



OpenAI GPT-4 ChatGPT LangChain 人口知能プログラミング実践入門

PDF版ダウンロードリンク

<https://www.borndigital.co.jp/book/30237.html>

Python3.10

Llamaindex0.6.12

LangChain0.0.181

Python開発環境の準備

Python開発環境の概要

無料で利用できるクラウドサービス「Google Colab」の利用法

初心者向けPythonコーディングの紹介

Pythonの概要

Pythonとは

プログラミング言語Python（パイソン）は、高水準で解釈型の汎用プログラミング言語です。Guido van Rossum氏によって作成され、1991年に最初にリリースされました。Pythonは、シンプルさ、可読性、汎用性が特徴であり、ウェブ開発、データ分析、人工知能、科学技術計算など、さまざまな用途に広く使われています。

Pythonの主な特徴:

可読性と表現力：

Pythonはコードの可読性を重視し、英語に似た構文を使用しています。これにより、開発者がコードを書くことや理解することが容易になります。

解釈型言語：

Pythonのコードは実行時にPythonインタプリタによって1行ずつ実行されます。そのため、コードを実行する前にコンパイルする必要がなく、開発プロセスがより迅速で対話的になります。

高水準言語：

Pythonは多くの低レベルの詳細を抽象化し、開発者に高い抽象化レベルを提供します。これにより、開発が迅速に進み、コードのメンテナンスが容易になります。

多パラダイム：

Pythonは手続き型、オブジェクト指向型、関数型のプログラミングスタイルなど、複数のプログラミングパラダイムをサポートしています。開発者は自分のニーズに合ったスタイルを選択できます。

多機能な標準ライブラリ：

Pythonには標準ライブラリが豊富に含まれており、ファイルI/O、ネットワーキング、正規表現、データ処理などのさまざまなタスクをサポートしています。これにより、一般的なタスクのためにコードをゼロから書く必要がありません。

動的型付け：

Pythonは動的型付け言語であり、変数の型は実行時に決定されます。変数のデータ型を明示的に宣言する必要がないため、コードの記述が柔軟になります。

クロスプラットフォーム：

PythonのコードはWindows、macOS、Linuxなど、さまざまなプラットフォームで修正することなく実行できます。

大規模なコミュニティとエコシステム：

Pythonには大規模で活発な開発者と愛好家のコミュニティがあり、多数のライブラリ、フレームワーク、ツールを含む豊富なエコシステムがあります。このエコシステムは多岐にわたるアプリケーションとユースケースをサポートしています。

Pythonの汎用性と使いやすさにより、初心者から経験豊富な開発者まで、広く利用されています。ウェブ開発、データ分析、機械学習、人工知能、科学技術計算、自動化など、さまざまな分野で活用されています。

Python3のAPIバージョンリファレンスサイト：

<https://docs.python.org/3.10/>

Python 3.13 (in development) Python 3.12 (pre-release) Python 3.11 (stable) Python 3.10 (security-fixes) Python 3.9 (security-fixes) Python 3.8 (security-fixes)

深層学習ライブラリを利用するプログラミング言語：

Pythonプログラミング言語は機械学習向けのライブラリも多く提供されるため、多くのプログラマに支持され、AI（人工知能）開発・深層学習開発のプログラミング言語のスタンダードとなっています。

PytorchやTensorFlowといった深層学習ライブラリの多くも「Python」で利用できます。LlamaIndexライブラリ、LangChainライブラリやOpenAI APIサービスはPytorchやTensorFlowを内部的に使っています。

Pytorch:

Facebookが開発した深層学習フレームワークです。動的な計算グラフ、Pythonのネイティブサポート、CUDA対応などの便利な機能が揃っています。研究者の多くに利用されています。

<https://pytorch.org/>

TensorFlow:

Googleが開発した深層学習フレームワークです。高水準APIであるKerasを利用することで、モデルの作成が非常に簡単になり、TensorFlow2.0からは、動的な計算グラフにも対応するようになりました。

<https://www.tensorflow.org/>

Google Colab (Google Colaboratory)

Googleが無料で提供しているPythonの開発環境プラットフォームである

無料で高性能なGPUが使えるので、深層学習モデルの学習に最適である

Colabの用途例:

- TensorFlow の導入
- ニューラル ネットワークの開発とトレーニング
- TPU を利用した実験
- AI リサーチの促進
- チュートリアルの作成

