配布システムです。

これらのパッケージ配布システムを複数同時に使うと、ライブラリバージョンの衝突などで動作不良になることがあります。注意してお使いください。

1.5 開発環境:

Python プログラムは様々な方法で開発することができます。簡単なテキストエディタから、プログラムの作成から、保存／管理／テスト／リリースまでの全てのステップを助けてくれる統合プログラミング環境 (IDE:Integrated Development Environment) まで多数の選択肢が存在します。　ここでは、現在 Python プログラム開発環境としてよく知られている Jupyter(IPython, Jupyter Lab) を紹介します。

1.5.1 IPython/Jupyter/Jupyter lab

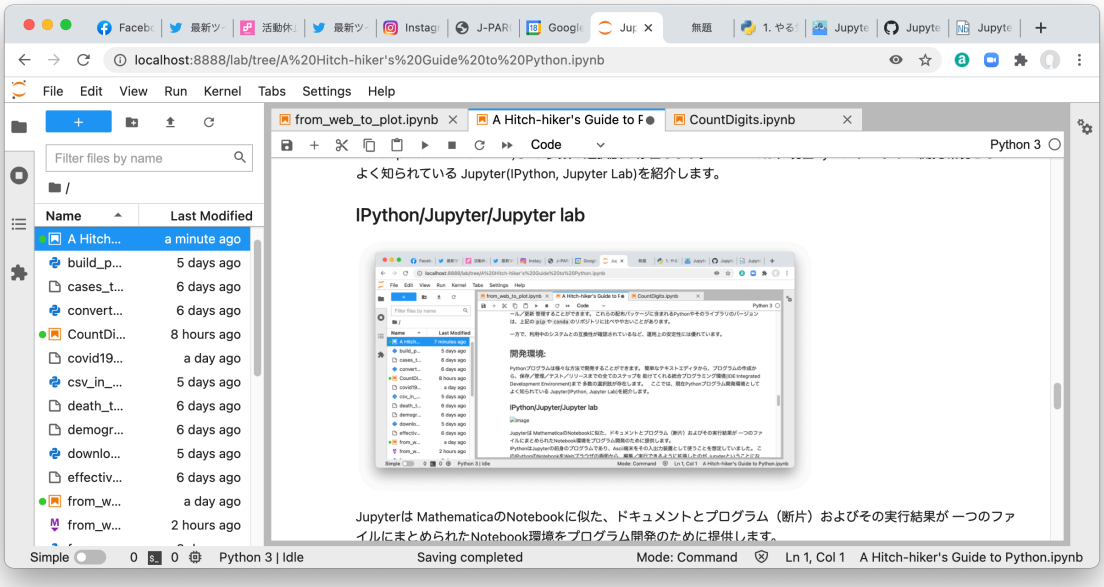


図-1.4: Jupyterlab screen Images

Notebook 形式で入力と出力 (結果）を保存しておくことができる。

Jupyter は Mathematica の Notebook に似た、ドキュメントとプログラム（断片）およびその実行結果が一つのファイルにまとめられた Notebook 環境をプログラム開発のために提供します。

IPython は Jupyter の前身のプログラムであり、Ascii 端末をその入出力装置として使うことを想定していました。この IPython の Notebook を Web ブラウザの画面から、編集／実行できるように拡張したのが Jupyter ということになります。 さらに、この web ブラウザ上の編集／実行環境を進化させ、次の世代の Jupyter になると考えられているのが、Jupyter-lab です。jupyterlab のドキュメントは https://jupyterlab.readthedocs.io/en/stable/ を探してみましょう。

Jupyter/ Jupyter-lab は pip を使って簡単にインストールすることができます。

pip install jupyter pip install jupyterlab

IPython/Jupyter/Jupyter-lab は python 言語専用の開発支援プログラムというわけではなく、C/C++など様々プログラミング言語の開発支援パッケージが存在しています。例えば、Open Source の数式処理システムである SageMath は、Jupyter を標準の開発環境としています。

Jupyter で利用可能なプログラム言語 (Kernel) は

List of Kernels (https://github.com/jupyter/jupyter/wiki/Jupyter-kernels)

のリストをご覧ください。

1.5.2 Python3 をサポートする エディター

python3 プログラムの開発にはどのようなテキストエディターでも使うことができます。しかしながら、 **emacs/ atom /VSC(Visual Stduio Code)** など多くのプログラム 開発用エディタは python プログラム開発をサポートする機能を有しています。 使い慣れたエディタに python サポート機能を追加することで、開発の効率も上がるでしょう（多分）。

1.5.3 IDLE: Python 標準配布に含まれる IDE

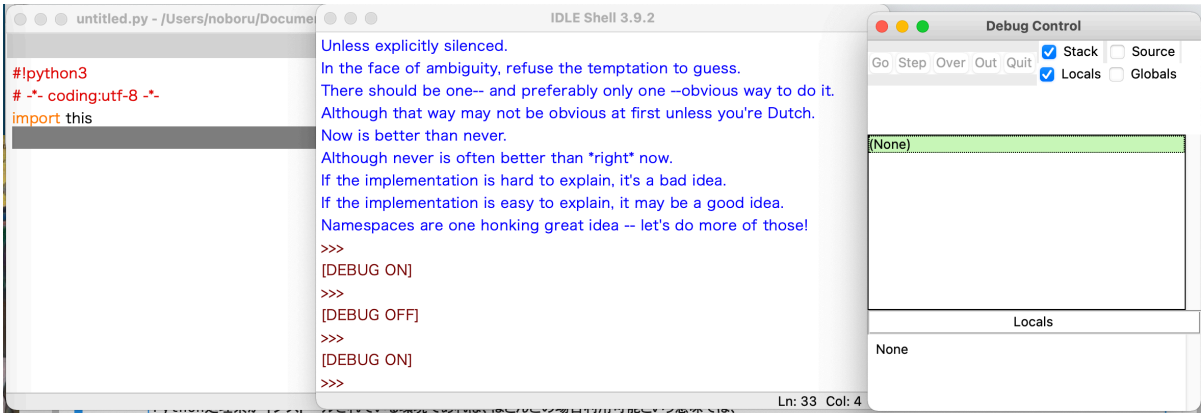


図-1.5: Image

IDLE は多くの（もし全てでなければ）Python 処理系の配布パッケージに含まれている、Python 専用のプログラム開発環境です。 IDLE 自体も Python を使って制作されています。IDE として期待される 基本的な機能はサポートされています。

Python 処理系がインストールされている環境であれば、ほとんどの場合利用可能という意味では、入門の説明にちょうど良いのですが、それほど使われているわけではなさそうです。Python と Tk でここまでできるというサンプルではあります。

1.5.4 emacs

EMACS editor

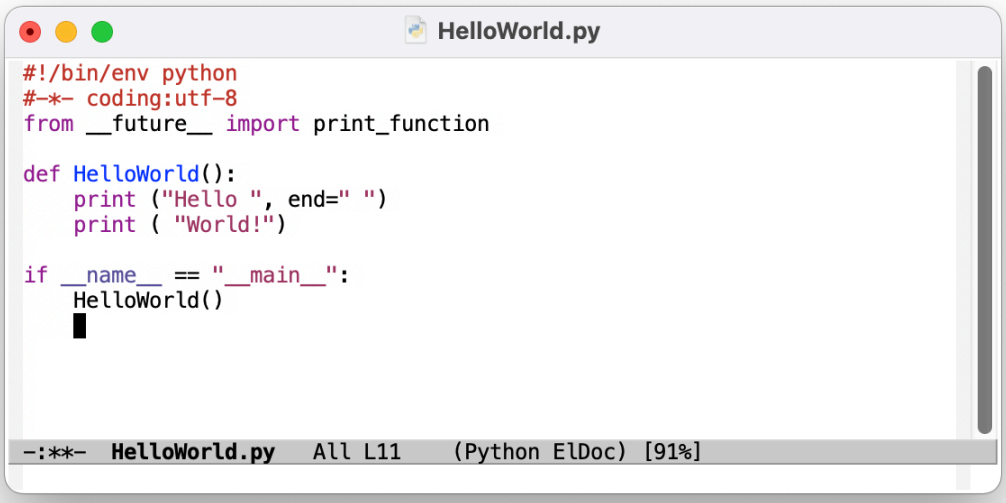


図-1.6: Image

GNU が配布しているテキストエディタ。著者の愛用です。python-mode/cython-mode などを実現する　拡張モジュール（Emacs Lisp のプログラム）が配布されており、これらのモジュールを導入することで、python プログラムの開発が効率化されます。

atom /VSC(Visual Stduio Code)

ATOM editor

atom/VSC はいずれも Javascript で開発された、近年流行りのエディターです。いずれも無料で入手できますが、atom は open source, VSC はマイクロソフトが開発提供しています。VSC には Python をサポートする機能拡張がマイクロソフトから提供されていますので、　これをインストールして利用します。

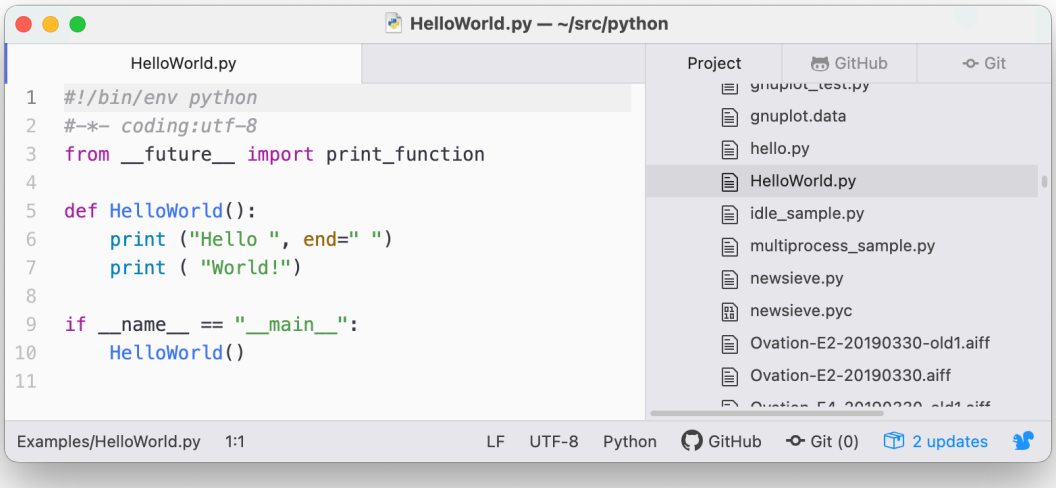


図-1.7: Image

1.6 仮想環境：venv

モジュールのコンフリクトを避けるには、 仮想環境を使いましょう。

色々なプログラムを同時に開発していると、インストールしている Python モジュールのコンフリクトが起きてしまうことがあります。また、新しい Python モジュールをテストしたいけれど、システムにはまだインストールしたくないということもあります。このような場合に役に立つのが venv などの Python 実行用仮想環境作成ツールです。venv は python3 の標準配布に含まれているので、これから始めるのが良いでしょう。

virtualenv は python2 の時代によく使われていたツールです。venv は virtualenv の機能のサブセットを python3 の標準機能として取り込んだものとされています。 フルセットの `virtualenv <https://virtualenv.pypa.io/en/latest/>`__ や `pyenv <https://github.com/pyenv/pyenv>`__ などの機能強化されたツールも 3rd パーティから登場しています。

1.6.1 venv の使い方

venv 環境の作成

python の独立した環境を作るために、venv を使ってみましょう。

まずこの環境をセットアップするディレクトリ（ここでは、/path/to/new/virtual/environment）とします。

```
python3 -m venv /path/to/new/virtual/environment
```

を実行することで、/path/to/new/virtual/environment の中に、独立した環境を設定するために必要なファイルを作成します。

```
% ls /path/to/new/virtual/environment
 bin         include     lib     pyvenv.cfg
```

pytenv.cfg には venv の設定が書き込まれています。

```
% cat pyvenv.cfg
home = /path/to/PythonBinary/3.9/bin
include-system-site-packages = false
version = 3.9.6
```

venv 環境に入る。

使用する Shell に応じた方法で venv 環境を activate します。

作成した venv 環境を activate することで、この独立した python 環境の利用が始まります。activate は venv の必要とする環境変数等を操作します。したがって、お使いになる SHELL に合わせた activate ファイルを利用することが必要です。

bash/shell では、

```
source <venv>/bin/activate
```

あるいは

```
. <venv>/bin/activate
```

csh/tcsh では、

```
source <venv>/bin/activate.csh
```

fish では、

```
source <venv>/bin/activate.fish
```

PowerShell Core

```
$ <venv>/bin/Activate.ps1
```

Windows の cmd.exe では、

```
C:\> <venv>\Scripts\activate.bat
```

PowerShell では

```
PS C:\> <venv>\Scripts\Activate.ps1
```

をそれぞれ実行します。 <venv> は作成した venv 環境の Path 名 (作成例では、/path/to/new/virtual/environment ）を指定します。

実行すると、

```
bash-3.2$ source  /path/to/new/virtual/environment/bin/activate
(environment) bash-3.2$
```

の様にシェル　プロンプトに利用している venv 環境の名前が追加されます。この状態で起動される python の本体は、

```
(environment) bash-3.2$ which python3
 /path/to/new/virtual/environment/bin/python3
```

となっています。

この venv 環境で、pip3 や“python3 -m pip”を使って、python モジュールをインストールすると、 <venv>/lib/python3.x/site-packages に必要なファイルが追加されます。この仕組みによって、通常の環境と venv 環境を切り分けている訳です。

venv 環境を抜ける。

この独立した環境を抜け出して、元の状態に戻るには、deactivate コマンドを使います。

```
(environment) bash-3.2$ deactivate
bash-3.2$
```

1.7 バージョン管理

今時のプログラム開発ではソースコードをバージョン管理することが必須でしょう。

git(git)/ mercurial(hg) などが現場での主要なバージョン管理ソフトウェアでしょう。
これらのソフトを環境に応じてお使いください。

1.7.1 mercurial(hg)

mercurial は python で作られているバージョン管理ツールで、

```
python3 -m pip install mercurial
```

で簡単にインストール可能です。

多くの統合開発環境でもこれらのツールはサポートされています。

1.8 その他

1.8.1 cython

もっと速く！: C/C++のライブラリと Python の結合

プログラムの実行時間の 99%はプログラムの 1%の実行であると言われることがあるように、プログラムのごく一部を高速化することで、飛躍的に性能が向上します。

python 処理系はインタプリタ形のシステムなので、実行速度は C++/Fortran などと比べれば遅くなります。　cython は python 言語のサブセットに独自の拡張を加えた cython 言語で書かれたプログラムをコンパイルし、python から呼び出せるようにしてくれます。cython 言語は python 言語と一定の互換性がありますので、python 言語で書かれたプログラム（関数）をコンパイルすることも可能です。cython は既存の別言語 (C++とか）で書かれたライブラリを python から利用する場合にも使われます。

numba といった同様の機能をもったツールもあらわれています。

1.8.2 numpy/f2py/scipy

numpy は python で高速な数値計算を行う場合の実質的な基盤となっている、numpy.ndarray データ構造などを提供しています。scipy は科学計算向けのライブラリです。

1.8.3 numba

numba は python(のサブセット）で記述されたプログラムを、実行時に機械語に変換して実行する JIT(Just In Time) 型コンパイルツールです。一旦機械語に翻訳された結果は再利用されますので、大量のデータを処理する際には、翻訳 (コンパイル）の時間を考慮しても、大きな実行速度の改善が見込まれます。

1.8.4 Python のチューニング： cProfile/profile/timeit

1.8.5 py2app/py2exe

python で作成したプログラムを単独のアプリケーションとして配布可能な形にまとめあげてくれるツールです。作成した python プログラムを必要な python モジュールおよび python の処理系をパッケージ化してくれます。

1.8.6 python2 と python3 の違いについて

Python2 から Python3 へのバージョンアップでは、

- print が文から関数になった。
- Tkinter の名前が tkinter に変更された。
- 文字型データ (str) が unicode になった。

など Python 言語の文法の変更、モジュール名の変更／統合など広範な変更があります。幸いなことに、2to3 などのツールを使うことで、python2 向けのスクリプトを python3 向けのスクリプトに書き換えることや、python2/python3 両用のスクリプトの書き換えることなどをサポートしてくれるツールが存在します。詳細は、拙著のメモ Python2 から Python3 へ移行するときに知っておくといいかもしれない幾つかの事 (http://www.j-parc.jp/ctrl/documents/articles/python2to3/index.html) がご参考になれば幸いです。

1.8.7 python の哲学 (禅)

import this を実行してみましょう。

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

“PEP 20 – The Zen of Python”は、 Python の先駆者の一人である Tim Perters 氏が拝察した Python の産みの親 Guido van Rossum 氏の Python 設計の指針を簡潔に表現したものです。

筆者の個人的見解ですが、これは”Zen(禅)”であって、 教条主義的に守るためのルールというわけではなく、迷った時の指針として使われるべきものと考えます。 （”禅宗は臨機応変”だそうです。）

“PEP 20”には、”Long time Pythoneer Tim Peters succinctly channels the BDFL’s guiding principles for Python’s design into 20 aphorisms, only 19 of which have been written down.”となっていて、あと一つ書き下されていない指針があることになります。 未完のまま残しておくのは、”Zen"らしいところですね。

第2章　始めの一歩： Hello World!

