Google Colaboratory でのプログラミング

2024年 A1 木曜2限

担当： 地引

TA： 董/石井

Google Colaboratory の基本操作

最近の情報通信環境を取り巻く状況を考慮し、ここではクラウド環境上で提供される処理系の利用について説明します。

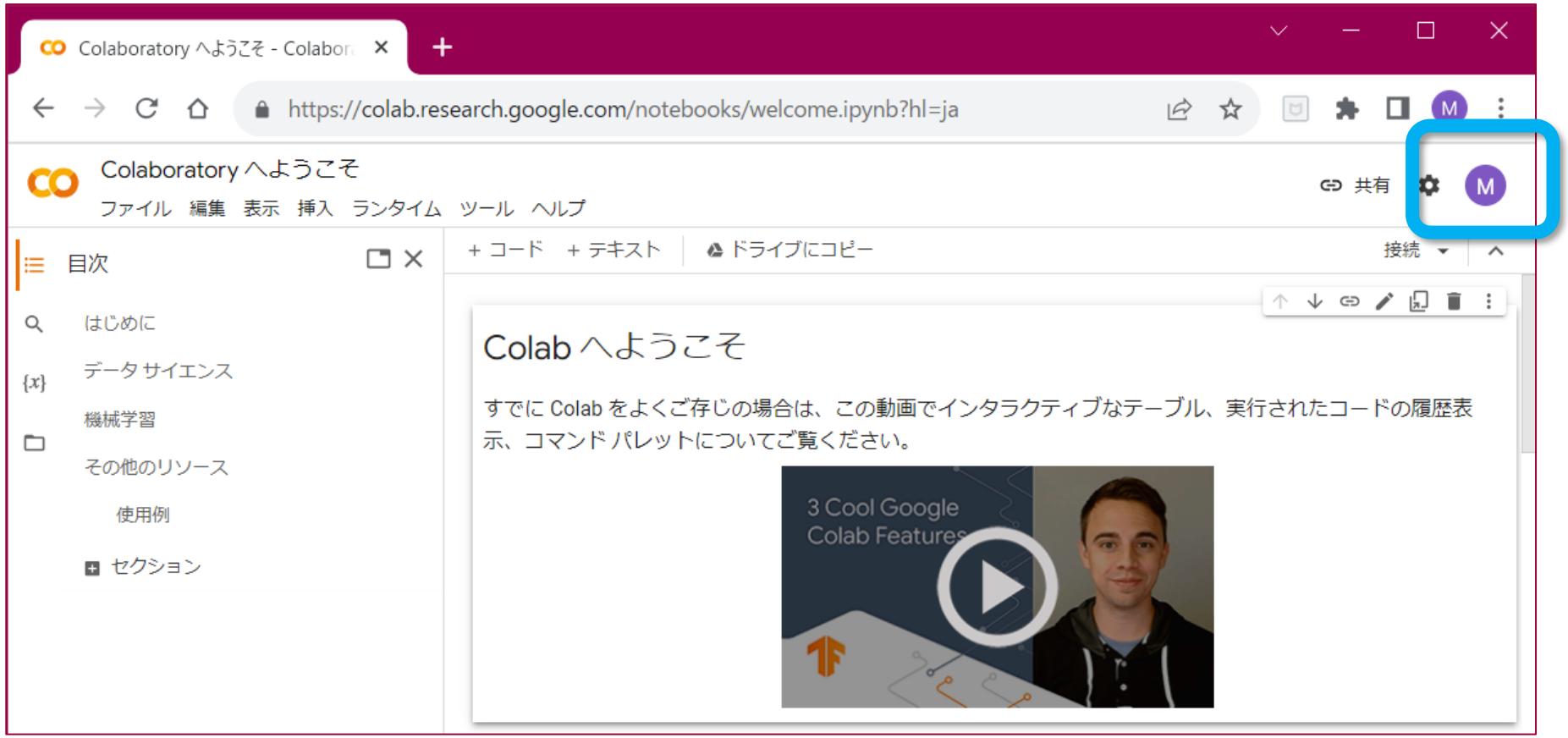
教育用計算機や個人のPCにインストールした処理系を利用してもよいです。

Google Coraboratory へのログイン（1）

- Google Coraboratory は、下記の URL からログインできます。
  - https://colab.research.google.com/notebooks/welcome.ipynb?hl=ja
  - 右端上に表示される ログイン をクリックし、ECCS クラウド メールのアカウント「????@g.ecc.u-tokyo.ac.jp」でログインして下さい。