Google Colabの始め方

1. Google Chromeブラウザを開く

2. Googleアカウント登録

Google ドライブ <https://www.google.co.jp/drive/>

3. Googleアカウントログイン



図3.1

出所: ITmediaより. 公式Webサイト | Google Drive ホーム画面

4. 左上の「新規」ボタンをクリックし、「その他→アプリを追加」を選択
5. Google ドライブと Google Colab※のアプリを選択する

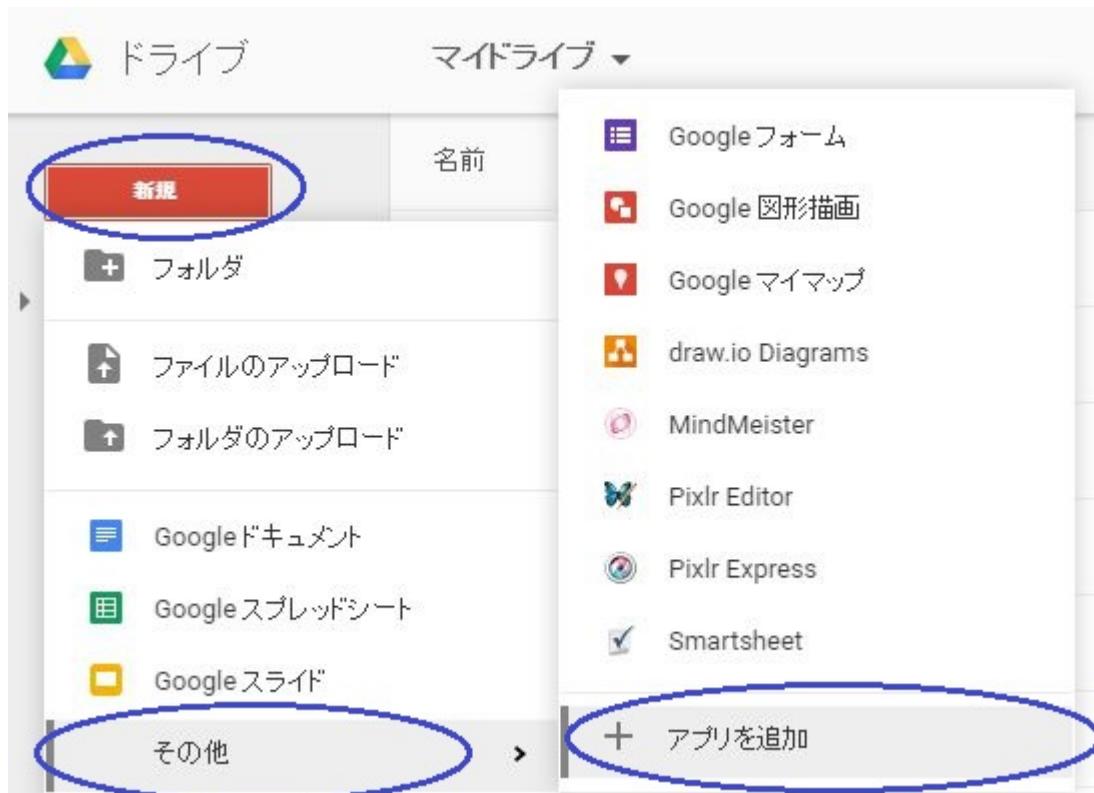


図3.2

出所: yukibowz.netより. 公式Webサイト | Google Driveの新規からその他、アプリの追加を選択するとアプリの検索画面

※Google Colabの場合、アプリにアイコンが表示されていないため、検索エリアに Colaboratoryと入力してアプリを選択し、インストールをクリック

6. 左上の「新規」ボタンをクリックし、「その他→Google Colaboratory」を選択

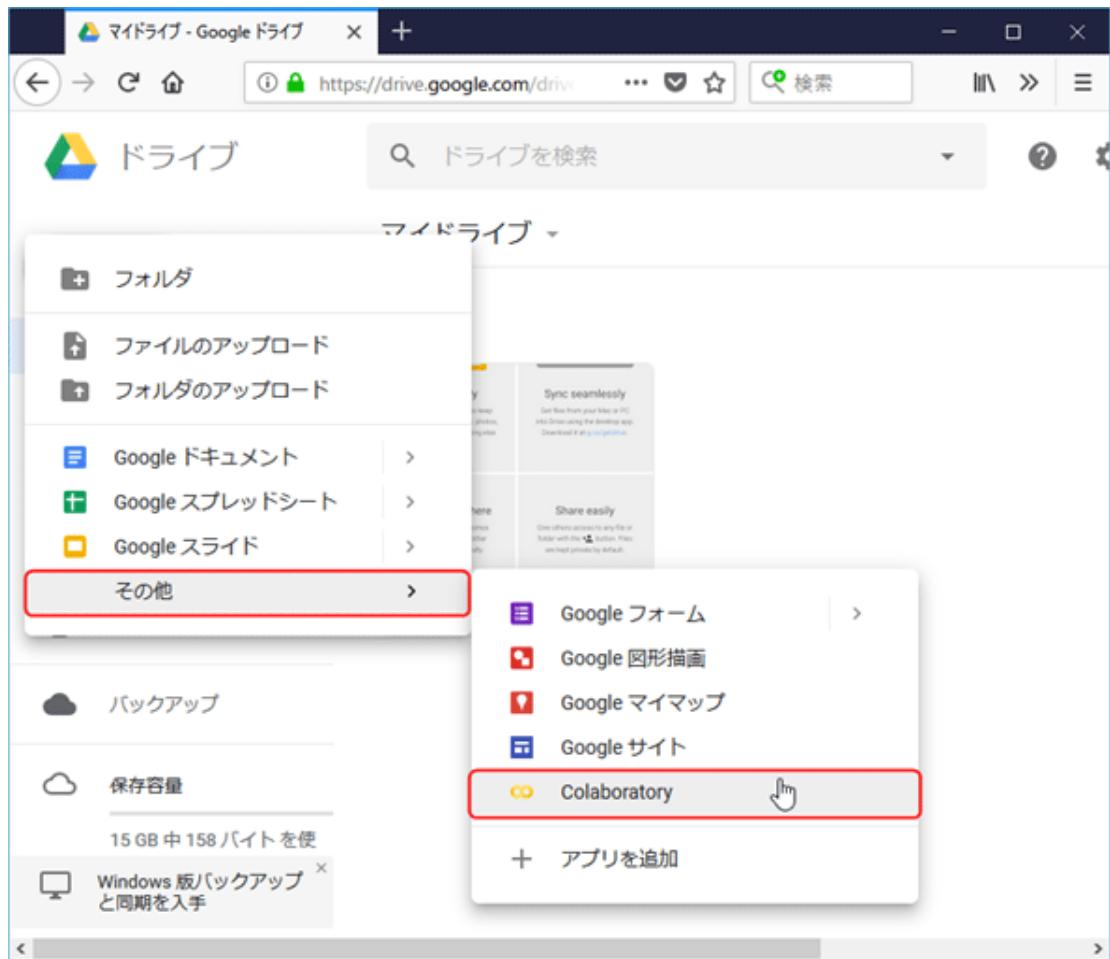


図3.3

出所: ITmediaはアイティメディア株式会社より、公式Webサイト |Colaboratoryノートブックの新規作成



図3.4

出所: ITmediaはアイティメディア株式会社より. 公式Webサイト |Colaboratoryアプリの追加画面

7. Google Colabの新規ノートブックが開く

ブラウザにGoogleアカウント（個人でもECCSでもどちらでもよい）でログインした後に、以下のURLを開いてください。

<https://colab.research.google.com/>

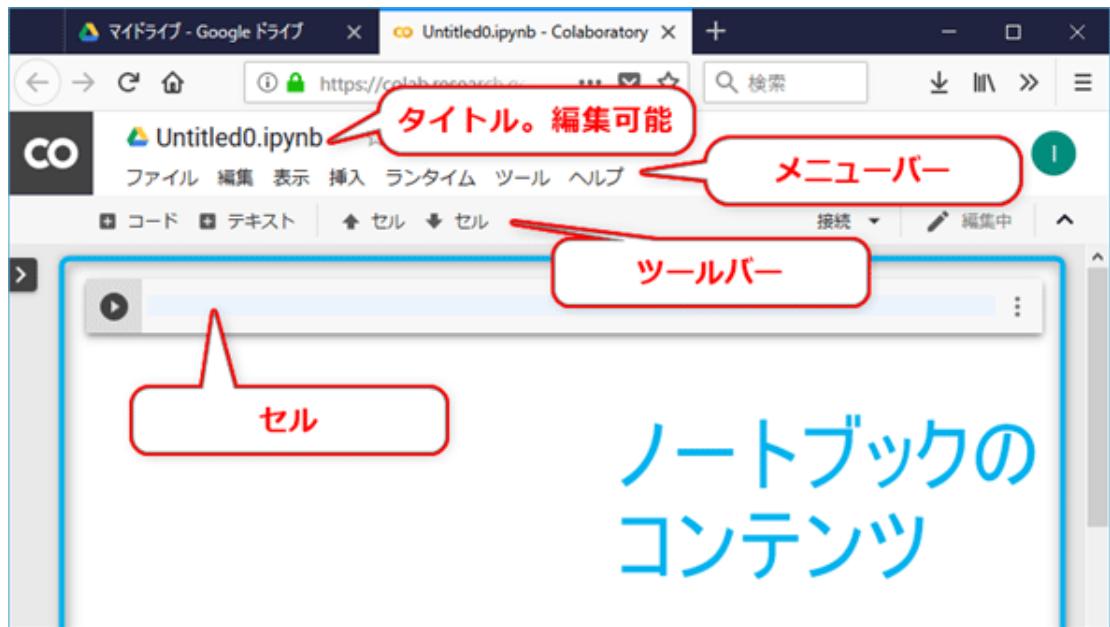


図3.5

出所: ITmediaはアイティメディア株式会社より、公式Webサイト |Google Colabアプリで開かれたColaboratoryノートブックファイル

セルの上にあるツールバーの +コード をクリックすると新しいセルが追加される

Google Colabの利用について

プラットフォームのファイルをノートブックと呼ばれ、ファイル形式は .jpynb で管理している

このファイルはGoogle ドライブに自動的に保存される

実行されたプログラミングスクリプト（Python文法）がクラウド上の仮想サーバーに保管される

Google Colabの利用制限について

無料版と有料版（Pro/Pro+/Pay As You Go）がある、有料の場合はメモリー量が多くなり、制限ルールも緩和される

以下は無料版のときの利用仕様となっている

項目	制限内容
RAM	12GBまで
ディスク	CPU/TPUは最大107GB、GPUは最大68GBまで
90分ルール	何も操作せずに90分経つとリセット、このため必要なときに設定し、利用しないときは設定を変更した方がいい
12時間ルール	インスタンス（ノートブック）が起動してから12時間経つとリセット

項目	制限内容
GPUの使用制限	GPUを使い過ぎるとリセット（上限未公開）
90分ルールの対策:	
1時間毎にノートブックをアクセル（開きなおす）することで回避できる	
以下のコードは1時間毎にリマインドするよう指示できるプログラムである	

```
import time
import datetime
import webbrowser

for i in range(12):
    browse = webbrowser.get("chrome")
    browse.open("<任意のノートブックのURL>")
    print(i, datetime.datetime.today())
    time.sleep(60*60)
```