ブライアン・カーニハンとデニス・リッチーによる著書「プログラミング言語 C」（1978年）以来、プログラミング言語の最初の例として、文字列“Hello, World!”を画面に印刷するプログラムがよく使われています。この習慣に迎合して、ここでも Python 版の HelloWorld プログラムの一例を示します。

2.1 最初のバージョン

Python はインタプリタ型言語なので、端末からプログラムを一行づつ入力して動作を確認することができます。

```
% python3
Python 3.9.4 (v3.9.4:1f2e3088f3, Apr  4 2021, 12:32:44)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print ("Hello, World!")
Hello, World!
```

端末で python3 コマンドを実行すると、python3 の入力プロンプト>>>が表示されます。

ここで print("Hello, World!") を入力して Enter キーを押せば、端末に文字列 “Hello, World!” が表示されます。

```
%%bash
python3
print ("Hello World!")
```

```
Hello World!
```

```
%%python3
print ("Hello, World!")
```

```
Hello, World!
```

…

が、ここでは繰り返し何度も使えるプログラムとしての HelloWorld プログラムをご覧いただきます。

2.2 Python プログラム：hello.py を用意する

作成したプログラムを繰り返し何度も使うために， プログラムはファイルに保存した形で用意します。

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# save this as hello.py
# 行中の#以降は行末までコメントとなる (Python プログラムの実行に影響しない)

def Hello():
    print("Hello, World!")
    return

if __name__ == "__main__":
    Hello()
```

```
Hello, World!
```

この様に Python の中で関数 hello() を定義することで、同じ動作を少ない手順で実行することができるようになります。(Python) プログラムの作成では、問題に応じて解決のための手順を分解／整理して関数やクラス （クラスについては後々説明します）の形でまとめておくということが大切です。

Jupyterlab, VSC, emacs などのプログラム開発支援機能のあるエディタでは、プログラムの実行結果をそれらの環境の中で確認することができます。具体的な手順についてはそれぞれの開発環境のマニュアルをご覧ください。近くに同じ環境をお使いの方がいらっしゃれば、その方に聞いてみるのも良い方法です。 質問の際には、疑問点が具体的にわかるように事前に質問を整理しておくと良いでしょう。

2.2.1 hello.py の中身

このプログラムの中心部分は def Hello():で始まる三行のプログラムです。この三行のプログラムによって、関数 Hello() を定義しています。

```
def Hello():                # 関数の定義を宣言
    print("Hello, World!") # 関数の本体、インデントに注意
    return                  # 関数呼び出しの終了
```

これらの行を一行づつ説明していきます。

2.2.2 関数の宣言

まず、def Hello():で関数名が Hello という関数を定義することを宣言しています。

次の、() はこの関数には引数がないことを示しています。

:はこの後に関数の本体が続くことを示しています。

2.2.3 関数の本体

次の行がプログラムの本体です。

python3 では print は（文ではなく）関数です。引数に与えられたデータを文字列として、端末に表示します。

この print("Hello, World!") が行頭から始まっていない、つまりインデントされている、ことが Python の特徴です。C/C++では{}を使って一連のプログラム文が一つの単位(ブロック、スイート）となっていることを示しますが、Python ではこのブロック構造を行のインデントを使って表現します。(C/C++などでは、プログラムの読みやすさのためにブロック構造をインデントさせて表示することが普通ですが、Python ではこのような読みやすさのためのインデントをプログラム言語の仕様に含んでしまったというところです。）

2.2.4 関数呼び出しの終了

次の return 文はこの行が実行されると関数呼び出しが終了し、関数を呼び出したところからプログラムの実行が継続されます。この return 文もインデントされていますので、hello() 関数本体のブロックに含まれています。

2.2.5 関数の呼び出し

```
if __name__ == "__main__":
    Hello()
```

```
Hello, World!
```

続く二行のプログラムは、ファイルに python プログラムを保存する際に使われるイディオムです。（なぜこうするのか？は後で説明します。）

- if 文は、文中の条件 (ここでは__name__ == "__main__") が満足された時、

- if 文の本体 (インデントで区別された一連のプログラム文, ここでは Hello()) を実行します。

if 文の文末は:で、実行すべきブロックがインデントされていることに注意しましょう。

__name__は Python システムが利用する変数で、**name 属性はモジュールの完全修飾名に設定されなければなりません。**となっています。　python プログラムが python3 コマンドから起動されたとき (後述）は“**main**”に設定されるという約束になっています。

if __name__ == "__main__":は、%python3 Hello.py のように, コマンドラインからこのプログラムが実行された時、続くコードブロック (:があり、続く行がインデントされていることにご注意ください。）を実行することを意味していいます。ここでのコードブロックの中身は Hello() の一行だけです。Hello() が実行されると、その前に定義した Hello 関数の定義に従って、print("Hello, World!") が実行されて端末に “Hello, World!”が表示されると言うわけです。

この簡単なプログラムでは必要ないとも言えるのですが、今後 python の実用的なプログラムを書いていく上では、このように関数定義と、実際の実行部分を分離しておくことで開発の効率が良くなります。ぜひ習慣化してください。

なお、このプログラムの最初の二行は Unix 系の流儀で、このファイルの中身が“python3”のプログラムであること (#!python3)。また、このファイルの文字はユニコード（”utf-8") であること (# -*- coding: utf-8 -*-) を宣言しています。これも習慣として書いておくようにしましょう。　なお python のプログラムでは　行中の#から行末までは**コメント**として取り扱われ、プログラムの実行には影響しません。

###　ここまでのまとめ

- python プログラムの作成では関数を定義していく。
- 関数の定義は def 文を使う。
- 条件判断は　if 文を使う。
- def 文、if 文の中身のプログラムは、行頭を**インデント**する

第3章　プログラムの実行

作成したプログラムを hello.py という名前のファイルに保存します。このプログラムを実行するにはいくつかの方法があります。

3.1 コマンドラインからの実行

第一の方法は、シェルのコマンドプロプトから pytho3(or py -3) コマンドを使って実行する方法です。

ファイル hello.py を保存したディレクトリで、

```
% python3 hello.py
```

を実行しますと、

```
% python3 hello.py
Hello, World!
```

のように、端末に文字列が印刷されます。

このように動作するのは、hello.py に `if __name__ == "__main__":`以下のプログラムを書きこんだことによるものです。

jupyter/jupyterlab をお使いの方は、Code 入力セルにプログラムを入力したら、Shift+Enter を押してみましょう。

もしエラーがでても心配しないでください。まずは以下の解説をお読みください。

jupyterlab ではセルマジック%%bash を使うことで、シェルコマンドの入力結果を文書の中に取り入れることができます。

```
%%bash
python3 hello.py
```

```
Hello, World!
```

端末から同じプログラムを起動する別の方法もあります。

```
%%bash
python3 -m hello
```

```
Hello, World!
```

-m のパラメータには python モジュール名を指定します。 .py がついていないことにご注意ください。

3.2 シェルスクリプトとしての実行

Linux や macos では作成した python プログラムをシェルコマンドとして実行することもできます。

```
%%bash
chmod +x hello.py
./hello.py
```

```
Hello, World!
```

`chmod +x hello.py` は作成した Python プログラムファイルをシェルコマンドとして実行可能とするためのおまじないです。

hello.py の冒頭に

```
#!/usr/bin/env python3
```

があることも必要です。

python プログラムを実行可能ファイルとすることは、セキュリティホールになる可能性もありますので、慎重に行いましょう。

python プログラムをシェルスクリプトとして使う場合には、`sys` モジュールの `sys.exit()` 関数でプログラムが異常に終了した場合、呼び出した shell にエラーを返すことができます。

3.3 python モジュール内の関数としての実行

こうして作ったプログラムを別の Python プログラムとして組み合わせて使うことも可能です。

```
% python3
python3
Python 3.9.6 (v3.9.6:db3ff76da1, Jun 28 2021, 11:49:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import hello
import hello
>>> hello.hello()
hello.hello()
Hello, World!
```

別の Python プログラムで HelloWorld.py の中の hello() 関数を使うために、まず HelloWorld プログラムを Python モジュールとして import します。

```
import HelloWorld
```

この時、HelloWorld.py の if __name__ == "__main__":の中身は実行されないことに注意してください。

import した HelloWorld モジュール中の hello() 関数の実行は次のように記述します。
　つまりモジュール名 HelloWorld と関数名 hello を . で繋いだ名前で、HelloWorld モジュール中の hello() 関数を指定します。

```
HelloWorld.hello()
```

```
Hello, World!
```

```
'Welcome to the World.'
```

Python プログラム中の関数として実行した時には、関数が返した値も表示しているのに注意しましょう。

```
%%python3
import HelloWorld

print(HelloWorld.hello())
```

```
Hello, World!
Welcome to the World.
```

3.4 プログラムの実行 (jupyter あるいは jupyterlab)

jupyter あるいは jupyterlab でこの解説をご覧の方は、次のコードブロック (セル）を選択して、shift+Enter　キーを押してみてください。出力の Hello World! が更新されたのがお分かりでしょうか？

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# save this as HelloWorld.py
#
"""
Python 入門講座　第 2 回の例題プログラム
"""

_DEFALUT_RETURN_VALUE="Welcome to the World."
__DEFALUT_MSG="Hello, World!"

#デフォルトの値が"Hello, World!"である引数 msg をもつ関数　hello を定義する。

def hello(msg:str=__DEFALUT_MSG)->str:
    """
    引数　msg(省略時の値は"Hello, World") を端末に出力し、
    "Welcome to the world。"を値として返す。
    """
    print(msg)
    #関数の戻り値として""Welcome to the World."を返す。
    return _DEFALUT_RETURN_VALUE

def test():
    help(hello)
    reply=hello()
    print(reply)

if __name__ == "__main__":
    test()
```

```
Help on function hello in module __main__:

hello(msg: str = 'Hello, World!') -> str
    引数　msg(省略時の値は"Hello, World") を端末に出力し、
    "Welcome to the world。"を値として返す。
```

(次のページに続く)

(前のページからの続き)

```
Hello, World!
Welcome to the World.
```

3.5 プログラムの実行：まとめ

- コマンドラインとして： ` python3 hello.py`
- シェルスクリプト　として：　` ./hello.py`
- (別の）python プログラム中で　：　`import hello; hello.hello`
- python　モジュールとしてコマンドラインから：　`python3 -m hello`
- 開発環境のエディタの中で実行する。　：開発環境に依存します。

第4章　変数名／関数名について

プログラムで使われる色々な名前 (変数名、関数名、クラス名、モジュール名、..) は識別子と呼ばれます。識別子は、これから説明するに従っていれば自由につけることができます。

とはいうものの、後でプログラムを再読したときに、それらの識別子がさすものの意味がわかるように名前をつけましょう。

4.1 識別子に使える文字

識別子はざっくり言って、

- 識別子の最初の文字は、大小のアルファベット (a-zA-Z), あるいはアンダースコア (’_’) が許される。- それ以降は、それに加えて数字 (0-9) も使用可能 - 長さには制限はない。 - 大文字小文字は区別される - python の予約語ではないこと

というルールに従います。

例えば、

x y address *my_name* a0 a1

などはいずれも有効な名前 (識別子）です。

4.2 Unicode 文字の利用

Python3 ではさらに　unicode 文字　も使えます。　識別子として使える unicode 文字の範囲については、Language Reference マニュアルをご覧ください。

ということで、こんなプログラムを書くこともできます。

```
#!python3
# -*- coding: utf-8 -*-

def 挨拶():
    print("Hello, World!")
    返事= "ようこそ, いらっしゃいませ。"
    return 返事

if __name__ == "__main__":
    挨拶()
```

```
Hello, World!
```

```
挨拶()
```

```
Hello, World!
```

```
'ようこそ, いらっしゃいませ。'
```

とはいえ、これをお薦めしているわけではありません。

4.2.1 予約語

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

これらの識別子は Python で予約されているため、　ユーザーが定義する、関数／変数／クラス／モジュール名などに使うことはできません。

これらの予約語は Python の固有の単語ということですから、この **35** 個の予約語の使い方をマスターすれば、どんな Python プログラムを読みこなすことが可能です（原理的には）。　簡単でしょう？

蛇足ですが、予約語の他にも記号　「~!@#%^&*()-+/={}[]|:;"’<>.,　」の使い方も学習する必要があります。

4.2.2 予約後語の分類

- 定数
  - False None True
- 論理演算
  - and not or in is
- 実行制御
  - if elif else while for
  - break continue pass return yield
- ライブラリ
  - import from as
- 非同期処理
  - async await
- 例外処理
  - try except finally raise
- 宣言／定義
  - class def del global nonlocal
- その他
  - lambda: ラムダ式
  - assert: デバッグ用
  - with :コンテキストマネージャ

4.2.3 ちょっと寄り道

python3 では print は関数名であって、予約語でもありません。だからこんなこともできてしまいます。

```
印刷=print
印刷 ("イロハ")
印刷=挨拶
印刷 ()
print('ABC')
印刷=print
```

```
イロハ
Hello, World!
ABC
```

4.3 文字列定数について

Python での文字列定数 (リテラル文字列）の記法には 2 種類あります。

- 一組の引用符で挟まれた文字列定数（ 「“…”」 あるは「‘…’」)
- 一組の 3 連引用符で挟まれた文字列定数（「"" “…” ""」 あるは「’’’ …’’’」)

後者は中身の文字列に改行を拭くんでいても良い、すなわち中身の文字列が複数行にまたがっていてもよいというところです。

なお、Python の文字列は C 言語のそれと異なりシングルクォート「’」を使った文字列とダブルクォート「”」を使った文字列に意味的な違いはありません。

また、Python の文字列は Unicode の文字の並びです。C 言語の文字列に相当するものは、`bytes` あるいは `bytearray` です。（いずれまた、しょうさい

文字列については、次回以降の講座でより詳しく説明します。

4.4 docstring を使おう。

ファイルの冒頭、関数定義の冒頭に置かれた文字列データは docstring と呼ばれます。コメントは後々プログラムを＊読む＊際に役立ちますが、docstring はそのプログラム／関数を利用する際に役立ちます。help() 関数を実行することで、docstring の中身が端末等に出力されます。

```
#!python3
# -*- coding: utf-8 -*-
# save this as HelloWorld.py
#
"""
Python 入門講座　第 2 回の例題プログラム
"""

_DEFALUT_RETURN_VALUE="Welcome to the World."
__DEFALUT_MSG="Hello, World!"

def hello(msg:str=__DEFALUT_MSG)->str:
    """
    引数　msg(省略時の値は"Hello, World") を端末に出力し、
    "Welcome to the world。"を値として返す。
    """
    print(msg)
    return _DEFALUT_RETURN_VALUE

def test():
    help(hello)
    reply=hello()
    print(reply)

if __name__ == "__main__":
    test()
```

```
Help on function hello in module __main__:

hello(msg: str = 'Hello, World!') -> str
    引数　msg(省略時の値は"Hello, World") を端末に出力し、
    "Welcome to the world。"を値として返す。

Hello, World!
Welcome to the World.
```

4.5 まとめ

- 関数定義には `def <関数名> (<引数のリスト>):` の構文を使う。
- 行末の“:”は　インデントされたコードブロックが続く事を示している。
- 関数の本体は　インデントされた python の文の集合となる。
- #から行末まではコメントとなる
- if 文は `if <条件> :`の構文を使う。
- <条件>の論理式としての値が`False`でないとき、続くコードブロックが実行される。

プログラム言語学習の定番である“Hello World!”プログラムの python 版を例に、python でのプログラムの最初の一歩をご紹介しました。　Python では「プログラムのブロック構造をインデントで表現する」と言う手法をとっています。これはその他のプログラム言語との大きな違いとなっており、他のプログラム言語を習得済みの方は特に、戸惑うところかもしれません。　Python でのプログラム開発をサポートしてくれるエディタなどを使うことで、この点は軽減することができるでしょう。

さて、hello.py を作成して実行してみたが、エラーメッセージが出てしまった方は、上記の解説と端末にあらわれたエラ～メッセージを読み解いて、プログラムを修正してみてください。また、エラーなく実行できた方は、　表示する文字列を変えてみて、実行してみましょう。

さあ、無事に最初の一歩は踏み出せたでしょうか？　次はどこに向かいましょうか？

蛇足：

`import this` の Zen と合わせて Yoda に帰せられるとされる次の格言も覚えておきましょう。

Yoda:.. If once you start down the dark path, forever will it dominate your destiny, consume you it will.

…

Yoda:　You will know. When your code you try to read six months from now.

4.6 おまけ：名前も印刷してみよう

出力するメッセージに、メッセージの受け手の名前も加えてみましょう。

引数としてメッセージの受けての名前を受け付ける、新しい関数 HelloTo を定義してみましょう。

```
"""
引数をもつ関数の定義の例
単独のスクリプトして実行された時には、標準のモジュール getpass を使ってユーザの名前を取り出して使う。
"""
def HelloTo(name:str)->bool:
    """
    引数 name に与えられた名前の文字列を使って、グリーティング メッセージを端末に表示する。
    """
    print("Hello {}!".format(name))
    print("Welcome to the Python world!!")
    return True

import getpass # getpass モジュールを import = モジュールの提供する関数などを使えるようにする。

def main():
    username=getpass.getuser() # getpass モジュールの getuser() 関数を使って、username を取得する。
    HelloTo(username)

if __name__ == "__main__":
    main()
```

```
Hello noboru!
Welcome to the Python world!!
```

関数の引数に値を与えて実行して見ます。

```
HelloTo("Albert")
HelloTo(name="Einstein")
main()
```

```
Hello Albert! Welcome to the Python world!!
Hello Einstein! Welcome to the Python world!!
Hello noboru! Welcome to the Python world!!
```

4.6.1 文字列データの Format(整形）

print() 関数は複数の引数を受け取った時、渡されたデータ型に応じて整形した文字列を端末に出力します。

出力される文字列をより細かく制御するためには文字列データの.format メソッドを使います。

(Python2 との互換性のために、別の方法 (""%) もサポートされていますが、.format メソッドが python3 での標準的な方法です。)

```
print("Hello {}!".format(name))
```

4.6.2 エスケープ文字列

文字列定数中に改行コードなどを挿入するには、C 言語と同じように、エスケープ文字\をつかったエスケープ文字定数を使います。

- 改行->\n, タブ　-> \t, ベル->\a,
- 八進数-> \012,　十六進数-> \x0a ,
- 一重引用符 (’)-> \', 二重引用符 (")-> \", エスケープ文字-> \\

4.7 101 回のプロポーズ

```
def propose(name):
    print("I love you,{}. Will you marry me?".format(name))

for i in range(101):
    propose("薫")
```

```
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
```

(次のページに続く)

(前のページからの続き)

```
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
I love you, 薫. Will you marry me?
```

(次のページに続く)

(前のページからの続き)

```
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
I love you,薫. Will you marry me?
```

第5章　Pythonプログラムの実例

5.1 数字の出現数を数える

5.1.1 プログラムの概要

あるシステムが生成する認証コードは 6 桁の整数です。このコードに現れる各桁の数字 (Digit) の分布がどの程度一様なのかを調べます。実際に生成されたコードの一覧から、各桁の数字に分解して頻度を数えます。まずは生成された code の一覧を変数 codes に割り当てます。

```
#!python3
#-*- coding:utf-8 -*-
# '#'以降は行末までがコメントになります。(プログラムとしては実行されない)
nlen=6 # コードの桁数は 6 桁
codes=(
    227524,  463251,  702567,  601620,
    129458,  413239,  629380,  526093,
    261547,  666552,  853626,  513298,
    142167,  612906,  995697,  660500,
    954404,  651454,  566439,  730975,
    578967,  603424,  435636,  891667,
    757294,  325567,  131075,  757309,
    198547,  542718,  511525,  357679,
    245280,
    # 2021/05/14
    40245,  314151, 689785, 246930, 429093,
    546479, 240262, 631261, 559379, 869925,
    814588, 766660, 588608, 815828, 946364,
    542718, 710428, 725610, 863406, 250107,
    629444, 865116, 86879,
    # 012345 のように 0 から始まる code は, 0 を省いて書くことにします。
)
```