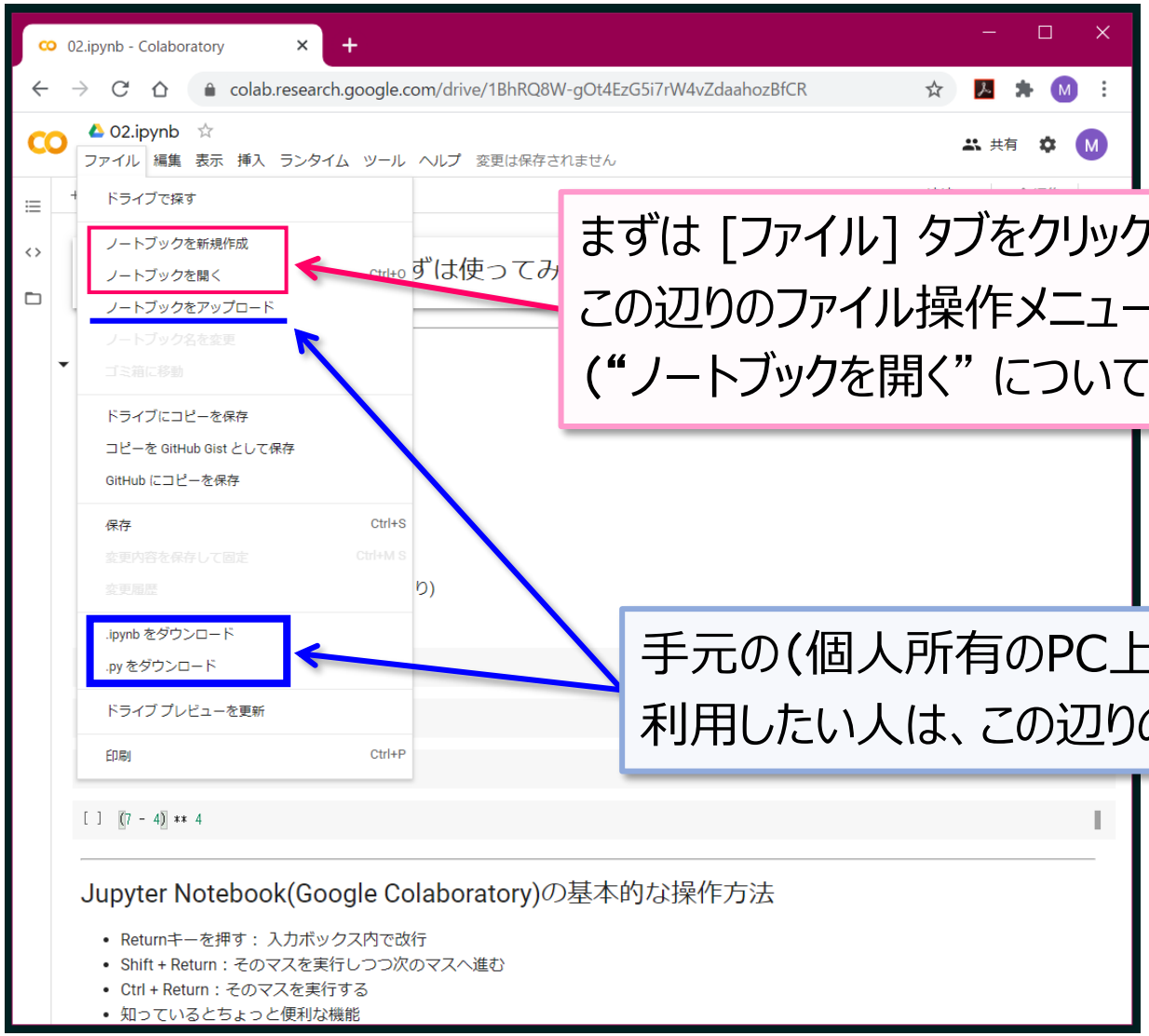
Google Coraboratory へのログイン（2）

ログイン後は、以下のようなウィンドウが表示されます（変化に気付きにくい･･･）。