注意:このコードの実行はGPU設定でないと実行できない

12時間ルールの対策:

以下のコードは12時間毎にリマインドするよう指示できるプログラムである

```
!cat/proc/uptime | awk '{printf("残り時間:%.2f", 12-$1/60/60)}'
```

これらの設定不備などによりリセットされてもデータの永続化をしていれば問題ない

次のGoogleドライブ利用についてデータの永続化をご覧ください

GPU(Graphic processing unit)/TPU(Tensor processing unit)の利用について

GPUの機能:

Google Colabの無料版で利用できるGPU名はTesla T4 (15GB) である、LlamaIndexやLangChainをローカルでの深層学習モデルの推論やベクトルデータベースの検索などにはGPUが必要である

TPUの機能:

Googleが開発した機械学習専用のハードウェアアクセラレータであり、機械学習の高速化を目的として設計され、TensorFlow関係のコードのパッケージ利用にはTPUが必要である

機能設定:

ツールバーメニュー → 編集 → ノートブックの設定 → GPU/TPU/Noneどれかを選択

(一般的な利用はNoneで十分、人工知能学習の利用についてはGPUまたはTPUを選択)

GPU/TPU無料版の場合使いすぎるとリセットされる（上限非公開）

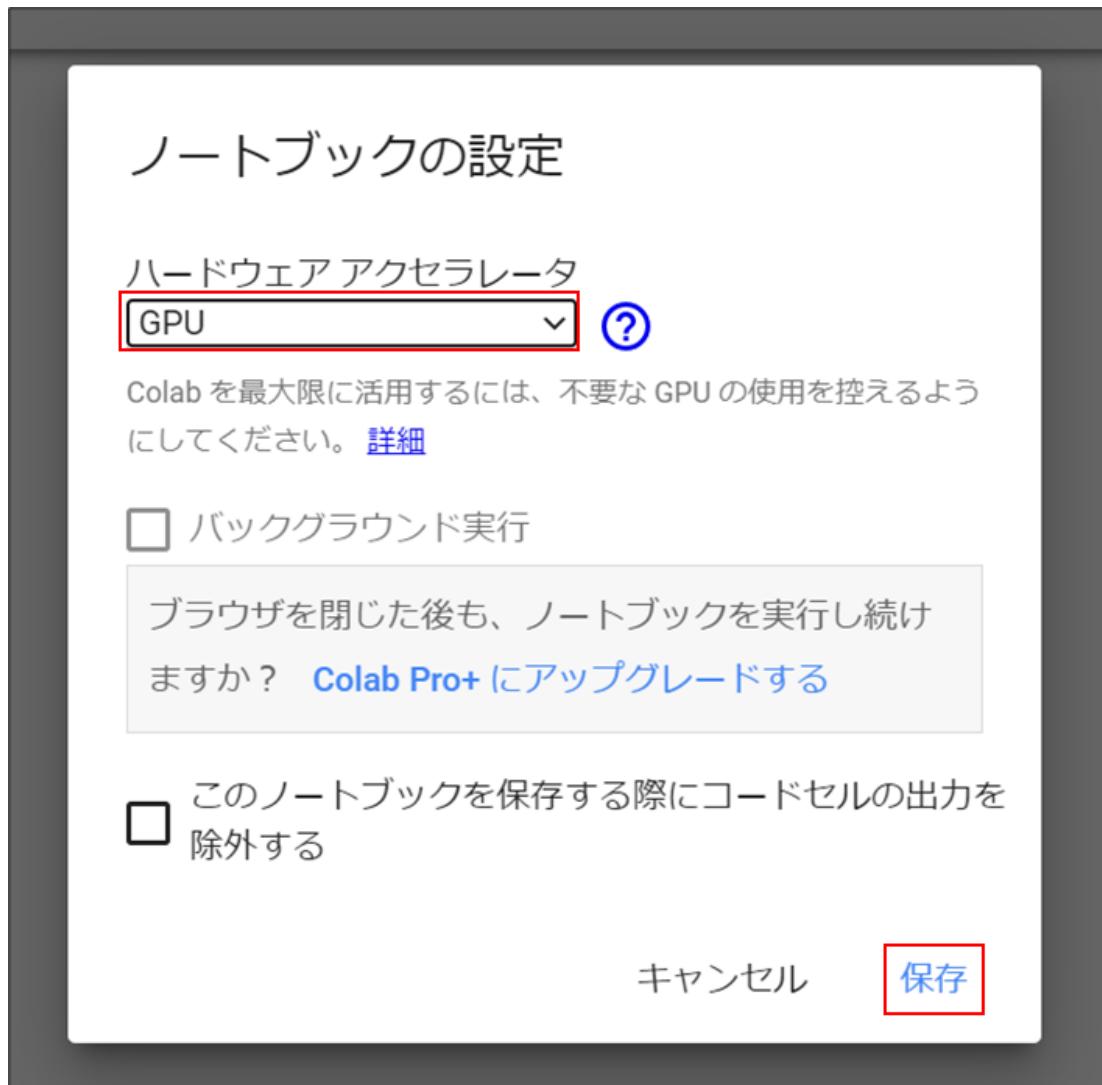


図3.6

出所: Skill Up AIより. 公式Webサイト | Colabの環境をGPUに設定するため、「ハードウェアアクセラレータ」を「GPU」に変更して「保存」をする画面

Google Driveの利用について

Google ドライブに作業フォルダを作成することで、データを永続化できる、リセットされても再開できる

フォルダ作成の流れ:

1. ドライブにフォルダを作る前にコードのセルに以下のコードを事前に実行し、接続を許可する必要がある

```
from google.colab import drive  
drive.mount("/content/drive")
```

2. 実行が成功すると Mounted at /content/drive と表示される

3. ドライブに"work"というフォルダを作るときにコードのセルに以下のコードを事前に実行する必要がある

```
import os  
os.makedirs("/content/drive/My Drive/work", exist_ok=True)  
%cd "/content/drive/My Drive/work"
```

4. 実行が成功すると /content/drive/My Drive/work と表示される

```
In [ ]: #1  
from google.colab import drive  
drive.mount("/content/drive")  
  
Mounted at /content/drive  
  
In [ ]: #3  
import os  
os.makedirs("/content/drive/My Drive/work", exist_ok=True)  
%cd "/content/drive/My Drive/work"  
  
/content/drive/My Drive/work
```