次に一つの code の各桁に現れる数字の出現回数を数えます。一番下の桁から順番にみていきます。

```
def count_digits(code,nlen=6): # 関数 `count_digits`を定義します。
    # 最大二つの引数を持ちますが、二つ目の引数`nlen`は省略可能で、
    # 省略された場合の値 (既定値) は 6 です。
    acc={} #これに数字毎の出現回数を入れていきます。
    code=code+10**nlen # 最上位が 0 の場合を取り扱うために、10**nlen を足しておく。
    # code =227524 であれば  code=1227524 とするということ。
```

(次のページに続く)

(前のページからの続き)

```
    while code>=10:
        code, d=divmod(code, 10) #商 `code` と余り `d` に分解します。
        # code=1227524  => code=122752,d=4
        acc[d]=acc.get(d,0)+1
        # acc の key に d があればその値に 1 を足したもの、さもなければ 0+1 を
acc[d] に設定する。
    return acc
```

一覧表 codes の全ての code に現れる数字に頻度を数えて、足していきます。結果の度数分布表 acc を関数の値として返します。

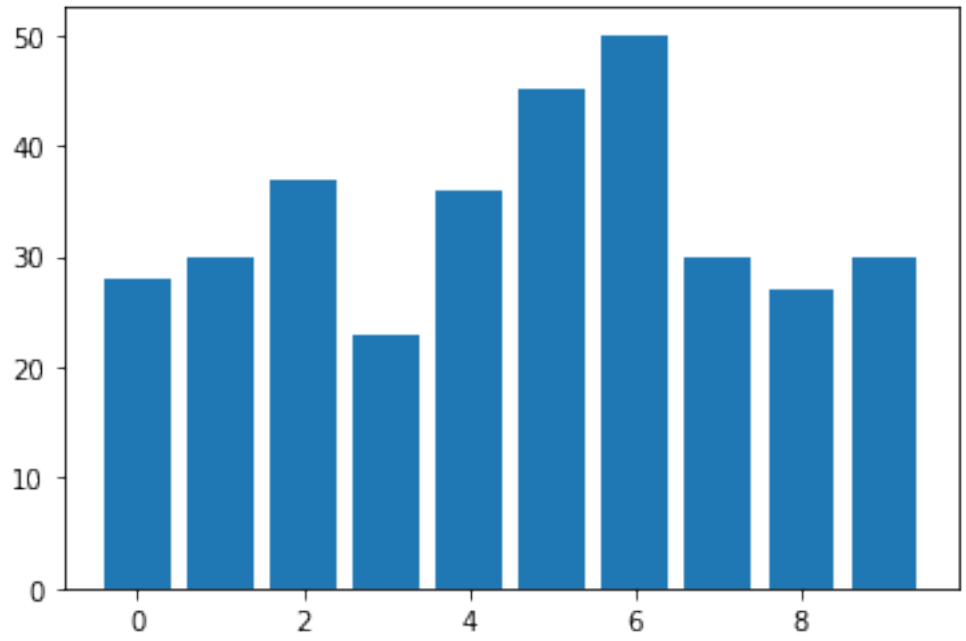
```
def count_digits_in_list(codes,nlen):
    acc={}
    for code in codes: #codes の中に入っている全ての code について
        count=count_digits(code,nlen) #コードの中の数字の出現表を作る。
        for d in count: #code での数字の出現数を acc に足して行く。
            acc[d]=acc.get(d,0)+count[d]
    return acc
```

codes の中の数字の度数分表を印刷し、ヒストグラムを表示させてみます。

```
import matplotlib.pyplot as pyplot #グラフ作成の準備のおまじない
import numpy

def main():
    counts=count_digits_in_list(codes,nlen) #code の一覧 codes から数字の出
現数の表を作成する。
    print("digit:  count")# 印刷する表のラベルを印刷する。
    for d in sorted(counts): # 出現表にあらわれる数字を整列させた順番に、
        print (d,":",counts[d]) # 数字 (d) とその出現数 (count[d]) を印刷す
る。
    # print("average:",len(codes)*nlen/10) #数字の出現数の平均値を印刷す
る。
    print("average:", sum(counts.values())/len(counts)) # 6*len(codes)/
↪10
    print("std. dev:{:4.1f}".format(numpy.std(list(counts.values()))))
    pyplot.bar(counts.keys(),counts.values()) # bar グラフを印刷する。
    pyplot.draw() #グラフ作成のおまじない
    pyplot.show() #グラフ作成の最後のおまじない. Jupyter Notebook では不要
```

```
#ファイルをコマンドとして実行するためのおまじない
if __name__ == "__main__":
    main()
```

```
digit:  count
0 : 28
1 : 30
2 : 37
3 : 23
4 : 36
5 : 45
6 : 50
7 : 30
8 : 27
9 : 30
average: 33.6
std. dev: 8.0
```



5.1.2 pandas/DataFrame

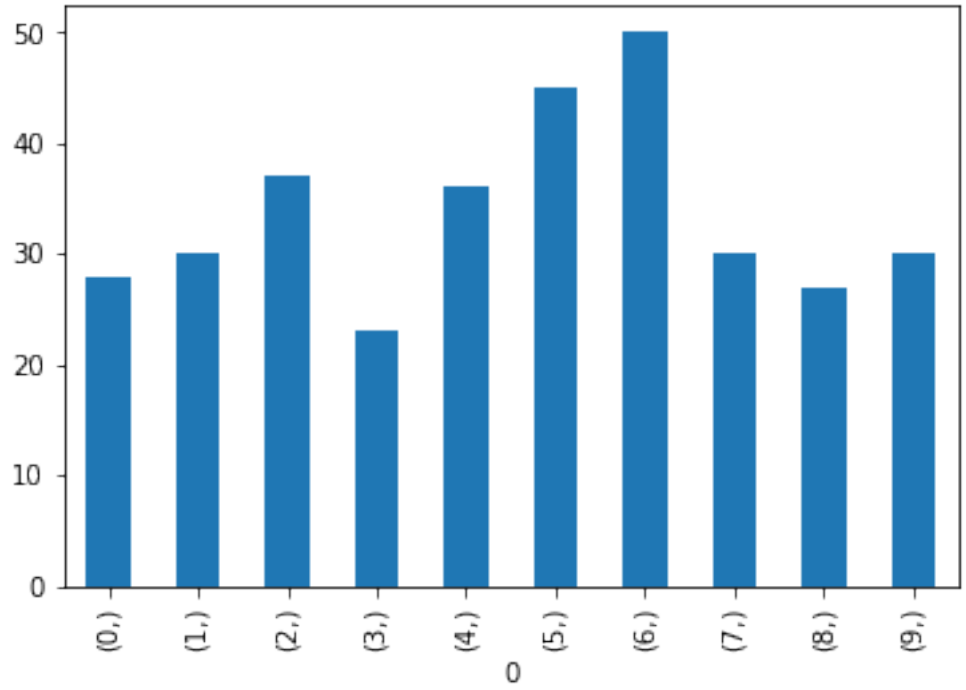
Data Analysis 分野で最近よく使われる DataFrame オブジェクトを使うと、短いプログラムでデータのリストから度数分表を取出し、またヒストグラムを表示させることができます。

```
import pandas # DataFrame が使えるように pandas をインポートして置きます。

def split_digits(code,nlen=6):
    acc=[] # あらわれた数字を順番に保存する。
    code=code+10**nlen # 最上位が 0 の場合を取り扱うため
    while code>=10:
        code,d=divmod(code,10)
        acc.insert(0,d)
    return acc

def split_codes():
    acc=[]
    for code in codes:
        acc +=split_digits(code)
    return acc
```

```
# codes のデータを一桁の数字に分解し、リストをつくります

df=pandas.DataFrame(split_codes()).value_counts(sort=False)
print(df)
print("average:",numpy.average(df))
df.plot.bar()
```

```
0    28
1    30
2    37
3    23
4    36
5    45
6    50
7    30
8    27
9    30
dtype: int64
average: 33.6
```

```
<AxesSubplot:xlabel='0'>
```

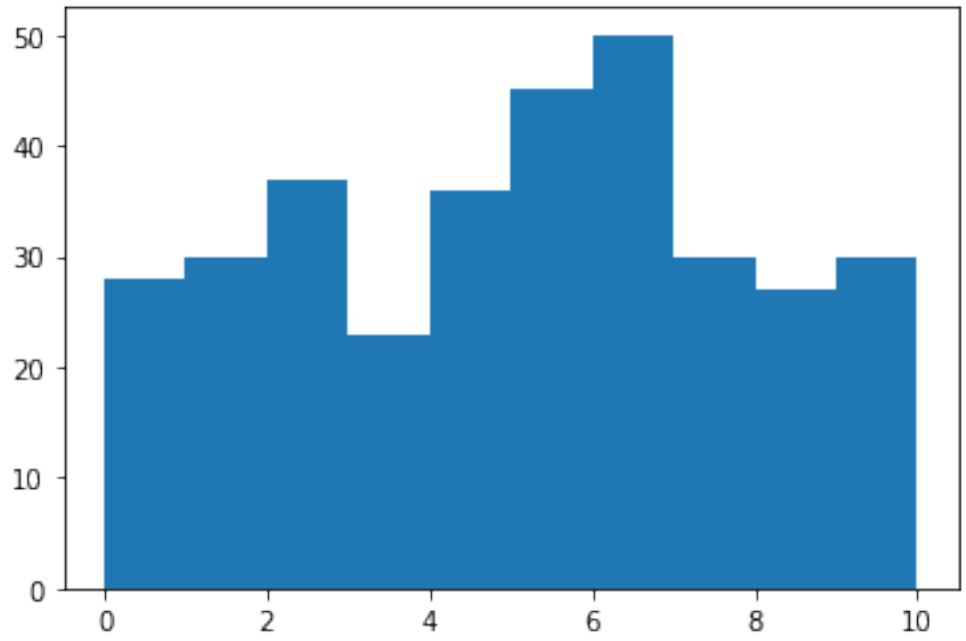


5.1.3 matplotlib.pyplot

matplotlib.pyplot を使って、ヒストグラムの表示と度数分布を求めることも可能です。

まずヒストグラムを作成します。

```
counts, digits, chart = pyplot.hist(
    split_codes(),
    bins=range(0,11)
)
```



matplotlib.hist() 関数は、ビン毎の計数値 (count)、bin の端の値 (digits) そして、ヒストグラムのオブジェクトを値として返してきます。ですから、次のようなスクリプトで、各ビンのラベルと係数値を印刷します。digits の要素数は counts の要素数より 1 多いことに注意が必要です。

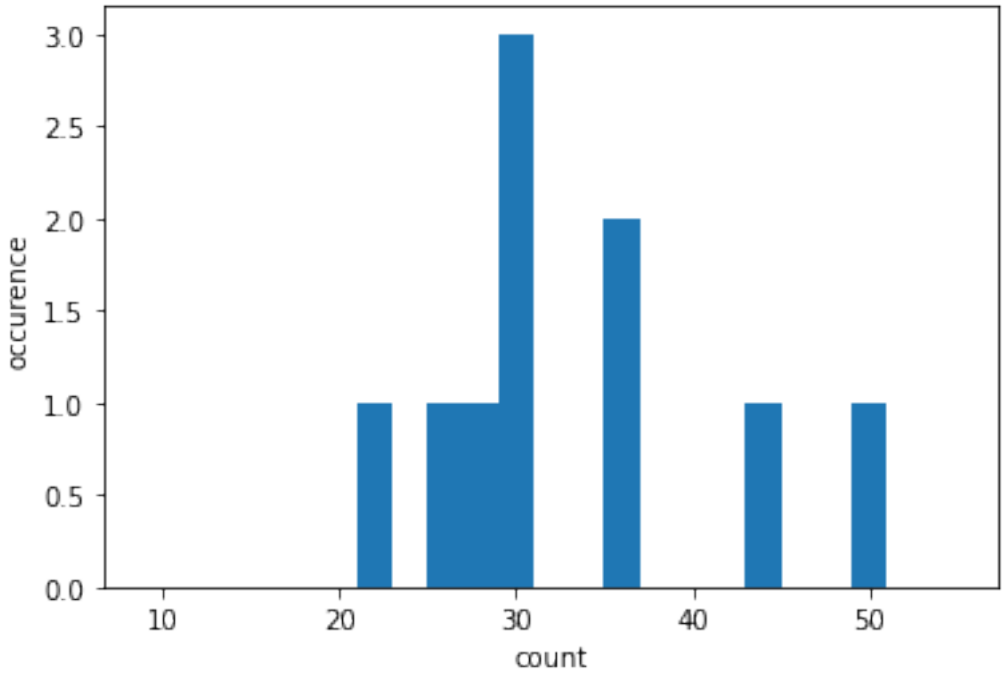
```
for p in (zip(digits,counts)):
    print(p)
print("average:", sum(counts)/len(counts))
print(len(digits), ">",len(counts))
```

```
(0, 28.0)
(1, 30.0)
(2, 37.0)
(3, 23.0)
(4, 36.0)
(5, 45.0)
(6, 50.0)
(7, 30.0)
(8, 27.0)
(9, 30.0)
average: 33.6
11 > 10
```

次に、数字の出現回数の分布をプロットしてみます。“5”および“6”の出現回数が突出していることが見て取れます。

```
pyplot.hist(
    counts,
    histtype="stepfilled",
    density=False,
    align="left",
    bins=range(34-3*8,34+3*8,2),
)
pyplot.gca().set_xlabel("count")
pyplot.gca().set_ylabel(u"occurence");
```

```
Text(0, 0.5, 'occurence')
```



5.2 今年の「13 日の金曜日」は何回？

— あるいは、僕が「13 日の金曜日」について知っている 2、3 のこと —

5.2.1 また「13 日の金曜日」がやってくる。

2015 年のある日、次の打ち合わせの確認でカレンダーを眺めると、その日が「13 日の金曜日」になることに気がついた。なんだかついこの間の打ち合わせも「13 日の金曜日」だったようが気がする。なぜだろうと気になったので、ちょっと調べてみようと Python の Notebook を開いてみた。

5.2.2 python プログラムをつかって「13 日」の金曜日は他の曜日に比べて多いのか少ないのか？を調べてみる。

手始めに、”「13 日の金曜日」が他の曜日に比べ多いとか、少ないとかいうことがあるのか？” を確かめてみることから初めてみよう。

まずは、いつものおまじないと、日付データを取り扱うための datetime モジュールをインポートしておこう。

```
#!python
#-*- coding:utf-8 -*-
# python2/python3 の互換性のために
from __future__ import print_function
# 日付データを取り扱うためのモジュールをインポートしておく。
import datetime
```

準備が出来たところで m2021 年から 2121 年までの 100 年間にある毎月の 13 の曜日の頻度を数え上げてみればいいだろう。

```
#  月曜から日曜の曜日の名前を定義しておきます。
wdname_e=("mon","tue","wed","thu","fri","sat","sun")
# 日本語での名前
wdname_j=(u"月",u"火",u"水",u"木", u"金", u"土", u"日")
wdname=wdname_j

wdcount={} #曜日毎の回数を覚えておくための辞書型データを用意する。

ny=100 #100 年分を調べてみます。
for y in range(2021,2021+ny): #range(2021,2022)->(2021,2022,...,2120)
```

(次のページに続く)

(前のページからの続き)

```
    for m in range(1,13,1): #range(1,13,1) -> (1,2,...,12)
        d=datetime.datetime(y,m,13) # y 年 m 月 13 日の日付データを作成。
        wd=d.weekday() # その日の曜日を取り出す。
        wdcount[wd] = wdcount.get(wd,0) + 1 #wdcount の key が wd のデータ
に 1 を加える。

print("曜日","回数" )
for k in range(7): # 曜日毎の登場回数を印刷する。
    print (wdname[k], wdcount[k])
```

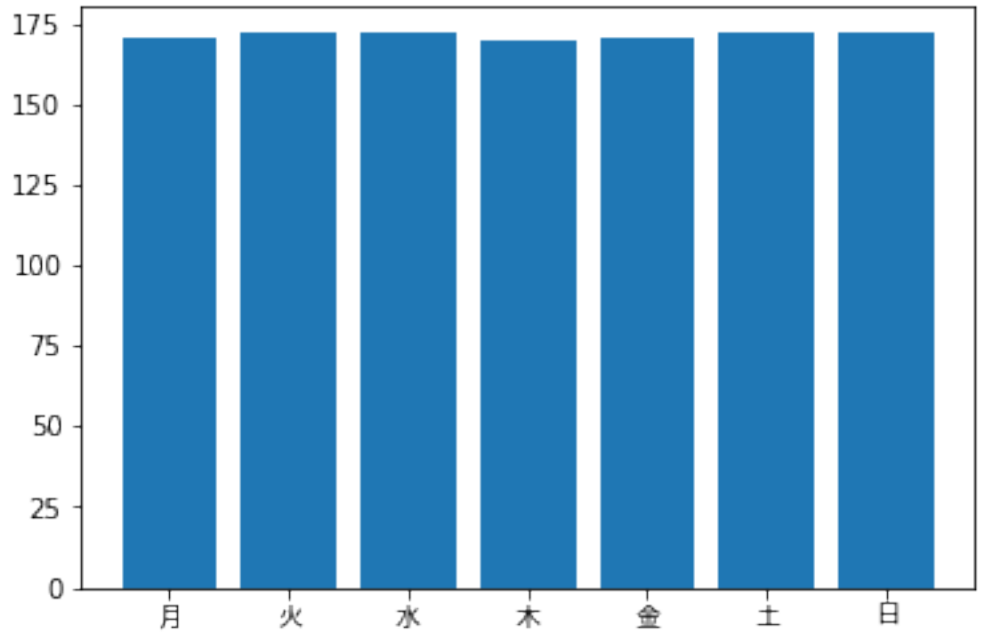
```
曜日 回数
月 171
火 172
水 172
木 170
金 171
土 172
日 172
```

これらは単純な平均値 $ 171.4= 100:raw-latex:*times*:raw-latex:`frac{12}{7}`$ に一致している。当たり前だけど、特に 13 日は金曜日が多いとか、その逆というわけではない。この分布を曜日毎の回数を示すバーグラフにすれば、わかりやすいだろうか。

5.2.3 グラフを書いてみる。

グラフを描くためのモジュールはいくつかあるが、これもいつものように matplotlib を使うことにする。matplotlib のサブモジュール pyplot は簡単なグラフの作成にはピッタリだ。細部にまで凝るなら matplotlib のオブジェクトを使いこなすことになる。

```
import matplotlib.pyplot as pyplot  #matplotli.pyplot を pyplot という名
前で使うことにする。
pyplot.bar(wdcount.keys(), wdcount.values(),align="center") #バーグラフ
を使ってグラフ化する。
# pyplot.bar(*zip(*wdcount.items()),align="center") #こういう書き方もあ
る。
pyplot.xticks(range(7), wdname_j, font="IPAexGothic") # x 軸のラベルを曜
日にしておこう。
pyplot.draw()
```