確認

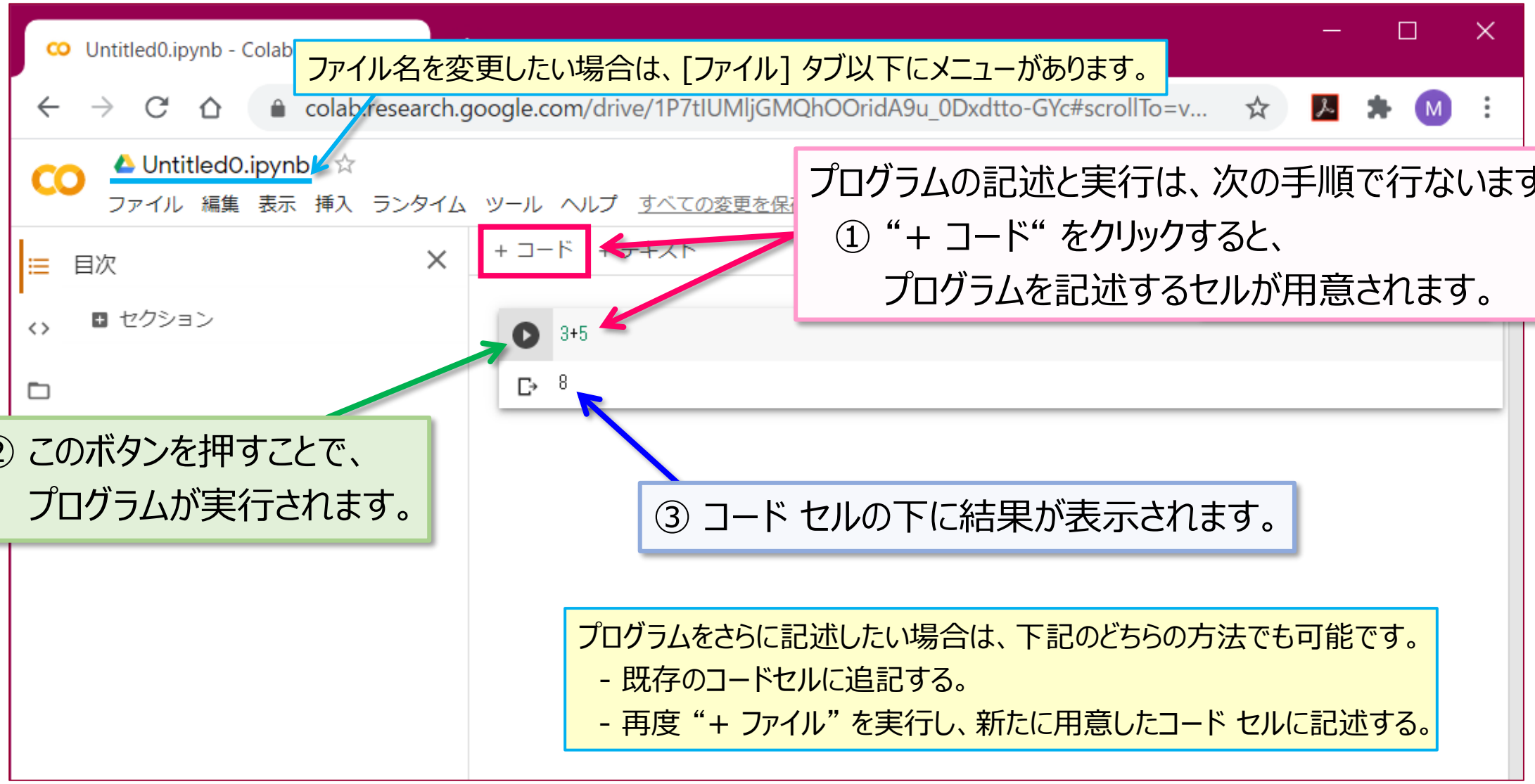
Python プログラム ファイルの操作