Google Colabの画面構成

コードセルの書き込みや実行機能ボタン

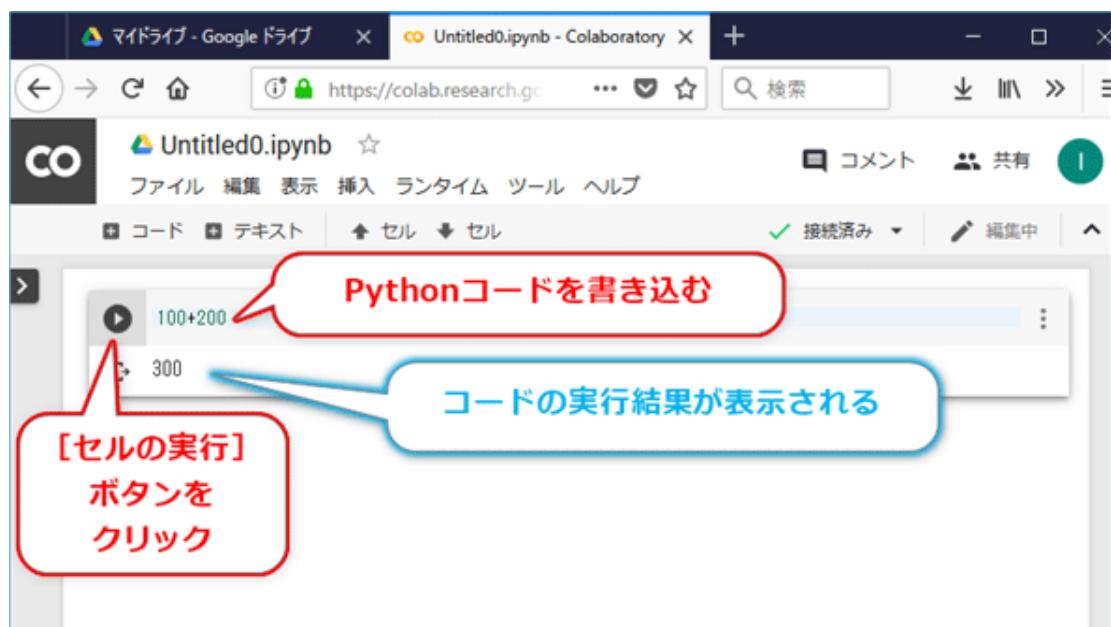


図3.7

出所: ITmediaより. 公式Webサイト |コードセルの操作方法

△をクリックするとコードが実行される

コードセルの下に結果が表示される

ツールバーの「+ コード」はコードセルの新規追加ボタン

ツールバーの「+ テキスト」はテキストセルの新規追加ボタン

テキストセル追加ボタン



図3.8

出所: ITmediaより. 公式Webサイト |テキストセルの挿入方法

コードセルの挿入ボタン

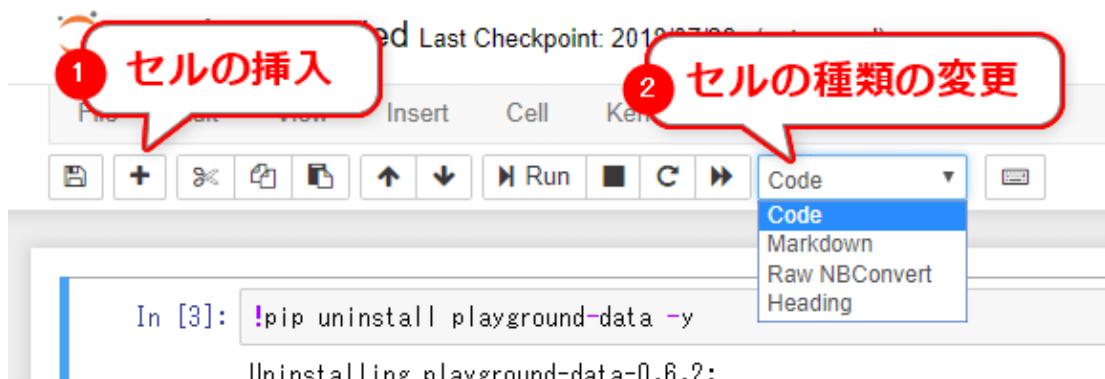


図3.9

出所: ITmediaより. 公式Webサイト |コードセルの挿入方法

テキスト記入とレビュー表示画面

*テキスト作成中は両面表示状態になっています。

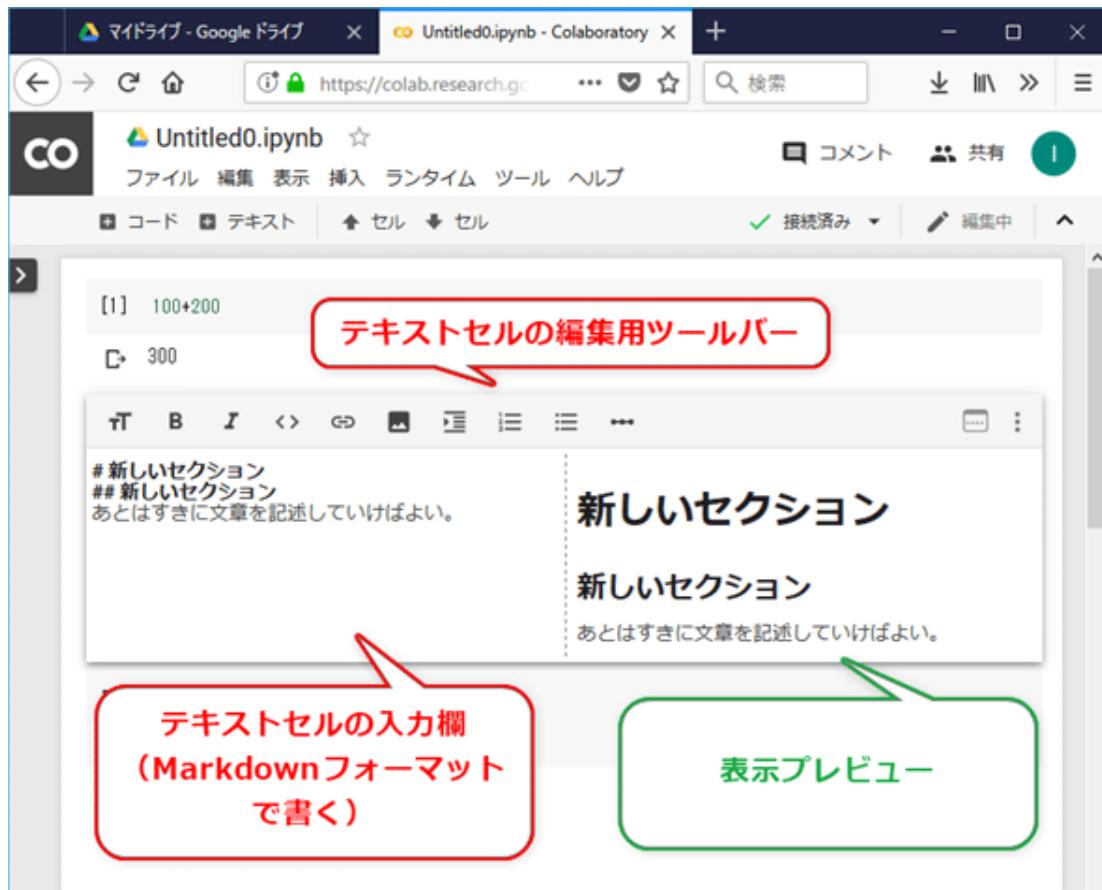


図3.10

出所: ITmediaより. 公式Webサイト |テキスト作成中の画面表示状態

テキスト・タイトルの目出し機能ボタン

見出し（ヘッダー）は、ドキュメント全体に階層構造（いわゆる章や節など）を作るのに欠かせない

例えば、1行の文を見出しに加工したい場合は、その行の先頭に#という記号を付けるだけである

以下に示すとおり、##、###のように#の数を増やすことで、見出しの階層構造を表現できる

Markdownのコード例

レベル1の見出ししたいとき

```
#  
## レベル2の見出し  
### レベル3の見出し
```



図3.11

出所: ITmediaより. 公式Webサイト |Markdownフォーマット見出しボタン

目次表示ボタン

目次欄にはテキスト欄に記入したタイトルがかてに表示される

目次を見たい場合は > をクリックするとタイトルが見出し設定通りに表示される

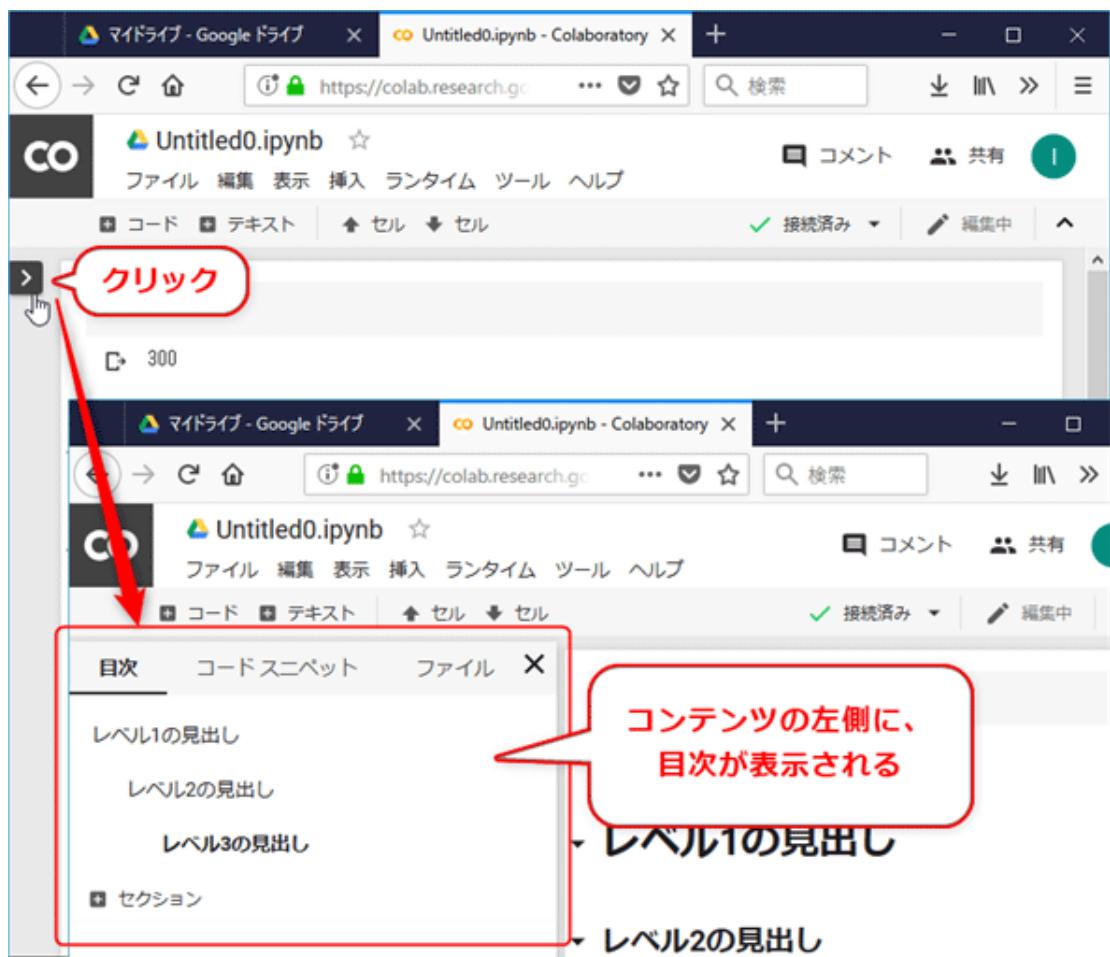


図3.12

出所: ITmediaより. 公式Webサイト |目次表示ボタン

Colabサーバー接続確認ボタン

Colabサーバー接続が有効な場合、ランタイム（ノートブック使用中）に接続ができるいる状態といい、接続済という表示に変わる

通常は自動的に接続されるが、使用時間制限などにより接続が切れる場合がある

そのときは再度接続をクリックするとよい



図3.13

出所: ITmediaより. 公式Webサイト | Colabサーバー接続確認ボタン

Pythonパッケージのインストール

Python演算向けに以下のパッケージをインストールできる

注意：パッケージが更新される場合もあるため、最新バージョンのインストールをする必要がある

```
pip install numpy  
pip install pandas  
pip install matplotlib  
pip install requests  
pip install beautifulsoup4  
pip install scikit-learn  
pip install tensorflow  
pip install torch torchvision  
pip install seaborn
```

```
pip install flask  
pip install django
```

パッケージインストール例:

- 1.コードセルを開く
- 2.コードを記入
- 3.shitt+enter または 実行ボタンを押す
- 4.インストール完了
- 5.成功すると下記の表示が現れる

Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages
(1.22.4)

```
In [ ]: #2  
pip install numpy  
  
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-pac  
kages (1.22.4)
```

Pythonによる文字列の表示

Pythonによる文字列の表示の流れ:

1. 表示させたい文字をシングルコート"またはダブルコート""内にテキストを入力
2. '表示させたい文字Hello Worldを記入'
- print(): 表示命令
- print('Hello World')
3. 実行
4. 文字 Hello World が表示

```