5.2.4 毎年の「13 日の金曜日」の回数を数えてみよう。

このように、毎年の「13 日の金曜日」は毎年 12/7=1.7 回あるわけだが、1 回の年もあれば、2 回の年もあるだろう。 もう一度 Python スクリプトをつかって、毎年の「13 日の金曜日」の回数を数えて確かめなければならない。

```
F13inY={} #年毎の「13 日の金曜日」の回数を記録する辞書型データ `F13inY`を用意する。
def count_F13inY(year):
    count=0 # カウンタ count を 0 クリアする。
    for m in range(1,13,1): # 毎月の
        d=datetime.datetime(year,m,13) #13 日が
        wd=d.weekday() # 曜日が
        if wd == 4:# 金曜日 (4) だったら
            count +=1 # カウンタを 1 増やす。
    return count

for y in range(2015,2115):
    count=count_F13inY(y)
    F13inY[count] = F13inY.get(count,0)+1 # 一年あたりの「13 日の金曜日」の回数が count となる年数を 1 増やす

for k, v in F13inY.items(): # 一年あたりの「13 日の金曜日」の回数が k のなる年数を印刷する。
    print (k,"occurense ", v,"times")
```

```
3 occurense  16 times
1 occurense  43 times
2 occurense  41 times
```

プログロラムの結果では、2015-2114 の 100 年間で、「13 日の金曜日」が 1 日しか無い年が 43 回、2 日ある年が 42 回、そしてなんと「年に 3 日の「13 日の金曜日」を持つ」当たり年が 16 回ある。まあ、この平均は確かに、

```
(43+2*42+3*15)/100.
```

```
1.72
```

となり、 平均すれば均等な確率による結果通りになってはいるんだけれど。

5.2.5 年に 3 回の「13 日の金曜日」がある年を探して。

それではとこの 100 年のうち「13 日の金曜日」が 3 回有るのはどの年かを調べてみたくなる。次のプログラムを使うと、2015-1214 年の 100 年間には、

2015, 2026, 2037, 2040, 2043, 2054, 2065, 2068, 2071, 2082, 2093, 2096, 2099, 2105, 2108, そして 2111

の 16 回であることがわかる。

確かに今年 (2015) は 3 回の「13 日の金曜日」がある年で、「またか」と思ったのもそのためだろう。

このプログラムをつかえば、次に「年に 3 回の『13 日の金曜日』」がある次の年がわかってしまう。それはなんと 2026 年だとプログラムは教えてくれる。当分は「えっ、また『13 日の金曜日』」と感じる様な年は巡ってこないわけだ。

```
years=[]
for y in range(2015,2115):
    count=count_F13inY(y)
    if count >=3:
        print ("year",y, "has ",count, "day(s) of \"Friday 13th\"")
        years.append(y)
print(len(years), years)
```

```
year 2015 has  3 day(s) of "Friday 13th"
year 2026 has  3 day(s) of "Friday 13th"
year 2037 has  3 day(s) of "Friday 13th"
```

(次のページに続く)

(前のページからの続き)

```
year 2040 has  3 day(s) of "Friday 13th"
year 2043 has  3 day(s) of "Friday 13th"
year 2054 has  3 day(s) of "Friday 13th"
year 2065 has  3 day(s) of "Friday 13th"
year 2068 has  3 day(s) of "Friday 13th"
year 2071 has  3 day(s) of "Friday 13th"
year 2082 has  3 day(s) of "Friday 13th"
year 2093 has  3 day(s) of "Friday 13th"
year 2096 has  3 day(s) of "Friday 13th"
year 2099 has  3 day(s) of "Friday 13th"
year 2105 has  3 day(s) of "Friday 13th"
year 2108 has  3 day(s) of "Friday 13th"
year 2111 has  3 day(s) of "Friday 13th"
16 [2015, 2026, 2037, 2040, 2043, 2054, 2065, 2068, 2071, 2082, 2093,␣
↪2096, 2099, 2105, 2108, 2111]
```

プログラムを短くするためにこんな書き方もできるらしい。

```
years=[y for y in range(2015,2115) if count_F13inY(y)>=3 ] # 3回(以上)の 「13日の金曜日」がある年のリスト
for y in years:
    print ("year",y, "has ","more than 3 days of \"Friday 13th\"")
print()
print(len(years), years)
```

```
year 2015 has  more than 3 days of "Friday 13th"
year 2026 has  more than 3 days of "Friday 13th"
year 2037 has  more than 3 days of "Friday 13th"
year 2040 has  more than 3 days of "Friday 13th"
year 2043 has  more than 3 days of "Friday 13th"
year 2054 has  more than 3 days of "Friday 13th"
year 2065 has  more than 3 days of "Friday 13th"
year 2068 has  more than 3 days of "Friday 13th"
year 2071 has  more than 3 days of "Friday 13th"
year 2082 has  more than 3 days of "Friday 13th"
year 2093 has  more than 3 days of "Friday 13th"
year 2096 has  more than 3 days of "Friday 13th"
year 2099 has  more than 3 days of "Friday 13th"
year 2105 has  more than 3 days of "Friday 13th"
year 2108 has  more than 3 days of "Friday 13th"
year 2111 has  more than 3 days of "Friday 13th"
```

(次のページに続く)

(前のページからの続き)

```
16 [2015, 2026, 2037, 2040, 2043, 2054, 2065, 2068, 2071, 2082, 2093,␣
↪2096, 2099, 2105, 2108, 2111]
```

この結果をよくみると、2015-2065 までは、「13 日の金曜日が年に 3 回ある年」は大体 11 年あるいは 13 年の間をおいて現ている。しかし (2065,2068,2071) および (2093,2096, 2099) は三年おきの間隔で“「13 日の金曜日が年に 3 回ある年”が繰り返している。

これは過去の繰り返しも調べてみる必要があるだろう。

```
years=[]
for y in range(1915,2016):
    count=count_F13inY(y)
    if count >=3:
        print ("year",y, "has ",count, "day(s) of \"Friday 13th\"")
        years.append(y)
print(len(years), years)
```

```
year 1925 has  3 day(s) of "Friday 13th"
year 1928 has  3 day(s) of "Friday 13th"
year 1931 has  3 day(s) of "Friday 13th"
year 1942 has  3 day(s) of "Friday 13th"
year 1953 has  3 day(s) of "Friday 13th"
year 1956 has  3 day(s) of "Friday 13th"
year 1959 has  3 day(s) of "Friday 13th"
year 1970 has  3 day(s) of "Friday 13th"
year 1981 has  3 day(s) of "Friday 13th"
year 1984 has  3 day(s) of "Friday 13th"
year 1987 has  3 day(s) of "Friday 13th"
year 1998 has  3 day(s) of "Friday 13th"
year 2009 has  3 day(s) of "Friday 13th"
year 2012 has  3 day(s) of "Friday 13th"
year 2015 has  3 day(s) of "Friday 13th"
15 [1925, 1928, 1931, 1942, 1953, 1956, 1959, 1970, 1981, 1984, 1987,␣
↪1998, 2009, 2012, 2015]
```

1915-2015 の間では、(1925, 1928, 1931), (1953, 1956, 1959) および (2009,2012,2015) も「三年おきの間隔で「13 日の金曜日が年に 3 回ある年が繰り返す」時期に当たっていることが判明する。その意味でも 2015 年はちょっとだけ特別な年であったわけだ。

次の「三年おきの間隔で「13 日の金曜日が年に 3 回ある年が繰り返す」期間は 2037 年ー 2043 年ということになる。その頃にはまた、”最近 13 日の金曜日が多いな”と思ったり

するのだろうか。

（終)

5.2.6 おまけ

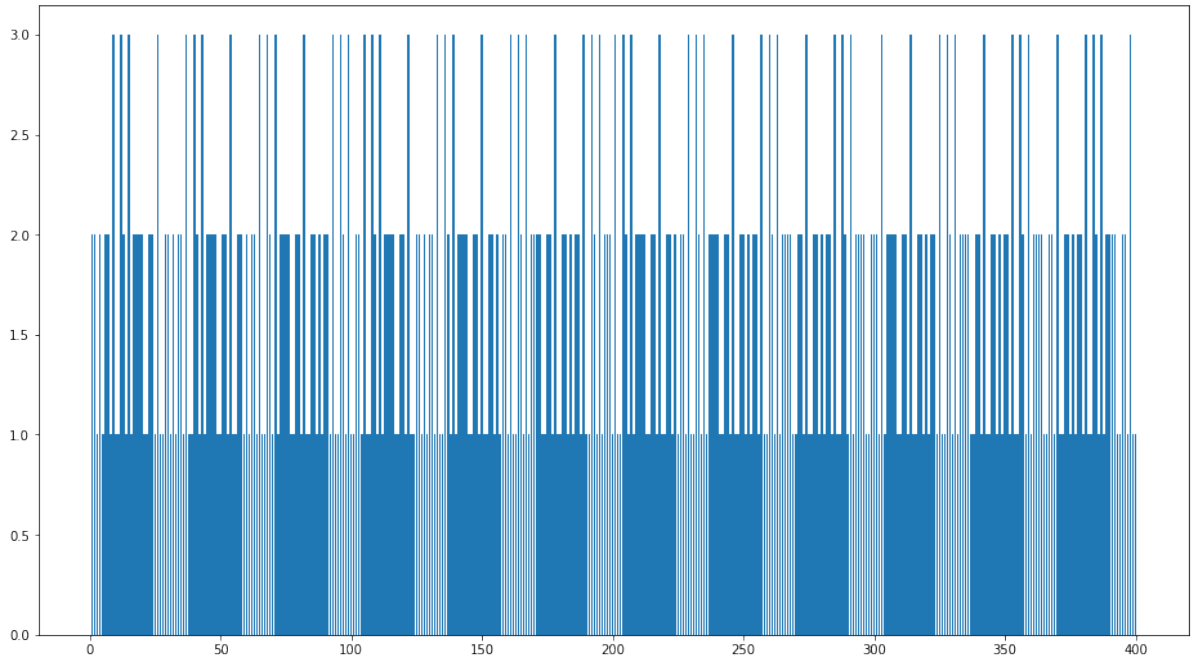
『1 年間にある 13 日の金曜日の日数』の分布。

グレゴリオ歴は 400 年でちょうど、1460972 日、20870 週となるので、分布のパターンは 400 年の周期を持つ。もし、この日数がちょうど 7 で割り切れるここで考えているような曜日の繰り返しは 2800 年の周期をもっていただろう。

この 400 年の周期の間に『1 年間にある 13 日の金曜日の日数』の分布をグラフにしてみよう。

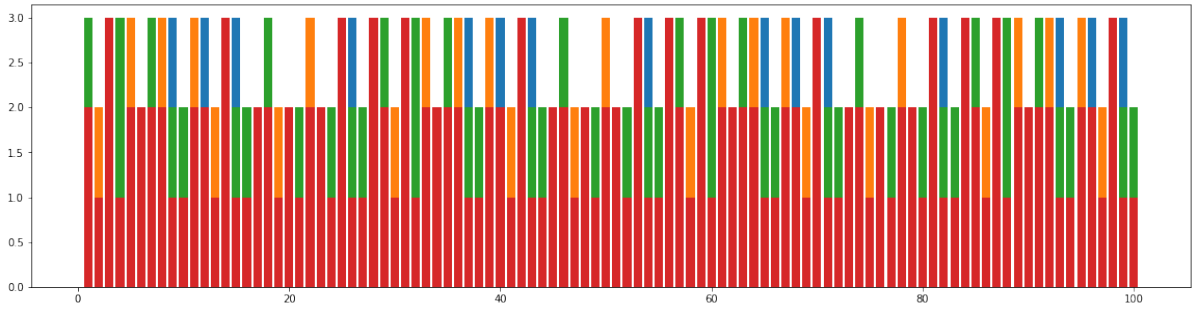
```
## %%timeit # 実行時間測定のためのおまじない (cell magic)
ymax=401
# 478 ms ± 7.68 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)␣
↪on Intel Mac with ymax=9999
# 220 ms ± 8.01 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)␣
↪on M1 Mac with arc -x86_64 and ymax=9999
F13inY={}

for y in range(1,ymax):
    count=count_F13inY(y)
    F13inY[count] = F13inY.get(count,0)+1

for k in F13inY:
    print (k, "occurense ", F13inY[k], "times")

years=range(1,ymax)
pyplot.figure(figsize=(16,9))
pyplot.bar(years,[count_F13inY(y) for y in years])
pyplot.draw()
```

```
2 occurense  170 times
1 occurense  171 times
3 occurense  59 times
```



グレゴリオ歴では、4 年に一回ある閏年が、100 年に一回なくなってしまう。この効果を見るために、100 年毎に区切って分布をプロットしてみよう、

```
ny=100
years=range(1,1+ny)
pyplot.figure(figsize=(20,5))
for n in range(1,401,100):
    pyplot.bar(years,[count_F13inY(y) for y in range(n,n+ny)])
pyplot.draw()
```



100 年

同じ曜日を持つ月

毎月の日数は、閏年の 2 月を除いては、いつも同じ。従って、1 月のある日付の曜日と他の月の同じ日付の曜日は一定の関係がある。閏年ではない、ある年の毎月の 13 日の曜日を調べて見る。

```
y=2021;print([(i,wdname_j[datetime.date(y,i,13).weekday()]) for i in␣
↪range(1,13)])
```

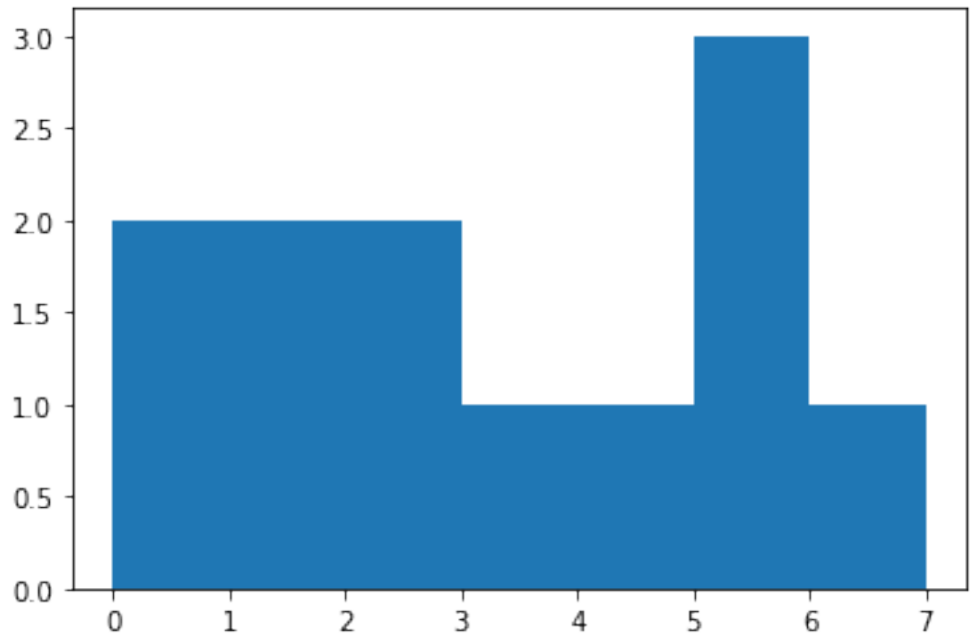
```
[(1, '水'), (2, '土'), (3, '土'), (4, '火'), (5, '木'), (6, '日'), (7,
 '火'), (8, '金'), (9, '月'), (10, '水'), (11, '土'), (12, '月')]
```

このように、閏年でない年は （ 1 月、10 月）, （2 月, 3 月, 11 月）、(4 月、7 月)、 (9 月,12 月） がそれぞれ同じ曜日になる。

2 月がちょうど四週間＝ 28 日なので、2 月と 3 月は 1-28 日の曜日が繰り返すのはよくわかる。

```
pyplot.hist([datetime.date(y,i,13).weekday() for i in range(1,13)],
↪bins=range(0,8));
```

```
(array([2., 2., 2., 1., 1., 3., 1.]),
 array([0, 1, 2, 3, 4, 5, 6, 7]),
 <BarContainer object of 7 artists>)
```



では閏年にはどうなるだろうか？　2 月が 29 日あるので、3 月の曜日は 2 月の同じ日付から一つズレるはずだ。

```
y=2020;print([(i,wdname_j[datetime.date(y,i,13).weekday()]) for i in␣
↪range(1,13)])
```

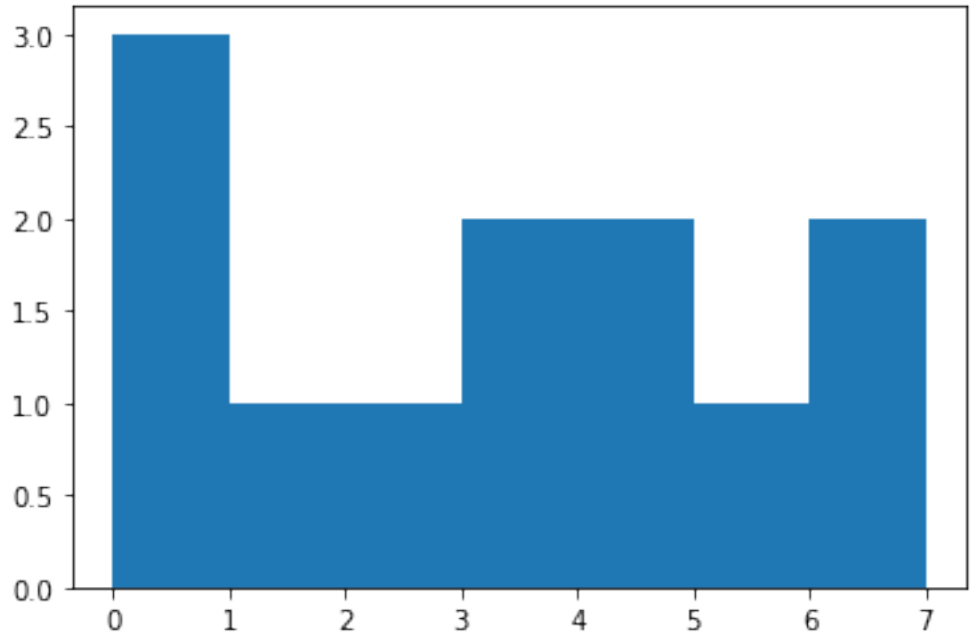
```
[(1, '月'), (2, '木'), (3, '金'), (4, '月'), (5, '水'), (6, '土'), (7,
 '月'), (8, '木'), (9, '日'), (10, '火'), (11, '金'), (12, '日')]
```