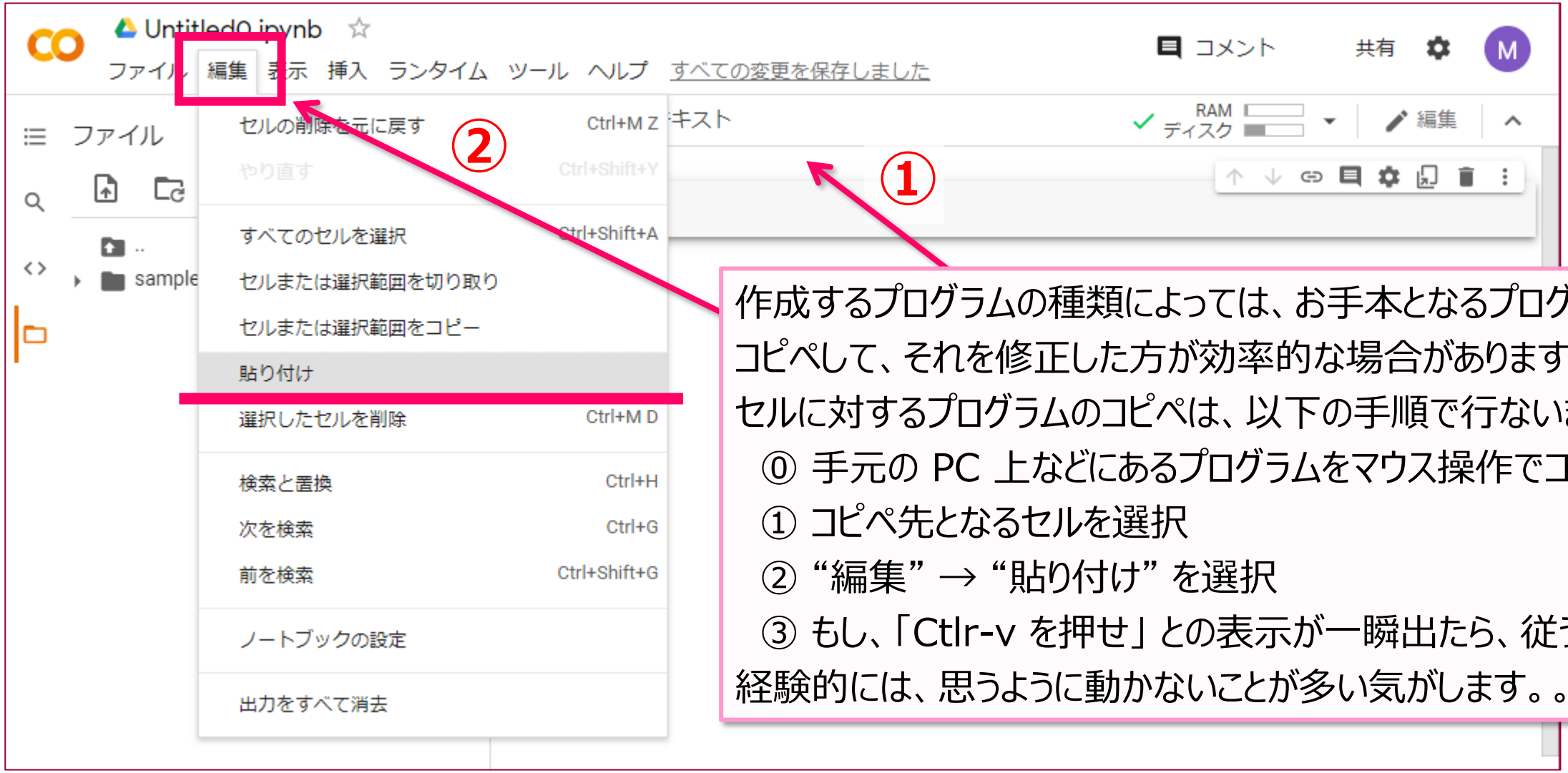
まずは [ファイル] タブをクリックします。
この辺りのファイル操作メニューは意味が分かるかと思います
(“ノートブックを開く”については、後ほど補足します)。