In [ ]: #3  
print('Hello World')  
  
Hello World
```

Python言語の変数(Variables) と演算子

Python言語の変数(Variables) と演算子表示の流れ:

- 1.変数をa,b,cと指定して定義する

```
a=1 (a変数を整数1とする)  
b=2 (b変数を整数2とする)  
c=a+b(c変数を変数aとbの足し演算子とする)
```

2. 変数a,b,c表示命令をする

, は区切りの意味

()なかに入れると引数と呼ばれる

print(): 表示命令

```
print(a,b,c)
```

3. 実行

4. 定義した式通りに結果が表示される

1 2 3

```
In [ ]: #Python言語の変数(Variables)と演算子表示  
a = 1  
b = 2  
c = a + b  
print(a,b,c)
```

1 2 3

例1: Pythonによる文字列に関する文法

テキストを定義して表示する流れ:

1. テキストを定義する

```
text = 'Hello World'
```

2. 定義した名前で表示命令をする

print(): 表示命令

```
print(text)
```

3. 実行

4. Hello Worldが表示される

```
In [ ]: #例1  
text = 'Hello World'  
print(text)
```

Hello World

例2:連結表示を指定する

連結表示を指定する流れ：

1. 表示させたい文字列の連結定義をする、+ サインで連結させる"で区切る

```
'Hello' + 'World'
```

2. 表示命令

```
print('Hello' + 'World')
```

3. 実行

4. Hello Worldが表示される

つまり、コードの組み方がさまざままで同じ結果が出せることが確認できます。

```
In [ ]: #例2  
print('Hello' + 'World')
```

```
HelloWorld
```

例3:定義したtextの要素を順番決めて表示させる

定義したtextの要素を順番決めて表示させる流れ:

1. テキストを定義する、テキスト文字（要素）数が10個ある

```
text = 'Hello World'
```

2. 要素の2と3番目([1:3]は2から3番までの意味)のテキストを表示する命令

```
print(text[1:3])
```

3. 実行

4. elが表示される

```
In [ ]: #例3  
text = 'Hello World'  
print(text[1:3])
```

```
el
```

例4:複数の形式（文字列・数字）を同時に表示させる

複数の形式（文字列・数字）を同時に表示させる流れ:

この場合複数の形式を変数別で定義をする

1. 複数変数a,b,cを違った形式で個別に定義する

```
a='Test' (a変数を文字列Testとする)  
b=100 (b変数を整数100とする)  
c=3.14159(c変数を浮動小数点3.14159とする)
```

2. 変数a,b,cを3種類の表示命令で表示させる

2-1. 「文字列= 文字列」 の表示命令

```
print('テスト= {}'.format(a))
```

3-1. テスト= Testが表示される

2-2. 「文字列= 整数」 の表示命令

```
print('点数= {}'.format(b))
```

3-2. 点数= 100が表示される

2-3. 「文字列= 浮動小数点」 の表示命令

{ } : 表示エリア

format():定義された変数の埋め込む場所を () で指定する

print(): 表示命令

```
print('πの値= {}'.format(c))
```

{ } 内に表示の指定ができる

{:.2f}:浮動小数点の行数(例：2) を指定する

```
print('πの値= {:.2f}'.format(c))
```

ちがう形式を同時に表示をするときは事前に変数別の定義をしておけば このコードで同時に表示することができる

{ } , { } , { } :3つ表示したいので表示エリアを3つ書く

定義された変数の埋め込む場所も同時にカンマ区切り (a,b,c) で指定する

```
print('複数変数= {},{}, {:.2f}'.format(a,b,c))
```