というわけで、閏年には、(1 月、4 月,7 月）, (2 月、8 月）, (3 月、11 月), (9 月、12 月) がそれぞれ同じ曜日となるこわけだ。

ついでに、どの年でも 3 月と 11 月、4 月と 7 月、9 月と 12 月はそれぞれ,31 日を除けば、同じ日付の日は同じ曜日となる。

```
pyplot.hist([datetime.date(y,i,13).weekday() for i in range(1,13)],
↪bins=range(0,8));
```

```
(array([3., 1., 1., 2., 2., 1., 2.]),
 array([0, 1, 2, 3, 4, 5, 6, 7]),
 <BarContainer object of 7 artists>)
```



最後に月の 31 日の曜日の関係をみてみると。

閏年の年には、

```
y=2020;print([(i,wdname_j[datetime.date(y,i,31).weekday()]) for i in␣
↪(1,3,5,7,8,10,12)])
```

```
[(1, '金'), (3, '火'), (5, '日'), (7, '金'), (8, '月'), (10, '土'),␣
↪(12, '木')]
```

閏年でない年には、

```
y=2021;print([(i,wdname_j[datetime.date(y,i,31).weekday()]) for i in␣
↪(1,3,5,7,8,10,12)])
```

```
[(1, '日'), (3, '水'), (5, '月'), (7, '土'), (8, '火'), (10, '日'),␣
↪(12, '金')]
```

となるので、31 日は、『閏年には、一月と七月、それ以外の年には一月と十月が同じ曜日となる。』ことがわかる。

具体的には、1 太陽年＝ 365.24219040 - 6.15 × 10-6 T [日]。ただし、T はユリウス世紀数. グレゴリオ暦からのずれは 10000 年に 3 日分短い。

```
>>> 365+1/4-1/100+1/400-1/3200+1/80000
365.2422
```

だから、3200=8*400 年に一回さらに閏年を減らし,80000=25*3200=200*400 年の倍数は閏年とすれば、ちょうどいい。もっともそれまで平均太陽日が現在のままということはないのでこのルールもあまり意味はないだろう。

5.3 From Web to Plot

python プログラムの実例の一つとして、Internet で入手可能なデータを取り込んで、グラフ化してみます。東洋経済オンラインが web で公開している、Covid19 の時系列データを取り込み、グラフ化してみます。

5.3.1 web 上のデータをダウンロード

まずは web 上で公開されている csv データをローカルファイルにダウンロードしてみます。python3 では urllib モジュールの request サブモジュールを使って、指定した URL の内容をプログラムを使って読み込むことができます。このモジュールをプログラムで利用するために、このモジュールをインポート（import) します。

python には豊富な標準ライブラリモジュールが用意されています。また、第三者によって提供されているモジュールも豊富です。Python でのプログラム開発では、これらのモジュールをうまく使いこなすことがコツの一つです。文献／ web page を利用して必要なモジュールを探してみましょう。

```
import urllib.request
```

ここでは、urllib.request モジュールの中の urlopen() 関数だけが必要です。　この様な場合には、urlopen 関数だけを選択的にインポートすることができます。

```
from urllib.request import urlopen
```

次の行は、 macos 上の python で https::に接続する際にシステムの SSL 証明書を使う様に指示しています。以下の二行をコメントアウトして、実行に問題がなければ不要です。（私の環境では、必要となっています。）

```
import os, certifi
os.environ["SSL_CERT_FILE"]=certifi.where()
```

次にデータをダウンロードする URL のリストを作っておきます。このリストに dataurls という名前をつけておきます。

###　NHK のデータソースはこちら https://www3.nhk.or.jp/n-data/opendata/coronavirus/nhk_news_covid19_prefectures_daily_data.csv

https://www3.nhk.or.jp/n-data/opendata/coronavirus/nhk_news_covid19_prefectures_daily_data.csv

```
dataurls=(
        "https://toyokeizai.net/sp/visual/tko/covid19/csv/pcr_positive_
↪daily.csv",
```

(次のページに続く)

(前のページからの続き)

```
        "https://toyokeizai.net/sp/visual/tko/covid19/csv/pcr_tested_
↪daily.csv",
        "https://toyokeizai.net/sp/visual/tko/covid19/csv/cases_total.
↪csv",
        "https://toyokeizai.net/sp/visual/tko/covid19/csv/recovery_
↪total.csv",
        "https://toyokeizai.net/sp/visual/tko/covid19/csv/death_total.
↪csv",
        "https://toyokeizai.net/sp/visual/tko/covid19/csv/pcr_case_
↪daily.csv",
        "https://toyokeizai.net/sp/visual/tko/covid19/csv/severe_daily.
↪csv",
        "https://toyokeizai.net/sp/visual/tko/covid19/csv/effective_
↪reproduction_number.csv",
        "https://toyokeizai.net/sp/visual/tko/covid19/csv/demography.
↪csv",
        "https://toyokeizai.net/sp/visual/tko/covid19/csv/prefectures.
↪csv"
    )
```

5.3.2 東洋経済　オンライン　「新型コロナウイルス国内感染の状況」 (https://toyokeizai.net/sp/visual/tko/covid19/) が使っているデータソース

結局上記の NHK と同じ

厚生労働省オープンデータ「陽性者数」pcr_positive_daily.csv
(https://toyokeizai.net/sp/visual/tko/covid19/csv/pcr_positive_daily.csv)

厚生労働省オープンデータ「PCR　検査実施人数」pcr_tested_daily.csv
(https://toyokeizai.net/sp/visual/tko/covid19/csv/pcr_tested_daily.csv)

厚生労働省オープンデータ「入院治療等を要する者の数」cases_total.csv
(https://toyokeizai.net/sp/visual/tko/covid19/csv/cases_total.csv)

厚生労働省オープンデータ「退院又は療養解除となった者の数」recovery_total.csv
(https://toyokeizai.net/sp/visual/tko/covid19/csv/recovery_total.csv)

厚生労働省オープンデータ「死亡者数」death_total.csv
(https://toyokeizai.net/sp/visual/tko/covid19/csv/death_total.csv)

厚生労働省オープンデータ「PCR 検査の実施件数」pcr_case_daily.csv (https://toyokeizai.net/sp/visual/tko/covid19/csv/pcr_case_daily.csv)

公表日ごとの全国の重症者数 severe_daily.csv (https://toyokeizai.net/sp/visual/tko/covid19/csv/severe_daily.c

日別全国の実効再生産数 effective_reproduction_number.csv (https://toyokeizai.net/sp/visual/tko/covid19/csv/effective_reproduction_number.csv)

年代別の国内発生動向 demography.csv (https://toyokeizai.net/sp/visual/tko/covid19/csv/demography.csv)

都道府県別の発生動向 prefectures.csv (https://toyokeizai.net/sp/visual/tko/covid19/csv/prefectures.csv)

python3 にはデータ (オブジェクト) の並びを表現するためにふたつのデータ形, リスト ([,]) とタプル ((,)) が用意されています。リストはデータを作成した後で、要素を変更 (取り替え) することができますが、タプルはその要素を変更 (取り替え) することができません。URL のリストはプログラム中で変更する必要はありませんから、`dataurls` は URL のタプルとして定義しました。

次に、URL で指定される csv ファイル (csv:comma separated value, カンマ区切り形式) をローカルファイルにダウンドードする関数 `load()` を定義します。

```
def load(dataurl):
    fn=dataurl.split("/")[-1]
    with urlopen(dataurl) as inf , open(fn,"wb") as outf:
        print ("downloading :", fn)
        data=inf.read()
        outf.write(data)
```

`def` キーワードを使って、関数 `load（）` を定義しています。`load` の引数は `dataurl` です。関数の本体は、`:`に続く文で定義します。関数本体のプログラム文は `def` キーワードに対してインデントされています。同様に `with` 文の本体 (`:`のあとの文) も `with` に対してインデントされていることに気をつけてください。インデントに TAB を使うこともできますが、1byte 文字の空白`' '` を使うことを推奨します。この python でのブロック構造の表現はちょっと特殊ですが、出来上がったプログラムが読みやすくなることを狙って採用されています。

この関数 `load()` は `dataurl` の最後の要素であるファイル名と同じ名前のローカルファイルにデータを保存します。`dataurl` の文字列は `"/"` を区切りとして切り分けられます `dataurl.split("/")` 。切り分けたリストの最後の要素 `[-1]` がファイル名 `xxxx.csv` です。

`urlopen()` で確立した web との接続を `inf`, 書き込み用 (`w`) にオープンしたバイナリ形式データ (`b`) のファイル指示子を `outf` として使うことを `with ... as ...,:`構文で宣言しています。

この構文 (context manager) を使うことで、確実にこれらのフィアル／接続がこのセクション実行後に自動的に `close` されることが保証されます。

あとはデータを読み込んで、data=inf.read(), それをファイルに書き出す、outf.write(data)、だけです。

dataurls の各要素についてこの関数を呼び出します。

```
for dataurl in dataurls:
    load(dataurl)
```

```
downloading : pcr_positive_daily.csv
downloading : pcr_tested_daily.csv
downloading : cases_total.csv
downloading : recovery_total.csv
downloading : death_total.csv
downloading : pcr_case_daily.csv
downloading : severe_daily.csv
downloading : effective_reproduction_number.csv
downloading : demography.csv
downloading : prefectures.csv
```

5.3.3 CSV から SQL Dtabase への変換

データの検索などの操作は CVS データを SQL Database（ここでは sqlite3 データベースを利用）に変換することで、簡単に行えるようになります。この変換は色々な手法がありますが、今回試してみた結果 pandas の dataframe を経由する方法が簡単確実です。

pandas は最近話題の anacond/miniconda などでも使われている python でのデータ処理向けのライブラリです。dataframe と呼ばれるデータ構造を提供しており、単純なアレイ構造を超えた取扱を実現しています。ただ、使いこなしにはある程度の練習が必要な様です。ここでは dataframe の備える cvs/sql への読み書きの機能だけを使います。

まず dataframe 形のデータを使える様に、pandas モジュールをインポートしましょう。sqlite3, os もインポートしておきます。os.path モジュールからは splitext 関数だけをインポートしておきます。

```
import pandas
import sqlite3
from os.path import splitext
import os
```

次に一つの dataurl について、先ほどダウンロードした csv ファイルを読み込み、sql データベースのテーブルとして追加する関数 from_cvs_to_sql() 関数を定義します。

```
def from_csv_to_df(dataurl):
    fn=dataurl.split("/")[-1]
    df=pandas.read_csv(fn)
    return df

def from_csv_to_sql(dataurl):
    fn=dataurl.split("/")[-1]
    df=from_csv_to_df(fn)
    with sqlite3.connect('covid19.db') as con:
        print ("convert ",splitext(fn)[0])
        df.to_sql(splitext(fn)[0],con)
    os.remove(fn)
```

```
import os, certifi
os.environ["SSL_CERT_FILE"]=certifi.where()

try:
    os.remove('covid19.db')
except FileNotFoundError:
    pass
for dataurl in dataurls:
    from_csv_to_sql(dataurl)
```

```
convert  pcr_positive_daily
convert  pcr_tested_daily
convert  cases_total
convert  recovery_total
convert  death_total
convert  pcr_case_daily
convert  severe_daily
convert  effective_reproduction_number
convert  demography
convert  prefectures
```

```
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
↪packages/pandas/core/generic.py:2778: UserWarning: The spaces in␣
↪these column names will not be changed. In pandas versions < 0.14,␣
↪spaces were converted to underscores.
  sql.to_sql(
```

5.3.4 sql database の確認

作成されたデータベースの schema をチェックしてみましょう。sqlite3 ではデータベースに含まれるテーブルなどの schme 情報は、sqlite_master テーブルに存在します。この中の SQL 欄をみることで、それぞれのテーブルが持つカラムとそのデータ型を知ることもできますが、sqlite3 では PRAGMA 機能を使って、それらの情報を取り出すこともできます。

```
def dump_table_info(dbf):
    db=sqlite3.connect(dbf)
    cur=db.cursor()
    table_info =cur.execute("select name,sql from sqlite_master where␣
↪type = 'table'").fetchall()
    for tbl,sql in table_info:
        print ("******* table: {:s} *******".format(tbl))
        print ("sql:",sql)
        print ("------ field info -----")
        cur.execute("PRAGMA table_info({:s});".format(tbl))
        print ("rowid","name","type","nullable","defalut", "pk(primary␣
↪key)")
        for e in cur.fetchall():
            print (*e)

    print("****** data count in tables ******")
    print( "table\tcout")
    for tbl,sql in table_info:
        cur.execute("select count(*) from %s;"%tbl)
        print (tbl,"\t", cur.fetchone()[0])

dump_table_info("covid19.db")
```

```
*** table: pcr_positive_daily ***
sql: CREATE TABLE "pcr_positive_daily" (
"index" INTEGER,
  "日付" TEXT,
  "PCR 検査陽性者数(単日)" REAL
)
------ field info -----
rowid name type nullable defalut pk(primary key)
0 index INTEGER 0 None 0
1 日付 TEXT 0 None 0
2 PCR 検査陽性者数(単日) REAL 0 None 0
*** table: pcr_tested_daily ***
```

```
sql: CREATE TABLE "pcr_tested_daily" (
"index" INTEGER,
  "日付" TEXT,
  "PCR 検査実施件数(単日)" REAL
)
------ field info -----
rowid name type nullable defalut pk(primary key)
0 index INTEGER 0 None 0
1 日付 TEXT 0 None 0
2 PCR 検査実施件数(単日) REAL 0 None 0
*** table: cases_total ***
sql: CREATE TABLE "cases_total" (
"index" INTEGER,
  "日付" TEXT,
  "入院治療を要する者" REAL
)
------ field info -----
rowid name type nullable defalut pk(primary key)
0 index INTEGER 0 None 0
1 日付 TEXT 0 None 0
2 入院治療を要する者 REAL 0 None 0
*** table: recovery_total ***
sql: CREATE TABLE "recovery_total" (
"index" INTEGER,
  "日付" TEXT,
  "退院、療養解除となった者" REAL
)
------ field info -----
rowid name type nullable defalut pk(primary key)
0 index INTEGER 0 None 0
1 日付 TEXT 0 None 0
2 退院、療養解除となった者 REAL 0 None 0
*** table: death_total ***
sql: CREATE TABLE "death_total" (
"index" INTEGER,
  "日付" TEXT,
  "死亡者数" REAL
)
------ field info -----
rowid name type nullable defalut pk(primary key)
0 index INTEGER 0 None 0
1 日付 TEXT 0 None 0
2 死亡者数 REAL 0 None 0
```

```
*** table: pcr_case_daily ***
sql: CREATE TABLE "pcr_case_daily" (
"index" INTEGER,
  "日付" TEXT,
  "国立感染症研究所" INTEGER,
  "検疫所" INTEGER,
  "地方衛生研究所・保健所" INTEGER,
  "民間検査会社（主に行政検査）" REAL,
  "大学等" REAL,
  "医療機関" REAL,
  "民間検査会社（主に自費検査）" REAL
)
------ field info -----
rowid name type nullable defalut pk(primary key)
0 index INTEGER 0 None 0
1 日付 TEXT 0 None 0
2 国立感染症研究所 INTEGER 0 None 0
3 検疫所 INTEGER 0 None 0
4 地方衛生研究所・保健所 INTEGER 0 None 0
5 民間検査会社（主に行政検査） REAL 0 None 0
6 大学等 REAL 0 None 0
7 医療機関 REAL 0 None 0
8 民間検査会社（主に自費検査） REAL 0 None 0
*** table: severe_daily ***
sql: CREATE TABLE "severe_daily" (
"index" INTEGER,
  "日付" TEXT,
  "重症者数" REAL
)
------ field info -----
rowid name type nullable defalut pk(primary key)
0 index INTEGER 0 None 0
1 日付 TEXT 0 None 0
2 重症者数 REAL 0 None 0
*** table: effective_reproduction_number ***
sql: CREATE TABLE "effective_reproduction_number" (
"index" INTEGER,
  "日付" TEXT,
  "実効再生産数" REAL
)
------ field info -----
rowid name type nullable defalut pk(primary key)
0 index INTEGER 0 None 0
```

```
1 日付 TEXT 0 None 0
2 実効再生産数 REAL 0 None 0
*** table: demography ***
sql: CREATE TABLE "demography" (
"index" INTEGER,
  "year" INTEGER,
  "month" INTEGER,
  "date" INTEGER,
  "age_group" TEXT,
  "tested_positive" INTEGER,
  "hospitalized" INTEGER,
  "serious" INTEGER,
  "death" INTEGER
)
------ field info -----
rowid name type nullable defalut pk(primary key)
0 index INTEGER 0 None 0
1 year INTEGER 0 None 0
2 month INTEGER 0 None 0
3 date INTEGER 0 None 0
4 age_group TEXT 0 None 0
5 tested_positive INTEGER 0 None 0
6 hospitalized INTEGER 0 None 0
7 serious INTEGER 0 None 0
8 death INTEGER 0 None 0
*** table: prefectures ***
sql: CREATE TABLE "prefectures" (
"index" INTEGER,
  "year" INTEGER,
  "month" INTEGER,
  "date" INTEGER,
  "prefectureNameJ" TEXT,
  "prefectureNameE" TEXT,
  "testedPositive" REAL,
  "peopleTested" REAL,
  "hospitalized" REAL,
  "serious" REAL,
  "discharged" REAL,
  "deaths" REAL,
  "effectiveReproductionNumber" REAL
)
------ field info -----
rowid name type nullable defalut pk(primary key)
```

```
0 index INTEGER 0 None 0
1 year INTEGER 0 None 0
2 month INTEGER 0 None 0
3 date INTEGER 0 None 0
4 prefectureNameJ TEXT 0 None 0
5 prefectureNameE TEXT 0 None 0
6 testedPositive REAL 0 None 0
7 peopleTested REAL 0 None 0
8 hospitalized REAL 0 None 0
9 serious REAL 0 None 0
10 discharged REAL 0 None 0
11 deaths REAL 0 None 0
12 effectiveReproductionNumber REAL 0 None 0
** data count in tables **
table       cout
pcr_positive_daily   570
pcr_tested_daily     550
cases_total          551
recovery_total       557
death_total          541
pcr_case_daily       536
severe_daily         550
effective_reproduction_number        525
demography   10
prefectures          26320
```

この様に、csv に含まれるデータから、SQL テーブルの各コラムのデータ型も適切に選択されていることがわかります。

5.3.5 データのプロット

作成されたデータベースから、茨城県の 2021 年の日毎の陽性者数の推移をグラフにしてみます。グラフのプロットには matplotlib モジュールの pyplot サブモジュールを使います。また、日付データを適切に取り扱うために datetime モジュールも使います。

```
#%matplotlib inline #なくても良いようです。
import matplotlib.pyplot as pyplot
import datetime
```