手元の(個人所有のPC上にある) Jupyter Notebook を利用したい人は、この辺りのファイル操作メニューを使います。

Python プログラムの記述・実行



プログラムのコピー&ペースト

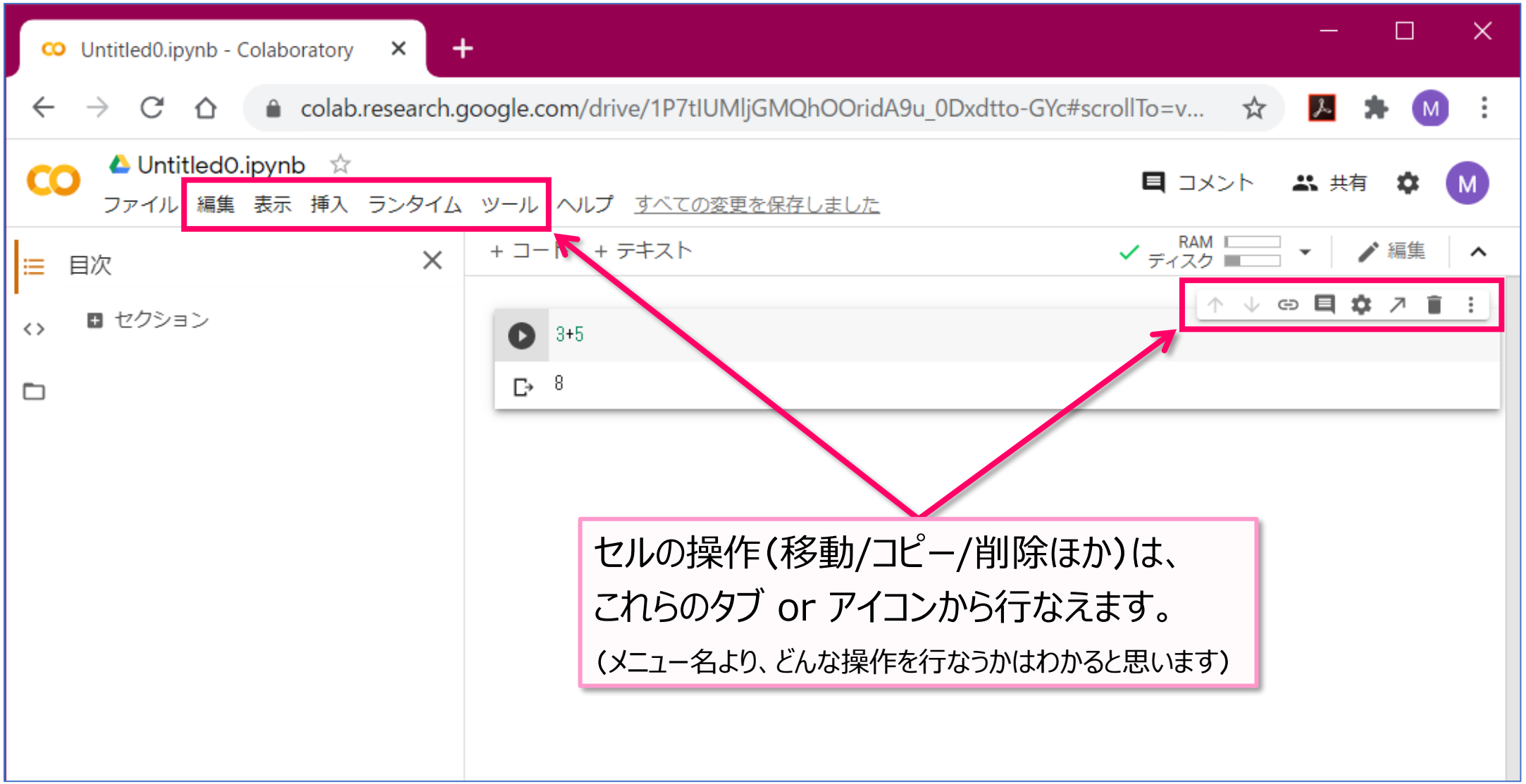


作成するプログラムの種類によっては、お手本となるプログラムをコピペして、それを修正した方が効率的な場合があります。
セルに対するプログラムのコピペは、以下の手順で行ないます。

⓪ 手元の PC 上などにあるプログラムをマウス操作でコピー
① コピペ先となるセルを選択
② “編集” → “貼り付け” を選択
③ もし、「Ctlr-v を押せ」との表示が一瞬出たら、従う。

経験的には、思うように動かないことが多い気がします。。

セルの操作


Untitled0.ipynb - Colaboratory
colab.research.google.com/drive/1P7tIUMljGMQhOOridA9u_0Dxdtto-GYc#scrollTo=v...
Untitled0.ipynb
ファイル 編集 表示 挿入 ランタイム ツール ヘルプ すべての変更を保存しました
コメント
共有
目次
セクション
+ コード + テキスト
RAM
ディスク
編集
3+5
8
セルの操作（移動/コピー/削除ほか）は、
これらのタブ or アイコンから行なえます。
（メニュー名より、どんな操作を行なうかはわかると思います）