3-3. 以下のように3種類の浮動小数点指示通りに表示される

πの値= 3.14159、cをそのまま表示

πの値= 3.14、2桁表示

複数変数= Test,100,3.14、a b c (2桁) の表示

```
In [ ]: #2-1
a='Test'
b=100
c=3.14159
print('テスト= {}'.format(a))
```

テスト= Test

```
In [ ]: #2-2
a='Test'
b=100
c=3.14159
print('点数= {}'.format(b))
```

点数= 100

```
In [ ]: #2-3
a='Test'
b=100
c=3.14159
print('πの値= {}'.format(c))
print('πの値= {:.2f}'.format(c))
print('複数変数= {},{},{}'.format(a,b,c))
```

πの値= 3.14159

πの値= 3.14

複数変数= Test,100,3.14

Python言語のリスト作成

Python言語のレンジによるリストの作成（数字リスト）

Python言語のレンジによるリストの作成（数字リスト）の流れ:

1. 表示したい数字をレンジ決めて定義

for num in range (5): レンジ5 (0~4) の数字という意味になる

```
for num in range (5):
```

2. 表示命令

```
print(num)
```

3. 01234が表示される

```
In [ ]: #レンジ(range)によるリストの作成
for num in range(5):
    print(num)
```

```
0  
1  
2  
3  
4
```

Python言語のリスト作成（数字・文字列類のリスト）

Python言語のリスト作成（数字リスト）の流れ：

1. リストの定義

```
list = [1,2,3,4,5]
```

2. 表示命令

```
print(list)
```

3. 選択表示

```
print(list[0])
```

4. 以下のようにリストが指定された表示命令の通りに結果が得られる

```
[1,2,3,4,5]
```

```
1
```

```
In [ ]: #Python言語の数字リスト作成  
list = [1,2,3,4,5]  
print(list)  
print(list[0])
```

```
[1, 2, 3, 4, 5]  
1
```

Python言語のリスト作成（文字列類のリスト）の流れ：

1. リストの定義

```
list = ['Apple', 'Banana', 'Citrus', 'Dorian',  
'Grapefruit']
```

2. 表示命令

```
print(list)
```

3. 選択表示

```
print(list[0])
```

4. 以下のようにリストが指定された表示命令の通りに結果が得られる

```
['Apple', 'Banana', 'Citrus', 'Dorian', 'Grapefruit']
```

Apple

In []:

```
#Python言語の文字列リスト作成
list = ['Apple', 'Banana', 'Citrus', 'Dorian', 'Grapefruit']
print(list)
print(list[0])
```

```
['Apple', 'Banana', 'Citrus', 'Dorian', 'Grapefruit']
Apple
```

Python言語のリストの要素の取得

リストの要素取得の流れ:

1. リストの定義

```
list = [1,2,3,4,5]
```

2. リストの最初のアイテムを1から10へ変更するための定義

list [0]:list の [0]のアイテム

= 10 : を10にする

```
list [0] = 10
```

3. 表示命令

```
print(list)
```

4. リストのアイテムを変更して表示する

```
list [1:5]= [20,30,40]
```

```
print(list)
```

5. 以下のようにリストが指定された表示命令の通りに結果が得られる

```
[10,2,3,4,5]
```

```
[10,20,30,40]
```

In []:

```
#リストの要素取得
list = [1,2,3,4,5]
list [0] = 10
print(list)
```

```
list [1:5] = [20,30,40]
print(list)
```

```
[10, 2, 3, 4, 5]
[10, 20, 30, 40]
```

Python言語のリスト要素の追加

リストの要素追加の流れ:

1. リストの定義

'文字列'2つをリスト化する

```
list = ['Apple', 'Cherry']
```

2. リストの追加定義

リスト名のあとに .append を書くと追加するという意味になる

うしろで () 内に追加したい'文字列'を書く

注意：文字列は"中に書くこと

```
list.append('Strawberry')
```

3. 表示命令

```
print(list)
```

4. ['Apple', 'Cherry', 'Strawberry']が表示される

```
In [ ]: #リストの要素追加
list = ['Apple', 'Cherry']
list.append('Strawberry')
print(list)
```

```
['Apple', 'Cherry', 'Strawberry']
```

Python言語のリスト要素の挿入

リストの要素挿入の流れ :

1. リストの定義

'文字列'3つをリスト化する

```
list = ['Apple', 'Cherry', 'Strawberry']
```

2. リストの挿入定義

リスト名のあとに .insert を書くと挿入するという意味になる

うしろで () 内に挿入位置の番号(0は最初の位置) や挿入する'文字列'の'Banana'を書く

```
list.insert(0, 'Banana')
```

3.表示命令

```
print(list)
```

4.['Banana','Apple', 'Cherry', 'Strawberry']が表示される

```
In [ ]: #リストの要素挿入  
list = ['Apple', 'Cherry', 'Strawberry']  
list.insert(0, 'Banana')  
print(list)
```

```
['Banana', 'Apple', 'Cherry', 'Strawberry']
```

Python言語のリスト要素の削除

リストの要素削除の流れ:

1. リストの定義

'文字列' 4つをリスト化する

```
list = ['Banana', 'Apple', 'Cherry', 'Strawberry']
```

2-1.リストの削除定義

リスト名のあとに .remove を書くと削除するという意味になる

うしろで () 内に削除する'文字列'の'Apple'を書く

```
list.remove('Apple')
```

3.表示命令

```
print(list)
```

4. ['Banana','Cherry', 'Strawberry']が表示される

2-2.別の削除し方:

del: 削除 (deleteの略)

delのうしろに削除したいリスト名を書いて削除アイテムの位置をできる

```
del list[0]:listの最初[0]のアイテムを削除するという意味になる
```

```
del list[0]
```

3-2.表示命令

```
print(list)
```

4-2,['Cherry', 'Strawberry']が表示される

```
In [ ]: #リストの要素削除
list = ['Banana', 'Apple', 'Cherry', 'Strawberry']
list.remove('Apple')
print(list)
del list[0]
print(list)
```

['Banana', 'Cherry', 'Strawberry']
['Cherry', 'Strawberry']

Python言語の辞書作成と取得

辞書作成と取得の流れ:

1.辞書を定義する

```
dic = {} :辞書の定義フォーマット
```

```
{' ': data, }:辞書のアイテム名'文字列' : データ (数字)
```

```
dic = {'Apple': 300, 'Cherry ': 200, 'Strawberry ': 3000 }
```

2.表示命令

この場合dicのリスト[]から'Apple'を取得すると命令

```
print(dic['Apple'])
```

3. データの300が表示される

```
In [ ]: #辞書作成と取得
dic = {'Apple': 300, 'Cherry ': 200, 'Strawberry ': 3000 }
print(dic['Apple'])
```

300

Python言語の辞書の要素更新

辞書の要素更新の流れ:

1. 定義した辞書のリストに対して更新したいリストを指定

```
dic = {'Apple': 300, 'Cherry ': 200, 'Strawberry ': 3000 }
```

2. 辞書の更新

この場合dicのリスト[]から'Apple'を取得し、データの300を400と更新する

```
dic['Apple'] = 400
```

3. 更新した内容を表示命令して確認

```
print(dic)
```

4. {'Apple': 400, 'Cherry ': 200, 'Strawberry ': 3000}が表示される

```
In [ ]: #辞書の要素更  
dic = {'Apple': 300, 'Cherry ': 200, 'Strawberry ': 3000 }  
dic['Apple'] = 400  
print(dic)
```

```
{'Apple': 400, 'Cherry ': 200, 'Strawberry ': 3000}
```

Python言語の辞書の要素追加

辞書の要素追加の流れ:

1. 定義した辞書のリストに対して要素の追加したいリストを指定

```
dic = {'Apple': 300, 'Cherry ': 200, 'Strawberry ': 3000 }
```