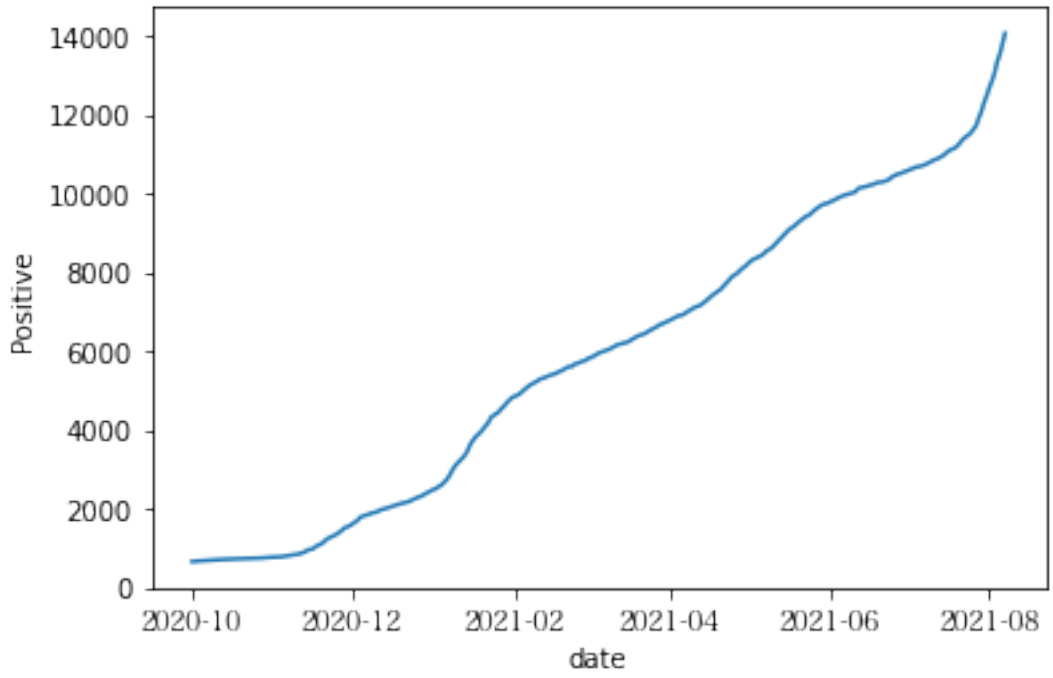
```
def plot_positive(pref):
    with sqlite3.connect("covid19.db") as db:
```

(次のページに続く)

(前のページからの続き)

```
        cur=db.cursor()
        data=cur.execute(
            """
            select year, month, date, testedPositive, peopleTested
            from  prefectures
            where  prefectureNameJ == \"{}\"
            and    (year,month) >= (2020,10)
            order by year,month,date;
            """.format(pref)
        ).fetchall()
        xdata=[datetime.date(y,m,d) for y,m,d,*v in data]
        ydata=[v for y,m,d,*v in data]
        pyplot.xlabel("date")
        pyplot.ylabel("Positive")
        pyplot.plot(xdata,[v[0] for v in ydata])
        pyplot.draw()
        locs,labels=pyplot.xticks()
        pyplot.xticks(locs[::2], labels[::2],font="Hiragino Mincho ProN
↪")
```

```
plot_positive("茨城県")
```

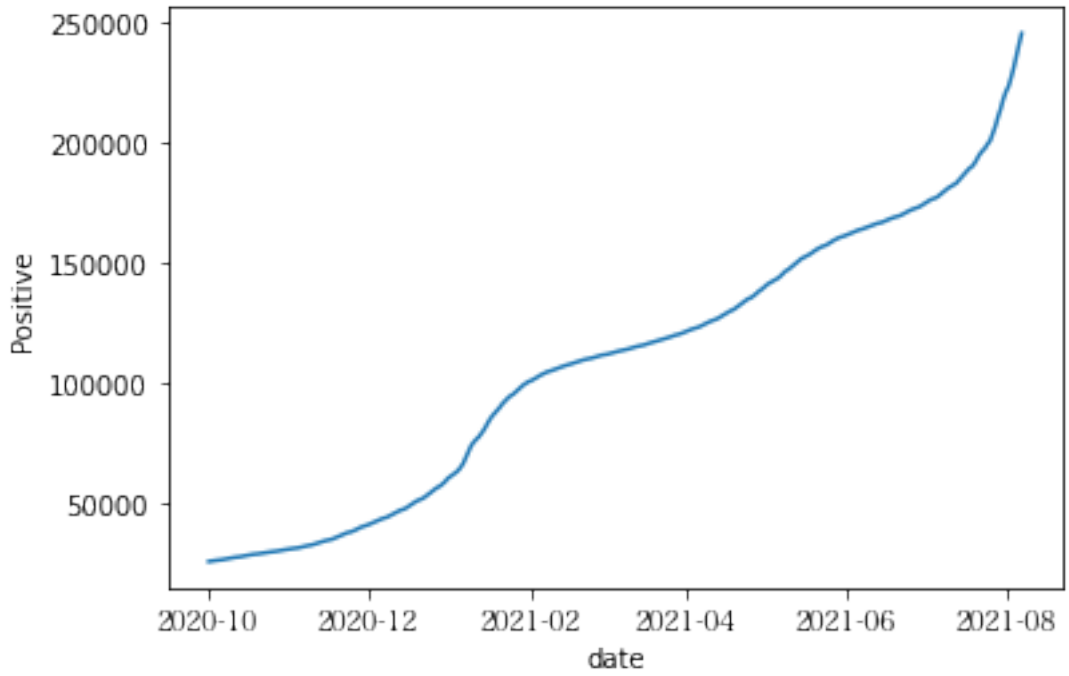



```
pyplot.clf()
plot_positive("東京都")
```

(次のページに続く)

(前のページからの続き)

```
#pyplot.show()
```



5.3.6 Dataframe を使って、同様のことができるか確認してみる。

Dataframe を使って同様のグラフを書かせてみた。loc () メソッドを使うと、SQL の select 相当のことが可能。StringIO を使うことで、中間の csv ファイルを生成せずに DataFrame に変換ができる。実は StringIO も使わないで、urlopen した object から pandas.read_cvs がデータを読み込める。

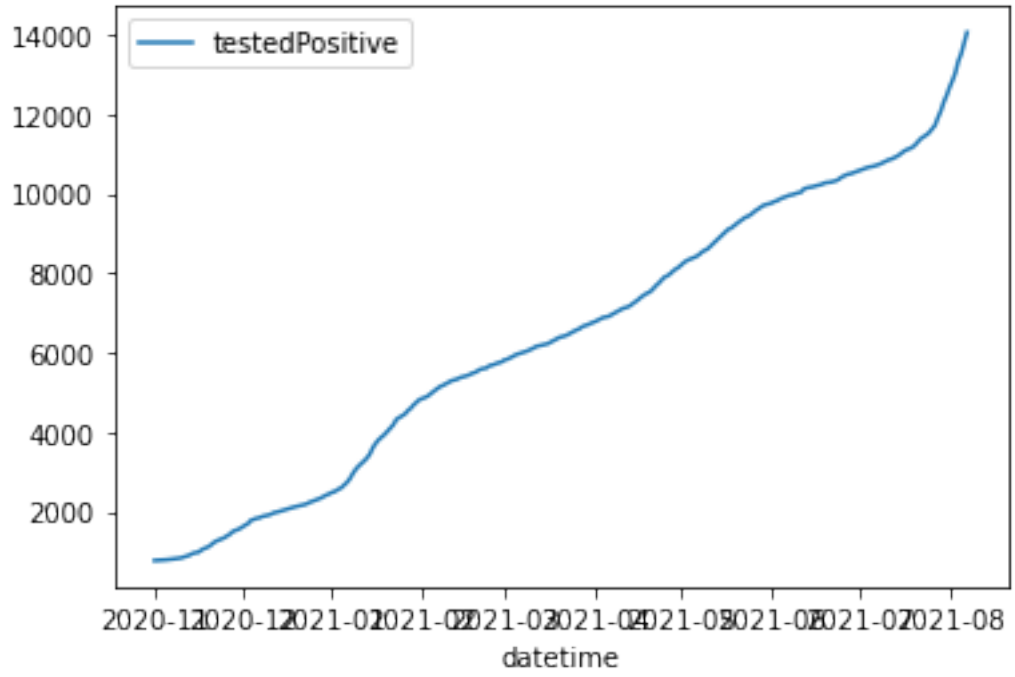
```
import io
from io import StringIO, BytesIO

def load_csv_to_df(dataurl):
    fn=dataurl.split("/")[-1]
    with urlopen(dataurl) as inf:
        print ("downloading :", fn)
        data=inf.read().decode('utf-8') # binary data を Unicode に変換
        sio=io.StringIO(data) # data を StringIO にセット。
    df=pandas.read_csv(sio) #
    return df

def plot_positive_df(pref):
    #load(dataurls[-1]) #csv を読んで
```

(次のページに続く)

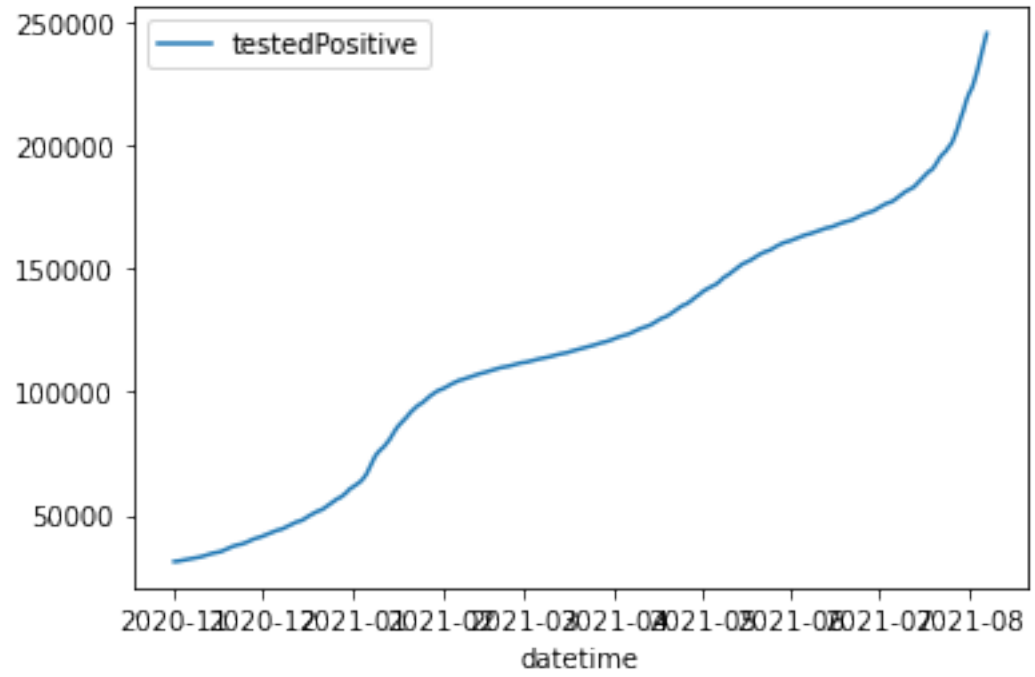
(前のページからの続き)

```
    #df=load_csv_to_df(dataurls[-1]) # pandas dataframe に変換
    # uri から直接 csv を読み込んで、dataframe を作成。
    df= pandas.read_csv(urlopen(dataurls[-1]))
    # .loc() メソッドを使って、データを選択。
    # df=df.loc[(df.prefectureNameJ == "{}".format(pref)) &
    #           ((df.year > 2020) | ((df.year == 2020) & (df.month >=␣
↪10))) , :]
    df=df.loc[df.apply(
                lambda x:
                   (x.prefectureNameJ == "{}".format(pref))
                   and ((x.year,x.month) > (2020,10)), axis=1) , :]
    #df=df[df.prefectureNameJ == "{}".format(pref)][(df.year > 2020)]
    # "datetime" column を追加。
    df['datetime']=df.apply(lambda x:datetime.date(x.year, x.month,x.
↪date), axis=1)
    # line plot でプロットしてみる。
    df.plot.line(x="datetime",y="testedPositive")
```

```
plot_positive_df("茨城県")
```




```
plot_positive_df("東京都")
```




```
df= pandas.read_csv(urlopen(dataurls[-1]))
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 26320 entries, 0 to 26319
Data columns (total 12 columns):
 #   Column                         Non-Null Count  Dtype
---  ------                         --------------  -----
 0   year                           26320 non-null  int64
 1   month                          26320 non-null  int64
 2   date                           26320 non-null  int64
 3   prefectureNameJ                26320 non-null  object
 4   prefectureNameE                26320 non-null  object
 5   testedPositive                 26320 non-null  float64
 6   peopleTested                   24226 non-null  float64
 7   hospitalized                   23767 non-null  float64
 8   serious                        21646 non-null  float64
 9   discharged                     23803 non-null  float64
 10  deaths                         23750 non-null  float64
 11  effectiveReproductionNumber  20513 non-null  float64
dtypes: float64(7), int64(3), object(2)
memory usage: 2.4+ MB
```

5.3.7 日毎検査数データ

日毎検査数データと日毎感染者数データから感染率を計算しグラフ化することを考えます。

```
dfcases= pandas.read_csv(urlopen(dataurls[5]))
dfcases
```

項目:民間検査会社（主に自費検査）項目：“民間検査会社（主に自費検査)”については 2021/7/12 から 2021/7/31 の間以外は全て NaN データが入っている。ちなみに東京オリンピック 2020 は 2021/7/13 開会式、同 8/8 に閉会式が催された。

pandas DataFrame の行の一部を取り出すには、index のスライスあるいは、index の長さ分の bool 値のリストを与える。DataFrame の列は列名を [] の指定することで要素を取り出せる。これを数値と比較することで, index 毎に審理中 w

```
(dfcases["民間検査会社（主に自費検査）"] > 0)
```

```
0      False
1      False
2      False
3      False
4      False
       ...
531    False
532    False
533    False
534    False
535    False
Name: 民間検査会社（主に自費検査）, Length: 536, dtype: bool
```

```
dfcases.loc[(dfcases["民間検査会社（主に自費検査）"] > 0 ),["日付","民間検査会社（主に自費検査）"]]
```

```
help(df.loc)
```

```
Help on _LocIndexer in module pandas.core.indexing object:

class _LocIndexer(_LocationIndexer)
 |  Access a group of rows and columns by label(s) or a boolean array.
 |
 |  .loc[] is primarily label based, but may also be used with a
 |  boolean array.
```

```
|
|  Allowed inputs are:
|
|  - A single label, e.g. 5 or 'a', (note that 5 is
|    interpreted as a label of the index, and never as an
|    integer position along the index).
|  - A list or array of labels, e.g. ['a', 'b', 'c'].
|  - A slice object with labels, e.g. 'a':'f'.
|
|    .. warning:: Note that contrary to usual python slices, both the
|        start and the stop are included
|
|  - A boolean array of the same length as the axis being sliced,
|    e.g. [True, False, True].
|  - An alignable boolean Series. The index of the key will be␣
↪aligned before
|    masking.
|  - An alignable Index. The Index of the returned selection will be␣
↪the input.
|  - A callable function with one argument (the calling Series or
|    DataFrame) and that returns valid output for indexing (one of␣
↪the above)
|
|  See more at Selection by Label.
|
|  Raises
|  ------
|  KeyError
|      If any items are not found.
|  IndexingError
|      If an indexed key is passed and its index is unalignable to␣
↪the frame index.
|
|  See Also
|  --------
|  DataFrame.at : Access a single value for a row/column label pair.
|  DataFrame.iloc : Access group of rows and columns by integer␣
↪position(s).
|  DataFrame.xs : Returns a cross-section (row(s) or column(s)) from␣
↪the
|      Series/DataFrame.
|  Series.loc : Access group of values using labels.
|
```

```
|  Examples
|  --------
|  Getting values
|
|  >>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
|  ...      index=['cobra', 'viper', 'sidewinder'],
|  ...      columns=['max_speed', 'shield'])
|  >>> df
|              max_speed  shield
|  cobra               1       2
|  viper               4       5
|  sidewinder          7       8
|
|  Single label. Note this returns the row as a Series.
|
|  >>> df.loc['viper']
|  max_speed    4
|  shield       5
|  Name: viper, dtype: int64
|
|  List of labels. Note using [[]] returns a DataFrame.
|
|  >>> df.loc[['viper', 'sidewinder']]
|              max_speed  shield
|  viper               4       5
|  sidewinder          7       8
|
|  Single label for row and column
|
|  >>> df.loc['cobra', 'shield']
|  2
|
|  Slice with labels for row and single label for column. As mentioned
|  above, note that both the start and stop of the slice are included.
|
|  >>> df.loc['cobra':'viper', 'max_speed']
|  cobra    1
|  viper    4
|  Name: max_speed, dtype: int64
|
|  Boolean list with the same length as the row axis
|
|  >>> df.loc[[False, False, True]]
```

```
|              max_speed  shield
|  sidewinder          7       8
|
|  Alignable boolean Series:
|
|  >>> df.loc[pd.Series([False, True, False],
|  ...         index=['viper', 'sidewinder', 'cobra'])]
|              max_speed  shield
|  sidewinder          7       8
|
|  Index (same behavior as df.reindex)
|
|  >>> df.loc[pd.Index(["cobra", "viper"], name="foo")]
|         max_speed  shield
|  foo
|  cobra          1       2
|  viper          4       5
|
|  Conditional that returns a boolean Series
|
|  >>> df.loc[df['shield'] > 6]
|              max_speed  shield
|  sidewinder          7       8
|
|  Conditional that returns a boolean Series with column labels␣
↪specified
|
|  >>> df.loc[df['shield'] > 6, ['max_speed']]
|              max_speed
|  sidewinder          7
|
|  Callable that returns a boolean Series
|
|  >>> df.loc[lambda df: df['shield'] == 8]
|              max_speed  shield
|  sidewinder          7       8
|
|  Setting values
|
|  Set value for all items matching the list of labels
|
|  >>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
|  >>> df
```

```
|              max_speed  shield
|  cobra               1       2
|  viper               4      50
|  sidewinder          7      50
|
|  Set value for an entire row
|
|  >>> df.loc['cobra'] = 10
|  >>> df
|              max_speed  shield
|  cobra              10      10
|  viper               4      50
|  sidewinder          7      50
|
|  Set value for an entire column
|
|  >>> df.loc[:, 'max_speed'] = 30
|  >>> df
|              max_speed  shield
|  cobra              30      10
|  viper              30      50
|  sidewinder         30      50
|
|  Set value for rows matching callable condition
|
|  >>> df.loc[df['shield'] > 35] = 0
|  >>> df
|              max_speed  shield
|  cobra              30      10
|  viper               0       0
|  sidewinder          0       0
|
|  Getting values on a DataFrame with an index that has integer labels
|
|  Another example using integers for the index
|
|  >>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
|  ...      index=[7, 8, 9], columns=['max_speed', 'shield'])
|  >>> df
|     max_speed  shield
|  7          1       2
|  8          4       5
|  9          7       8
```

```
|
|  Slice with integer labels for rows. As mentioned above, note that␣
↪both
|  the start and stop of the slice are included.
|
|  >>> df.loc[7:9]
|     max_speed  shield
|  7          1       2
|  8          4       5
|  9          7       8
|
|  Getting values with a MultiIndex
|
|  A number of examples using a DataFrame with a MultiIndex
|
|  >>> tuples = [
|  ...    ('cobra', 'mark i'), ('cobra', 'mark ii'),
|  ...    ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
|  ...    ('viper', 'mark ii'), ('viper', 'mark iii')
|  ... ]
|  >>> index = pd.MultiIndex.from_tuples(tuples)
|  >>> values = [[12, 2], [0, 4], [10, 20],
|  ...         [1, 4], [7, 1], [16, 36]]
|  >>> df = pd.DataFrame(values, columns=['max_speed', 'shield'],
index=index)
|  >>> df
|                         max_speed  shield
|  cobra      mark i            12       2
|             mark ii            0       4
|  sidewinder mark i            10      20
|             mark ii            1       4
|  viper      mark ii            7       1
|             mark iii          16      36
|
|  Single label. Note this returns a DataFrame with a single index.
|
|  >>> df.loc['cobra']
|           max_speed  shield
|  mark i          12       2
|  mark ii          0       4
|
|  Single index tuple. Note this returns a Series.
|
```