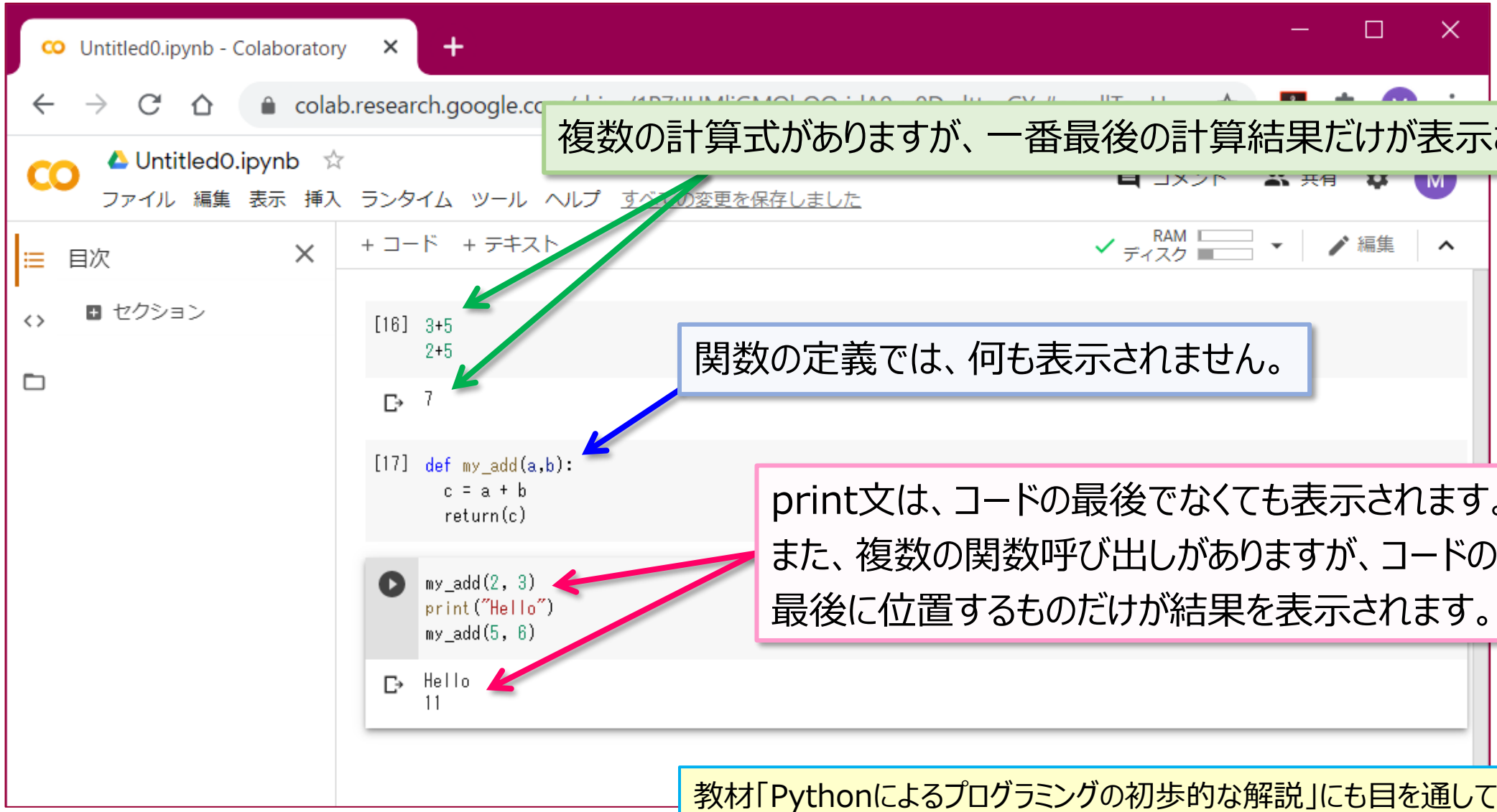
注意：Python プログラムの実行結果（1）

- 教材「Python によるプログラミングの初歩的な解説」でも説明していますが、Python プログラムの実行結果は、基本的に次の方針に沿って表示されます。

  1. print 文などにより明示的に表示を指定された場合は、それを表示する。
  2. コード セルにある一番最後のコードが結果を表示できる場合は、それを表示する（それより前にあるコードは、上記 1 以外表示されません）。
     - ≫ 計算式やクォーテーションで囲った文字列は、上記 2 に該当します。
     - ≫ 呼び出した関数が値を返す場合は、上記 2 に該当します。
     - ≫ 関数定義は、上記 2 に該当しません。