2. 辞書のアイテム追加

この場合dicのリスト[]に'Banana'を追加し、データを400とする

```
dic['Banana'] = 400
```

3. 追加した内容を表示命令して確認

```
print(dic)
```

4. {'Apple': 400, 'Cherry ': 200, 'Strawberry ': 3000, 'Banana ': 400}が表示される

```
In [ ]: #辞書の要素追加  
dic = {'Apple': 300, 'Cherry ': 200, 'Strawberry ': 3000 }  
dic['Banana'] = 400  
print(dic)
```

```
{'Apple': 300, 'Cherry ': 200, 'Strawberry ': 3000, 'Banana': 400}
```

Python言語の辞書の要素削除

辞書の要素削除の流れ:

1.定義した辞書のリストに対して要素の削除したいリストを指定

```
dic = {'Apple': 300, 'Cherry ': 200, 'Strawberry ': 3000 }
```

2.辞書のアイテム削除

この場合dicのリスト[]に'Apple'を削除する

```
del dic['Apple']
```

3.削除した内容を表示命令して確認

```
print(dic)
```

4.{'Cherry ': 200, 'Strawberry ': 3000}が表示される

```
In [ ]: #辞書の要素削除  
dic = {'Apple': 300, 'Cherry ': 200, 'Strawberry ': 3000 }  
del dic['Apple']  
print(dic)
```

```
{'Cherry ': 200, 'Strawberry ': 3000}
```

Python言語のタプル作成

タプル作成の流れ:

1.タプルの定義

```
taple = (1,2,3,4)
```

2.表示命令

```
print(taple)
```

3.(1,2,3,4)が表示される

```
In [ ]: #タプル作成  
taple = (1,2,3,4)  
print(taple)
```

```
(1, 2, 3, 4)
```

Python言語のタプルの要素取得

タプルの要素取得の流れ:

1.タプルの定義

```
taple = (1,2,3,4)
```

2.表示命令

最初のアイテムを表示

```
print(taple[0])
```

2～3番目のアイテムを表示

```
print(taple[1:3])
```

3.

1が表示される

(2,3)が表示される

```
In [ ]: #タプルの要素取得  
taple = (1,2,3,4)  
print(taple[0])  
print(taple[1:3])
```

```
1  
(2, 3)
```

Python言語の制御構文作成

if（条件分岐）関数で制御文を書く

if（条件分岐）関数で制御文を書く流れ:

1.数字リストを定義

```
numbers = [5, 8, 2, 10, 3]
```

2.条件>5数字を制御する関数を書く

```
for num in numbers:  
    if num > 5:
```

3.表示命令

```
print(num)
```

4.

8

10 が表示される

```
In [ ]: #if(条件分岐)関数で制御文を書く
numbers = [5, 8, 2, 10, 3]

# Print numbers greater than 5
for num in numbers:
    if num > 5:
        print(num)
```

8

10

if else (条件分岐) 関数で制御文を書く

if else (条件分岐) 関数で制御文を書く流れ:

if:もしもの条件 else:もしもの条件と違った条件

1.数字リストを定義

```
numbers = [5, 8, 2, 10, 3]
```

2.もしnumbersリスト中にnumがあるときの条件を制御する関数を書く

```
for num in numbers:
```

3.数字は2と同じ (==2)ときの条件を制御する関数を書く

```
if num % 2 == 0:
```

4.偶数のときにevenと書くことを表示命令とする

num:数字

f"{num}:表示エリア{}中にnumにあたる条件 (数字) を表示させる

```
print(f"{num} is even.")
```

5.上記の命令と違った条件 (奇数) を表示する命令

```
        else:  
            print(f"{num} is odd.")
```

6.

1 is odd.

2 is even.

3 is odd.

4 is even.

5 is odd.

が表示される

```
In [ ]: #if else(条件分岐)関数で制御文を書く  
numbers = [1, 2, 3, 4, 5]  
  
for num in numbers:  
    if num % 2 == 0:  
        print(f"{num} is even.")  
    else:  
        print(f"{num} is odd.")
```

1 is odd.
2 is even.
3 is odd.
4 is even.
5 is odd.

if elif else (条件分岐) 関数で制御文を書く

if elif else (条件分岐) 関数で制御文を書く流れ:

if:もしもの条件 elif: ifと違うもしもの条件 else:前述全てと違った条件

1. 数字リストを定義

```
numbers = [10, 5, -3, 0, 8]
```

2.もしnumbersリスト中にnumがあるときの条件を制御する関数を書く

```
for num in numbers:
```

3.num>0 の条件を制御する関数を書く

```
if num > 0:
```

4.numは (+) のときにis a positive numberと書くことを表示命令とする

num:数字

f"{num}:表示エリア{}中にnumにあたる条件（数字）を表示させる

```
print(f"{num} is a positive number.")
```

5.num<0 の条件を制御する関数を書く

```
elif num < 0:
```

6.上記の命令と違った条件 numが (-) のときを表示する命令を書く

```
print(f"{num} is a negative number.")
```

7.上記のnum +/- の条件と違ったnum (0)のときを表示する命令を書く

```
else:  
    print(f"{num} is zero.")
```

8.

10 is a positive number.

5 is a positive number.

-3 is a negative number.

0 is zero.

8 is a positive number.

が表示される

```
In [ ]: #if elif else(条件分岐)関数で制御文を書く  
numbers = [10, 5, -3, 0, 8]  
  
for num in numbers:  
    if num > 0:  
        print(f"{num} is a positive number.")  
    elif num < 0:  
        print(f"{num} is a negative number.")  
    else:  
        print(f"{num} is zero.")
```

10 is a positive number.

5 is a positive number.

-3 is a negative number.

0 is zero.

8 is a positive number.

for (繰り返し) 関数で制御文を書く

for (繰り返し) 関数で制御文を書く流れ:

for関数でリストの要素を順番に変数に代入しながら繰り返し処理を行う制御文

例1.nはリスト[]の「中in」にある「ための:for」条件と書く

```
for n in [1,2,3]:
```

表示命令

```
print(n)
```

例2.計算式を代入して表示命令をすることもできる

```
print(n*10)
```

例3:別のfor関数で数字をレンジの形で表示させることができる

nはレンジ()の「中in」にある「ための:for」条件と書く

(5) : 0~4

```
for n in range(5):
```

表示命令

```
print(n)
```

```
In [ ]: #例1: for(繰り返し)関数で制御文を書く
for n in [1,2,3]:
    print(n)
```

1
2
3

```
In [ ]: #例2: for(繰り返し)関数で制御文を書く
for n in [1,2,3]:
    print(n*10)
```

10
20
30

```
In [ ]: #例3: for(繰り返し)関数で制御文を書く
for n in range(5):
    print(n)
```

0
1
2
3
4

while (繰り返し) 関数で制御文を書く

while (繰り返し) 関数で制御文を書く流れ:

while:条件が成立している間に処理を繰り返す制御条件

1.変数 i の開始ポイントを 0 と決める

i = 0

2. i<10 という「成立させる条件」を決める

while i < 10:

3. i += 1はi = i + 1という意味で「繰り返し処理」条件を書く

i += 1

4. i % 2 == 0は、iが2で割った余りが0と等しいかどうかを「判定する処理条件」を書く

if i % 2 == 0:

5.

繰り返し対象エリアはwhileより内側（インデント）に書く

このエリア内に処理を繰り返して終わったた先頭に戻るよう「続き条件:continue」を書く

continue

6.

表示させる

print(i)

7.