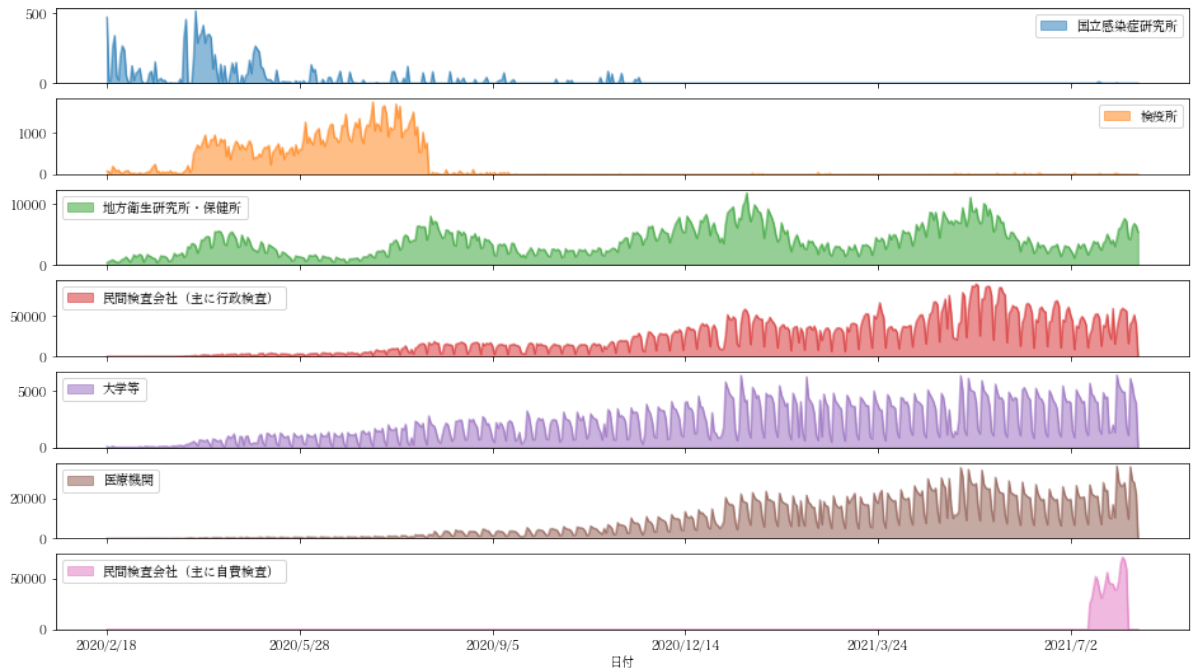
```
|  >>> df.loc[('cobra', 'mark ii')]
|  max_speed    0
|  shield       4
|  Name: (cobra, mark ii), dtype: int64
|
|  Single label for row and column. Similar to passing in a tuple,
this
|  returns a Series.
|
|  >>> df.loc['cobra', 'mark i']
|  max_speed    12
|  shield        2
|  Name: (cobra, mark i), dtype: int64
|
|  Single tuple. Note using [[]] returns a DataFrame.
|
|  >>> df.loc[[('cobra', 'mark ii')]]
|                 max_speed  shield
|  cobra mark ii          0       4
|
|  Single tuple for the index with a single label for the column
|
|  >>> df.loc[('cobra', 'mark i'), 'shield']
|  2
|
|  Slice from index tuple to single label
|
|  >>> df.loc[('cobra', 'mark i'):'viper']
|                       max_speed  shield
|  cobra      mark i           12       2
|             mark ii           0       4
|  sidewinder mark i           10      20
|             mark ii           1       4
|  viper      mark ii           7       1
|             mark iii         16      36
|
|  Slice from index tuple to index tuple
|
|  >>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
|                      max_speed  shield
|  cobra      mark i          12       2
|             mark ii          0       4
|  sidewinder mark i          10      20
```

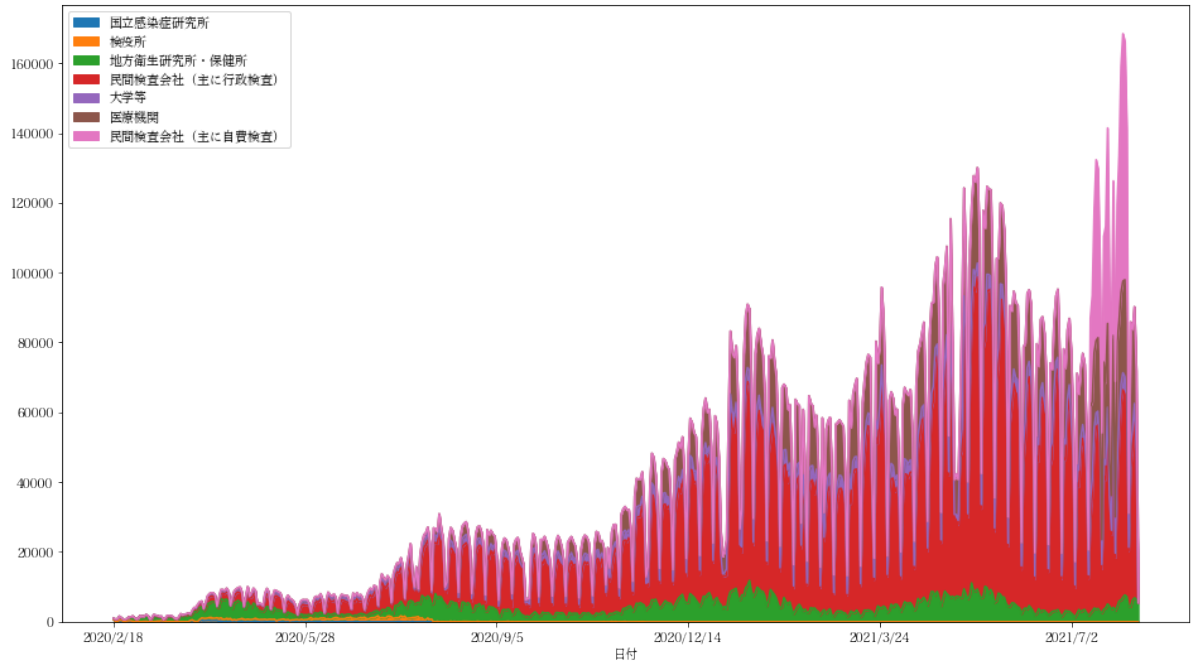
```
|             mark ii          1       4
|  viper      mark ii          7       1
|
|  Method resolution order:
|      _LocIndexer
|      _LocationIndexer
|      pandas._libs.indexing.NDFrameIndexerBase
|      builtins.object
|
|  Data and other attributes defined here:
|
|  __annotations__ = {'_takeable': <class 'bool'>}
|
|  ----------------------------------------------------------------------
|  Methods inherited from _LocationIndexer:
|
|  __call__(self, axis=None)
|      Call self as a function.
|
|  __getitem__(self, key)
|
|  __setitem__(self, key, value)
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from _LocationIndexer:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Data and other attributes inherited from _LocationIndexer:
|
|  axis = None
|
|  ----------------------------------------------------------------------
|  Methods inherited from pandas._libs.indexing.NDFrameIndexerBase:
|
|  __init__(self, /, *args, **kwargs)
|      Initialize self.  See help(type(self)) for accurate signature.
|
```

```
 |  __reduce__ = __reduce_cython__(...)
 |
 |  __setstate__ = __setstate_cython__(...)
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from pandas._libs.indexing.
↪NDFrameIndexerBase:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate␣
↪signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from pandas._libs.indexing.
↪NDFrameIndexerBase:
 |
 |  name
 |
 |  ndim
 |
 |  obj
```

```
#pyplot.style.use('ggplot')
import matplotlib
font = {'family' : "Hiragino Mincho ProN"}
matplotlib.rc('font', **font)
dfcases.plot("日付",kind="area",stacked=False,figsize=(16,9),
↪subplots=True)
dfcases.plot("日付",kind="area",stacked=True,figsize=(16,9))
```

```
<AxesSubplot:xlabel='日付'>
```






```
df.sum(axis=1)
```

```
0           2048.00
1           2047.00
2           2047.00
3           2047.00
4           2047.00
            ...
26315      91777.46
```

(次のページに続く)

(前のページからの続き)

```
26316     55309.24
26317     80013.63
26318    142069.24
26319     84154.86
Length: 26320, dtype: float64
```

各行毎の和を計算する。

各列毎の総和を求める。また、各行の和を求め、新い列 total として追加する。

```
grand_total = dfcases.sum(axis=0)
dftotal=dfcases.copy()
dftotal["total"]=dftotal.sum(axis=1)
```

```
grand_total
```

```
日付                    2020/2/182020/2/192020/2/202020/2/212020/2/222...
国立感染症研究所                                                      ␣
↪12192
検疫所                                                          108640
地方衛生研究所・保健所                                                ␣
↪2091278
民間検査会社（主に行政検査）                                          ␣
↪12265627.0
大学等                                                       1138452.0
医療機関                                                     4651157.0
民間検査会社（主に自費検査）                                          ␣
↪925516.0
dtype: object
```

```
dftotal
```

このデータフレームをプロットし、新しい列“total”が追加されていることを確認。

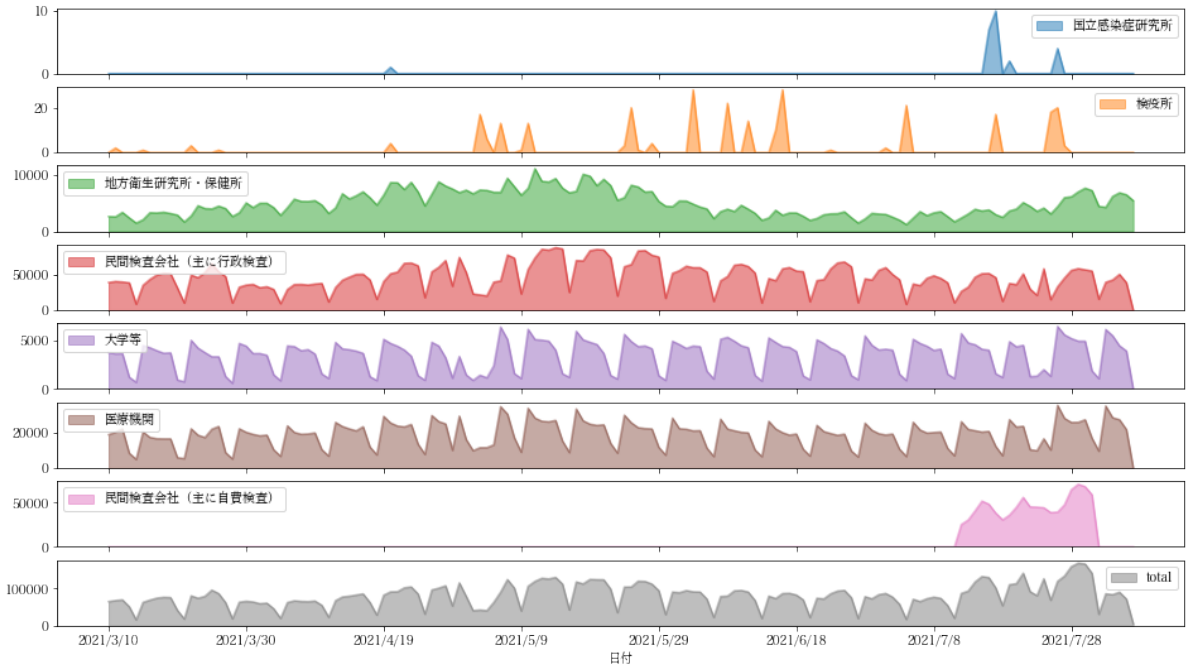
```
dftotal[-150:].plot("日付",kind="area",stacked=False,figsize=(16,9),
↪subplots=True)
```

```
array([<AxesSubplot:xlabel='日付'>, <AxesSubplot:xlabel='日付'>,
       <AxesSubplot:xlabel='日付'>, <AxesSubplot:xlabel='日付'>,
```

(次のページに続く)

(前のページからの続き)

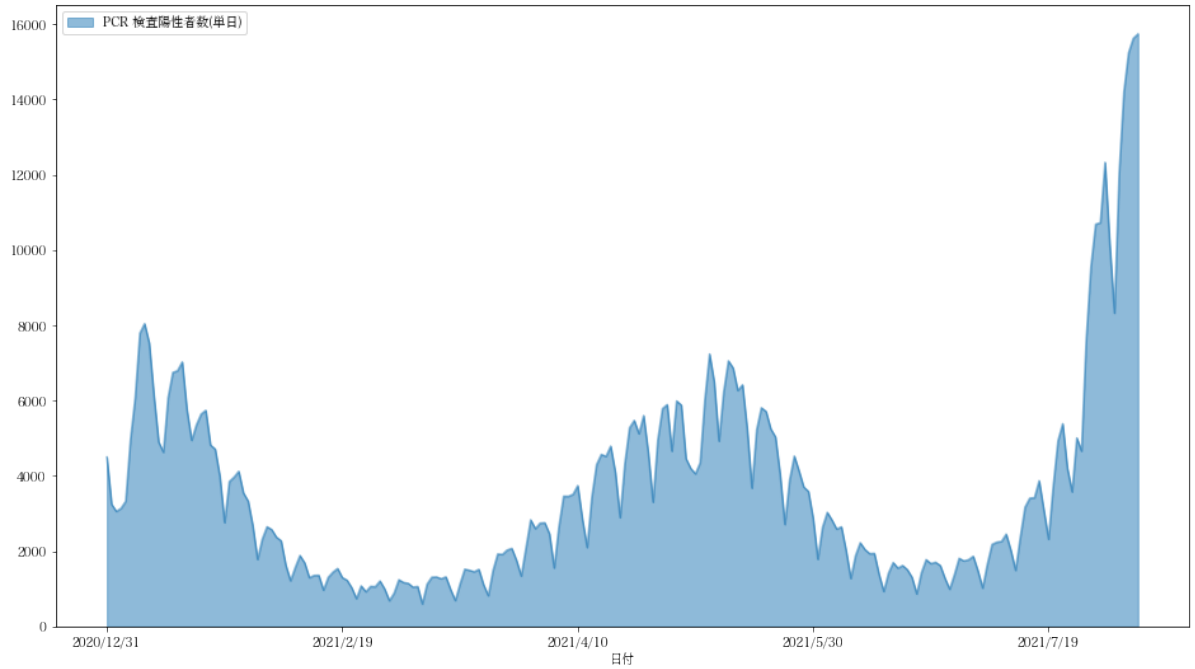
```
       <AxesSubplot:xlabel='日付'>, <AxesSubplot:xlabel='日付'>,
       <AxesSubplot:xlabel='日付'>, <AxesSubplot:xlabel='日付'>],␣
↪dtype=object)
```



この検査数データ (dftotal) と日毎陽性者のデータ (pcr_positive_daily.cs->dfpositive) を付き合わせてみます。

```
dfpositive= pandas.read_csv(urlopen(dataurls[0]))
dfpositive[-220:].plot("日付",kind="area",stacked=False,figsize=(16,9),
↪subplots=True)
```

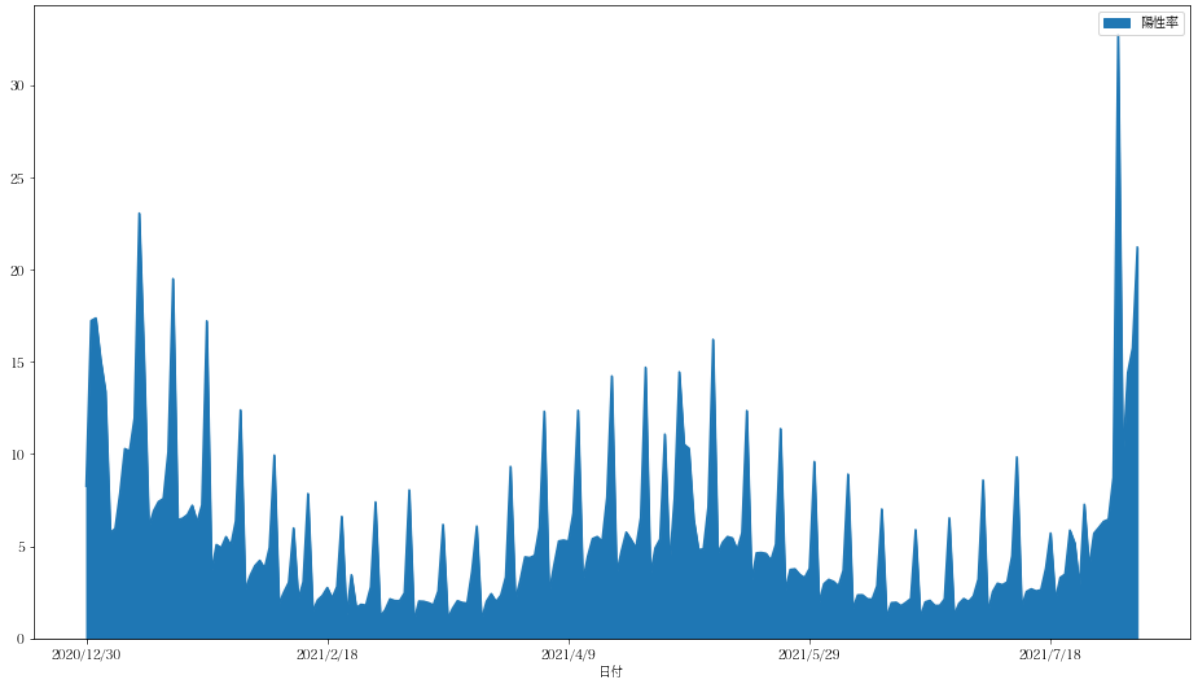
```
array([<AxesSubplot:xlabel='日付'>], dtype=object)
```