- Google Colaboratory での実行例を見てみましょう。

注意：Python プログラムの実行結果（2）



複数の計算式がありますが、一番最後の計算結果だけが表示されます。

関数の定義では、何も表示されません。

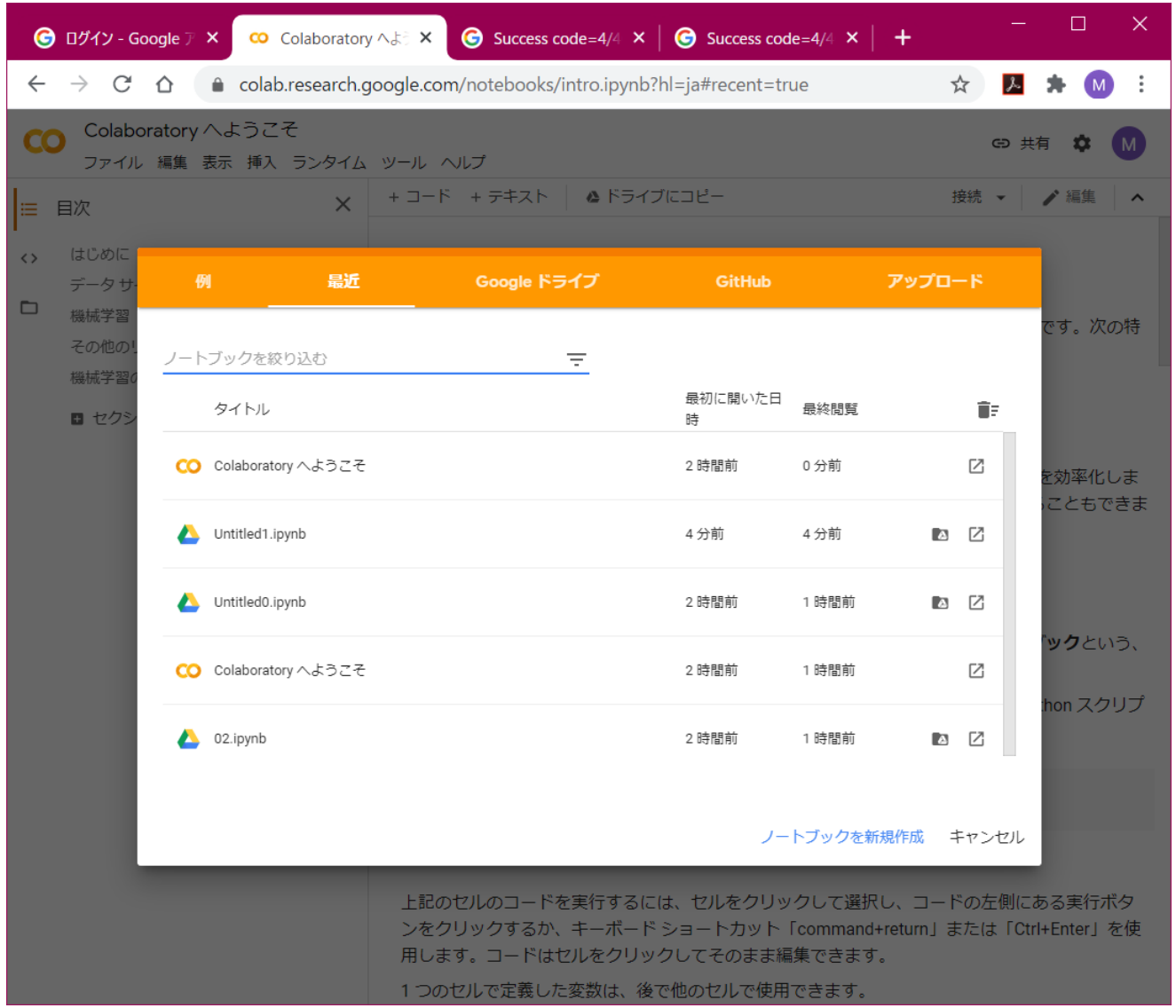
print文は、コードの最後でなくても表示されます。
また、複数の関数呼び出しがありますが、コードの最後に位置するものだけが結果を表示されます。