1

3

5

7

9 が表示される

```
In [ ]: #while(繰り返し)関数で制御文を書く
i = 0

while i < 10:
    i += 1
    if i % 2 == 0:
        continue
    print(i)
```

1
3
5
7
9

enumerate (行挙) 関数で制御文を書く

enumerate (行挙) 関数で制御文を書く流れ:

1. リストを定義

```
fruits = ["apple", "banana", "cherry"]
```

2. Enumerateの定義

for index条件:変数をindexとする条件のため

enumerate()関数:リストや他のシーケンス (タプル、文字列など) の要素に対して「インデックス:index」と要素のペアを返す「イテレータ:Iterator」を作成する関数

enumerate関数で返す結果の表示例:

Index 0: 要素0

Index 1: 要素1

Index 2: 要素2

index変数に要素のインデックスが代入、fruit変数に要素・値が代入される制御文

for インデックス変数, 変数 in enumerate(リスト):

```
for index, fruit in enumerate(fruits):
```

3. 表示命令

f"Index{index}:表示エリア{ }中にindexにあたる条件 (数字) を表示させる

{fruit}:表示エリア{}中にfruitにあたる条件（要素・値）を表示させる

```
print(f"Index {index}: {fruit}")
```

4.以下のようにenumerate関数制御によってインデックス:要素の連番が表示される

Index 0: apple

Index 1: banana

Index 2: cherry

```
In [ ]: #enumerate(行挙)関数で制御文を書く  
fruits = ["apple", "banana", "cherry"]  
for index, fruit in enumerate(fruits):  
    print(f"Index {index}: {fruit}")
```

Index 0: apple

Index 1: banana

Index 2: cherry

内包表記の関数で制御文を書く

内包表記の関数で制御文を書く流れ:

例1:

1. 空のリストを作成する

```
list = []
```

2. レンジ(10):0 ~ 9の「変数:x」条件「のため:for」制御文を書く

```
for x in range(10):
```

3 メソッド.appendで要素を作成した空のリストに「変数:x」を[]内包表記させる

```
list.append(x)
```

4. 表示命令

```
print(list)
```

5. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]が表示される

例2:

新たにlistの2号を作る、作ったlistと混合しないためにリスト名を番号つけて管理することができる

空のリスト作らずに[]なかにレンジの数式をfor制御文で書いて定義する方法がある

```
for x in range (10) : これは上記のレンジ制御文と同じ
```

このレンジ制御文の前に数式 $x * 2$ を追記するだけで数式つきのfor制御文が書ける

```
list2 = [x * 2 for x in range (10)]
```

表示命令（ここではlist2を表示させるのでlist2を()中に書く）

```
print(list2)
```

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]が表示される

```
In [ ]: #例1: 内包表記の関数で制御文を書く
list = []

for x in range(10):
    list.append(x)

print(list)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [ ]: ##例2: 内包表記の関数で制御文を書く
list2 = [x * 2 for x in range (10)]

print(list2)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Python言語の関数とlambda式作成

関数とlambda式作成流れ:

2種類のlambda関数の定義し方がある

例1:

def: definitionによる定義する方法:

```
def 関数名 (引数)
```

```
    def radian(x):
```

return 戻り数の数式

```
        return x / 180 * 3.1415
```

例2:

lambda 引数：戻り値のある関数による定義する方法:

```
lambda_radian = (lambda x:x / 180 * 3.1415)
```

ラムダ関数を用いて度数からラジアンに変換する関数を定義する実行例：

1. Lambda定義

ラジアン (radian) :角度の単位

1ラジアン:半径と弧の長さが等しい場合の中心角が約57.3度 ($\pi/180$) に相当する

ラジアン = (角度 * π) / 180

lambda x:ラムダ関数 lambda_radian は、度数 x を入力として () 中に受け取り

x / 180 * 3.1415 :計算によってラジアンに変換する

```
lambda_radian = (lambda x:x / 180 * 3.1415)
```

2.条件の0度から360度まで90度ずつの角度に対して繰り返し制御文 (for ループ)

```
for x in range(0,360,90):
```

3.ラムダ関数を用いて度数をラジアンに変換させて表示すると命令

度:{} 度数の表示エリア

ラジアン:{:.2f}' 2桁の表示命令

.format(x, lambda_radian(x)) lambda計算結果が値で表示される

```
print('度:{} ,ラジアン:{:.2f}'.format(x,
lambda_radian(x)))
```

4.

以下のように指定した順で表示される

度:0,ラジアン:0.00

度:90,ラジアン:1.57

度:180,ラジアン:3.14

度:270,ラジアン:4.71

```
In [ ]: #def方法による関数とlambda式作成
def radian(x):
```

```

    return x / 180 * 3.1415
for x in range(0,360,90):
    print('度:{} ,ラジアン:{}'.format(x, lambda_radian(x)))

度:0,ラジアン:0.00
度:90,ラジアン:1.57
度:180,ラジアン:3.14
度:270,ラジアン:4.71

```

```

In [ ]: #dlambda引数方法による関数とlambda式作成
lambda_radian = (lambda x:x / 180 * 3.1415)

for x in range(0,360,90):
    print('度:{} ,ラジアン:{}'.format(x, lambda_radian(x)))

度:0,ラジアン:0.00
度:90,ラジアン:1.57
度:180,ラジアン:3.14
度:270,ラジアン:4.71

```

Python言語のクラス作成

クラスの作成流れ:

1. クラスの定義のフォーマット

```

class クラス名:

    def __init__(self, 引数, 引数,...):

        def メソッド名(self, 引数, 引数,...):

```

2. クラス名をFruitと定義する

```
class Fruit:
```

3. Fruitクラスの中身（要素）のカテゴリー名（fruit, price）を定義する

```

    def __init__(self, fruit, price):

        self.fruit = fruit

        self.price = price

```

4. 要素のカテゴリーをさらに細かく定義する

```
要素 = クラス(" ", )
```

```

fruit1 = Fruit("りんご", 300)

fruit2 = Fruit("メロン", 500)

```

5. 出力の定義をする(1行目の表示の定義)

関数"要素1.要素" は {要素1の値段} 円です。

1 行目の表示命令

```
print(f"{fruit1.fruit} は {fruit1.price} 円です。")
```

6. 出力の定義をする(2行目の表示の定義)

```
def output(self): 出力をoutputと定義、selfメソッドで要素とリンクさせる
```

```
def output(self):
```

7. 関数"selfメソッドの要素を代入" は果物です。と表示させる

```
print(f"{self.fruit} は果物です。")
```

8. 2行目の表示命令

要素1を出力する

```
fruit1.output()
```

9. 要素2の出力定義や命令の説明は要素1と同じである

```
print(f"{fruit2.fruit} は {fruit2.price} 円です。")
```

```
fruit2.output()
```

10.

以下のように要素名と値段が定義した表示のフォーマットで表示される

リンゴ は 300 円です。

リンゴ は果物です。

メロン は 500 円です。

メロン は果物です。

```
In [ ]: #クラスの作成
class Fruit:
    def __init__(self, fruit, price):
        self.fruit = fruit
        self.price = price

    def output(self):
        print(f"{self.fruit} は果物です。")

fruit1 = Fruit("リンゴ", 300)
fruit2 = Fruit("メロン", 500)

print(f"{fruit1.fruit} は {fruit1.price} 円です。")
fruit1.output()
```

```
print(f'{fruit2.fruit} は {fruit2.price} 円です。')
fruit2.output()
```

リンゴ は 300 円です。
リンゴ は果物です。
メロン は 500 円です。
メロン は果物です。

Pythonのパッケージ呼び出し

パッケージ呼び出し流れ:

クラス・関数・定数・などを定義されたPythonのプログラムが「モジュール」と呼ばれ、

その複数のモジュールで構成されたものが「パッケージ」と呼ばれる

これらを使うにはパッケージ呼び出しが必要で、コードの最初にインポートをしてからモジュールが利用される

1. 呼び出したいパッケージをインポート

```
import numpy as np
```

2. 定義

このコードは、NumPyを使って3x3の行列（行列の要素が2次元配列で表現される）を作成する

```
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
```

- 3.表示命令

```
print(a)
```

```
In [ ]: #パッケージ呼び出し例
import numpy as np

a = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(a)
```

[[1 2 3]
 [4 5 6]
 [7 8 9]]