二つの Dataframe を JOIN するには DataFrame.merge 関数が便利です。ほぼ SQL 文の JOIN になっています。DataFrame.join ではジョインするキーとして　Index を使うことが必要となります。merge を使って、二つのデータセットの共通部分を取り出し、マージします。マージ後のデータフレームに新しい列“陽性率”を追加します。

```
# SQL の JOIN に対応するものは DataFrame.merge と考えた方がいい様です。
DataFrame.join は index での joinn を仮定している様です。
dfjoin=dftotal.merge(dfpositive,left_on="日付",right_on="日付", how=
↪"inner")
dfjoin["陽性率"]=dfjoin["PCR 検査陽性者数(単日)"]/dfjoin["total"]*100
dfjoin[-220:-1].plot("日付",["陽性率"],kind="area",figsize=(16,9))
#pyplot.savefig("陽性率.png")
```

```
<AxesSubplot:xlabel='日付'>
```



5.3.8 データフレーム:列へのアクセス

DataFrame の各列へのアクセスは幾つかの方法がある。

```
dftotal["日付"], dftotal.日付, dftotal.loc[:,"日付"],dftotal.iloc[:,0]
```

```
(0      2020/2/18
 1      2020/2/19
 2      2020/2/20
 3      2020/2/21
 4      2020/2/22
          ...
 531     2021/8/2
 532     2021/8/3
 533     2021/8/4
 534     2021/8/5
 535     2021/8/6
 Name: 日付, Length: 536, dtype: object,
 0      2020/2/18
 1      2020/2/19
 2      2020/2/20
 3      2020/2/21
 4      2020/2/22
          ...
```

(次のページに続く)

(前のページからの続き)

```
531     2021/8/2
532     2021/8/3
533     2021/8/4
534     2021/8/5
535     2021/8/6
Name: 日付, Length: 536, dtype: object,
0       2020/2/18
1       2020/2/19
2       2020/2/20
3       2020/2/21
4       2020/2/22
         ...
531     2021/8/2
532     2021/8/3
533     2021/8/4
534     2021/8/5
535     2021/8/6
Name: 日付, Length: 536, dtype: object,
0       2020/2/18
1       2020/2/19
2       2020/2/20
3       2020/2/21
4       2020/2/22
         ...
531     2021/8/2
532     2021/8/3
533     2021/8/4
534     2021/8/5
535     2021/8/6
Name: 日付, Length: 536, dtype: object)
```

複数の列を取り出すことも可能です。

```
dfcases[["日付", "医療機関"]]
```

5.3.9 選択

さらに条件にあう行だけを選択してみます。

```
dfcases[["日付", "医療機関"]][dfcases["医療機関"]>0]
```

5.4 日本の休日

Thunderbird などのカレンダーに日本の休日を表示したいと思った。Thunderbird の web サイト (https://www.thunderbird.net/ja/calendar/holidays) には各国の休日の情報が iCalender ファイル (.ics) の形式で収められている。しかし、日本の情報は 2007 年以降更新されていないが、2020-2021 は東京オリンピックの影響で、休日が臨時に変更されている。(その他の休日にも 2007 以降の変更がある。)

日本の休日については、内閣府がそのホームページ (https://www8.cao.go.jp/chosei/shukujitsu/gaiyou.html) で公式なアナウンスを出している。特に、“昭和 30 年（1955 年）から令和 4 年（2022 年）国民の祝日”を csv 形式のファイル (https://www8.cao.go.jp/chosei/shukujitsu/syukujitsu.csv) として公開している。

このプログラムの目的はこの csv 形式のファイルから Thunderbird などの Calendar プログラムで利用可能な iCal 形式のファイル (.ics) を生成することです。

5.4.1 利用するモジュール

このプログラムでは以下のモジュールを使います。`import` に失敗するようでしたら、`pip` などのツールを使って準備しましょう。

urllib: 標準の web アクセスのためのモジュール。csv ファイルをダウンロードするために使用。

io: データをあたかも File(stream) のように取り扱えるようにするモジュール。ダウンロードしたデータを、中間ファイルに保存しないで、メモリ上に置いたまま利用するため。

pandas: データ解析のためのモジュール。このなかの DataFrame を利用する。csv ファイルの日付データ変換などを効率よく (短いプログラムで）実現するために利用。

sqlite: ファイルベースの RDBMS sqlite を使うためのモジュール。データの検索などを簡単に行うために利用できる。本体のプログラムでは未利用なので、なくても良い。

icalendar: iCal データ取扱のためのモジュール。ics ファイル作成に使用する。

```
#!/usr/bin/env python
# coding: utf-8
"""
 This program generates ics file for the Japanese Holidays
 based on the information from Japanese goverment web page.
"""
import urllib.request
from urllib.request import urlopen
# macos では SSL 証明書を正しくつかえるように以下の文が必要
```

(次のページに続く)

(前のページからの続き)

```
import os, certifi
os.environ["SSL_CERT_FILE"]=certifi.where()

import io
from io import StringIO, BytesIO

import pandas

import sqlite3

import icalendar, pytz, datetime, uuid
from icalendar import vBoolean, vCalAddress, vDate, vDatetime,
↪vDuration,vFloat, vFrequency, vInt, vPeriod, vText, vTime, vUri,␣
↪vUTCOffset, vWeekday

JST = pytz.timezone("Asia/Tokyo")

# note: this csv uses shift-jis encoding.
JHuri="https://www8.cao.go.jp/chosei/shukujitsu/syukujitsu.csv"
# a list of codecs supported by python can be found in https://docs.
↪python.org/ja/3/library/codecs.html
```

5.4.2 データのダウンロード

データを web アドレス (cvsuri) からダウンロードして、

1. 手元のファイルとして保存したり (load_and_save_cvs)、
2. pands DataFrame に変換して (load_as_dataframe) 別のプログラムで利用したり、
3. sqlite3 の on memory データベスに変換 (load_as_ db) する

関数等をここで定義しています。

```
def load_and_save_cvs(cvsuri):
    fn=cvsuri.split("/")[-1]
    with urlopen(cvsuri) as inf , open(fn,"wb") as outf:
        print ("downloading :", fn)
        data=inf.read()
        outf.write(data)
```

(次のページに続く)

(前のページからの続き)

```
def load_as_dataframe(cvsuri):
    fn=cvsuri.split("/")[-1]
    with urlopen(cvsuri) as inf:
        print ("downloading :", fn)
        data=inf.read().decode("shift_jis") #file encoding in this csv␣
↪file is shift JIS.
        sio=io.StringIO(data) # store data in to StringIO object.
        # interprete string in '国民の祝日・休日月日'(the first)  column␣
↪as datetime.
        df=pandas.read_csv(sio, # use StringIO object as if it is a␣
↪open file object.
                           parse_dates=['国民の祝日・休日月日'], #␣
↪column 0 shoud be read as date
                           infer_datetime_format=True
                          )
        #  df=pandas.read_csv(sio, parse_dates=[0], infer_datetime_
↪format=True)
    print(df.info())
    return df

def load_as_db(cvsuri):
    """
    download csv file then convert to sqlite3 database on memory.
    """
    df=load_as_dataframe(cvsuri) # create dataframe
    db=sqlite3.connect(":memory:") # create sqlite3 database on memory.
    df.to_sql('holidays', db) # write all data to database.

    return db
```

5.4.3 Calendar オブジェクト

iCal のファイルを作成する準備として、icalender.Calendar クラスを継承した Calendar Class を定義します。このクラスには休日 (日付と休日名) を登録するための `add_holiday` メソッドを追加しておきます。　Calendear ファイルは icanldear.Calendar クラスが持つ、`to_ical` を使って書き出します。

```
class Calendar(icalendar.Calendar):
    """
```

(次のページに続く)

(前のページからの続き)

```
    """
    # notify tis event 15 min before event
    alarm_setting=icalendar.Alarm().from_ical(
        "BEGIN:VALARM\n"
        "ACTION:DISPLAY\n"
        "TRIGGER;VALUE=DURATION:-PT15M\n"
        "DESCRIPTION:Default Mozilla Description\n"
        "END:VALARM"
    )
    def __init__(self):
        """
        """
        super(icalendar.Calendar,self).__init__()
        self.add(u"prodid","-//Japanese Holidays/based on Python.
↪iCalendar//")
        self.add(u"Version",2.0)
        self.add(u"Category","Japanese Public Holidays")
        self.add(u"Summary","Official Japanese Public Holidays ")
        self.add_component(self.alarm_setting)
        self.add_component(icalendar.Timezone.from_ical(
            "BEGIN:VTIMEZONE\n"
            "TZID:/mozilla.org/20070129_1/Asia/Tokyo\n"
            "X-LIC-LOCATION:Asia/Tokyo\n"
            "BEGIN:STANDARD\n"
            "TZOFFSETFROM:+0900\n"
            "TZOFFSETTO:+0900\n"
            "TZNAME:JST\n"
            "DTSTART:19700101T000000\n"
            "END:STANDARD\n"
            "END:VTIMEZONE\n")
        )

    def add_holiday(self, date, name):
        ev=icalendar.Event()
        ev['uid']=uuid.uuid4()
        ev["categories"]=vText("Japanese Public Holidays")
        ev.add(
            'dtstart',
            vDate(
                datetime.datetime.fromtimestamp(
```

(次のページに続く)

(前のページからの続き)

```
                    date.timestamp(),
                tz=JST)
            ),
            encode=False
        )
        ev.add(
            'dtend',
            vDate(
                datetime.datetime.fromtimestamp(
                (date + datetime.timedelta(days=1)).timestamp(),
                tz=JST)
            ),
            encode=False
        )
        ev.add(
            'summary',
            name
        )
        ev.add(
            'description',
            vText(name, 'utf-8')
        )
        self.add_component(ev)
```

5.4.4 プログラムの本体

上で用意した関数やクラスを使って web から取り込んだデータを使って、ics ファイルを作成します。

```
def main():
    # web 上のデータから DataFrame を作成。
    df=load_as_dataframe(JHuri)

    cal=Calendar() # Calendar オブジェクトを作成し、DataFrame の休日データを登録
    for id, date, name in df.itertuples():
            cal.add_holiday(date, name)
    #.ics ファイルを作成し、データを iCalendar 形式で書き出します。
    with open(
```

(次のページに続く)

(前のページからの続き)

```
            "JapaneseHolidays.ics",
            mode="w",
            encoding="utf-8" # for py2app. you need explicitly specify␣
↪encoding.
    ) as f:
        s=cal.to_ical() # icalendar 形式に変換。
        s=s.decode("utf-8", "replace") # unicode に変換
        f.write(s) #ファイルに書き出す。
```

5.4.5 プログラムの実行

完成したプログラムを単独で利用することを考えて、いつものおまじないをいれておきます。

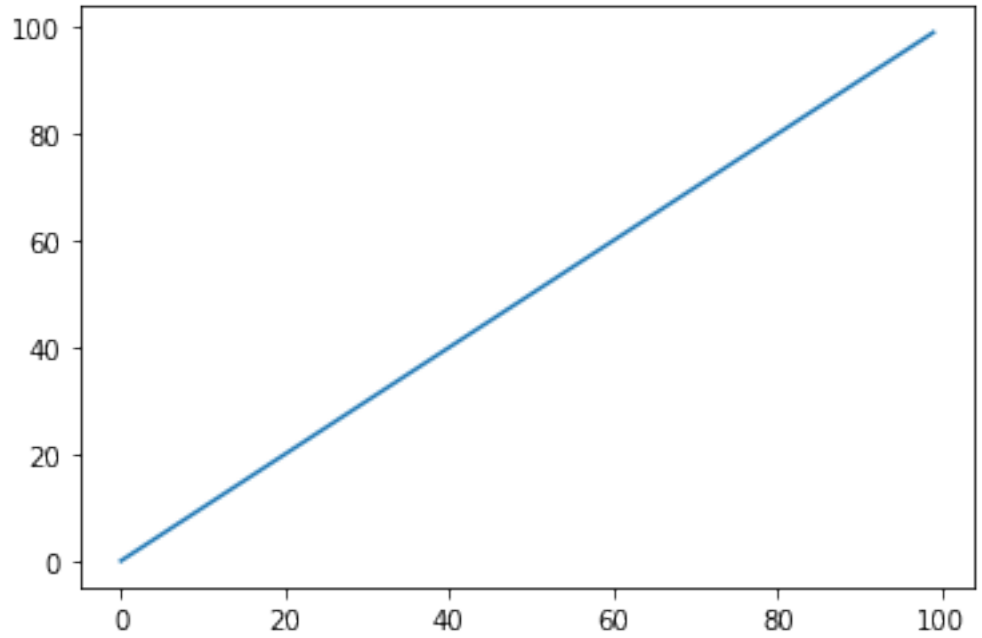
```
if __name__ == "__main__":
    main()
    print("Done!")
```

```
downloading : syukujitsu.csv
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 975 entries, 0 to 974
Data columns (total 2 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   国民の祝日・休日月日  975 non-null    datetime64[ns]
 1   国民の祝日・休日名称  975 non-null    object
dtypes: datetime64[ns](1), object(1)
memory usage: 15.4+ KB
None
Done!
```

5.5 jupyter-execute の見本

```
name = 'world'
print('hello ' + name + '!')
import matplotlib.pyplot as pyplot
pyplot.plot(range(100))
pyplot.draw()
```

```
hello world!
```




```
this code is invisible
```

```
print('this output is invisible')
```

```
this output is above the code
```

```
print('this output is above the code')
```

```
print('A')
print('B')
print('C')
```

```
A
B
C
```

```
print('A')
print('B')
print('C')
```

```
A
B
C
```

```
d = {
'a': 1,
'b': 2,
'c': 3,
'd': 4,
'e': 5,
}
```

```
1 / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-9-bc757c3fda29> in <module>
----> 1 1 / 0

ZeroDivisionError: division by zero
```

```
a = {'hello': 'world!'}
a['jello']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-10-c54633378b1e> in <module>
      1 a = {'hello': 'world!'}
----> 2 a['jello']

KeyError: 'jello'
```

```
import sys

print("hello, world!", file=sys.stderr)
```

```
hello, world!
```

```
CountDigits
```

第6章　Pythonにポインターはないの？

C/C++の学習の難関のひとつがポインターの扱いだと言われています。C/C++のポインターは習熟すれば非常に強力な武器とな理ます。その一方で使い方を誤れば、システムを破壊することにもなりかねない力を持つ、”最後の武器"とも言えるでしょう。

そう思って改めて Python を見ると、python のデータ型の中にはポインター型は有りませんし、ポインターを操作するためのオペレーターや関数も存在しません。同じプログラム言語なのに？と思われませんか？

実は、Python では (そして Java でも)、変数の意味が C/C++とは異なっています。あえて乱暴な言い方をすれば、"Python(Java) では全ての変数はポインター”なのです。

C/C++では、変数は次の様に宣言されます。

```
long l;
double f;
long *pl=&l;
double *pf = &f;
```

この例では、 l, f が変数名で、 pl, pf がそれらの変数へのポインタを保持するポインタ変数ということになります。変数の値を変更すると、変数名に割り当てられたメモリ領域にストアされた値が変更されていることが分かります。

```
l=1;
printf("l=%ld, pl=0x%p, *pl=%ld \n", l, pl, *pl);
l+=1;
printf("l=%ld, pl=0x%p, *pl=%ld \n", l, pl, *pl);
```

```
% clang pointer.c
% ./a.out
l=1, pl=0x0x304edb708, *pl=1
l=2, pl=0x0x304edb708, *pl=2
```

一方 python で同様のプログラムを実行して見ると、

```
1 a=1
2 b=2
```

(次のページに続く)

(前のページからの続き)

```
print("a= {} id(1)={} id(a)={}".format(a , id(1),  id(a)))
print("b= {} id(2)={} id(b)={}".format(b , id(2),  id(b)))
a += 1
print("a= {} id(1)={} id(a)={}".format(a , id(1),  id(a)))
print("a= {} id(2)={} id(a)={}".format(a , id(2),  id(a)))
print("b= {} id(2)={} id(b)={}".format(b , id(2),  id(b)))
```

```
a= 1 id(1)=140498178337072 id(a)=140498178337072
b= 2 id(2)=140498178337104 id(b)=140498178337104
a= 2 id(1)=140498178337072 id(a)=140498178337104
a= 2 id(2)=140498178337104 id(a)=140498178337104
b= 2 id(2)=140498178337104 id(b)=140498178337104
```

というように、変数 a の値を変更すると、その id (メモリアドレスに相当）が変わってしまうことが分かります。またその値は、変数 a の値となっているオブジェクトの id と同じとなっています。このように、python では変数は、メモリ領域につけられた名前ではなく、その値のオブジェクトを示すポインタの様な物と思った方が間違いがありません。

```
var('x y')
y=(1+x)**20
show(y.expand())
```

第7章　章題名のアイデア

https://www.data.jma.go.jp/obd/stats/data/mdrr/tem_rct/alltable/mxtemsadext00_rct.csv

7.1 小さくまとめて、何度も使う (def)/分割して統治せよ

関数の定義

7.2 繰り返しを任せよう。

制御構造

7.3 見た目が全て？

文字型データと format

7.4 変わるもの変わらないもの

mutable type と immutable type

7.5 GUI

Tkinter

7.6 開いて、読み書き、最後は閉じる

7.6.1 ファイルの入出力

File open/read/write/close

7.6.2 ネットワーク通信

socket

7.7 EPICS CA

7.8 ハードと通信

7.8.1 socket 通信

7.8.2 Serial 通信

7.8.3 VXI11

7.9 (matplotlib.pyplot)

7.10 (matplotlib.splot)

7.11 (matplotlib.animation)

7.12 (read excel file:xlrd, openpyxl)

7.13 (write excel file:xlwt, openpyxl)

7.14 (Error 処理： try-except-else-finally)

例外処理

7.15 （Web からデータを入手する。:Beautiful Soup)

httplib

7.16 （数式処理： sympy, SageMath)

7.17 (jupyter lab)