教材「Pythonによるプログラミングの初歩的な解説」にも目を通しておいて下さい。

ノートブックを開く

Google Colaboratory や Jupyter Notebook では、プログラム ファイルをノートブックと呼びます。

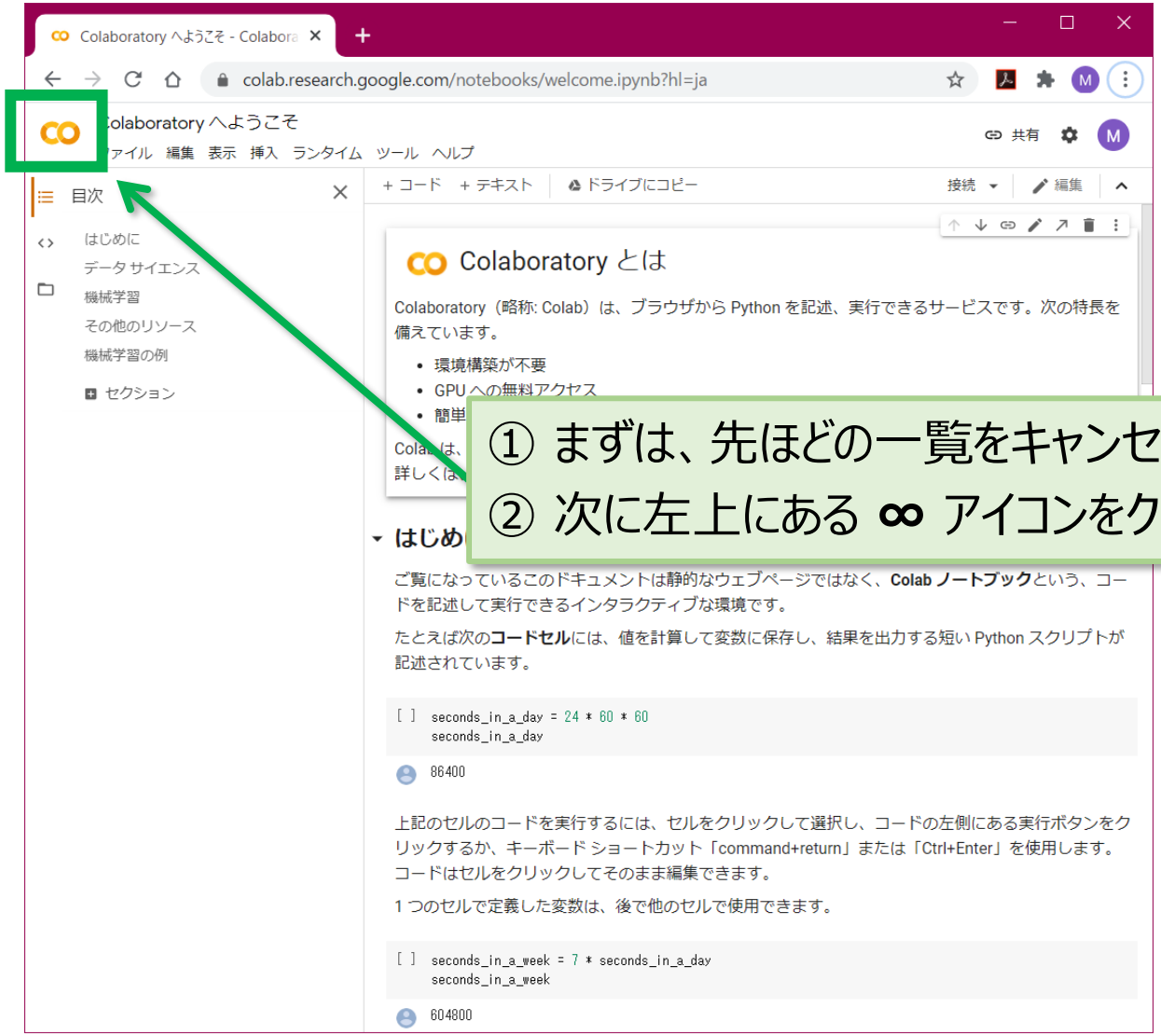
ノートブックを開く（1）



Google Coraboratory にログインする or [ファイル] → [ノートブックを開く] を選択すると、左のようなファイル一覧が表示されます。
しかし、これは最近開いたファイルの一覧であり、自分が開きたいファイルはこの中にないかも知れません（履歴を消すなど）。

そのような場合は、これから説明する方法でファイルを探して下さい。

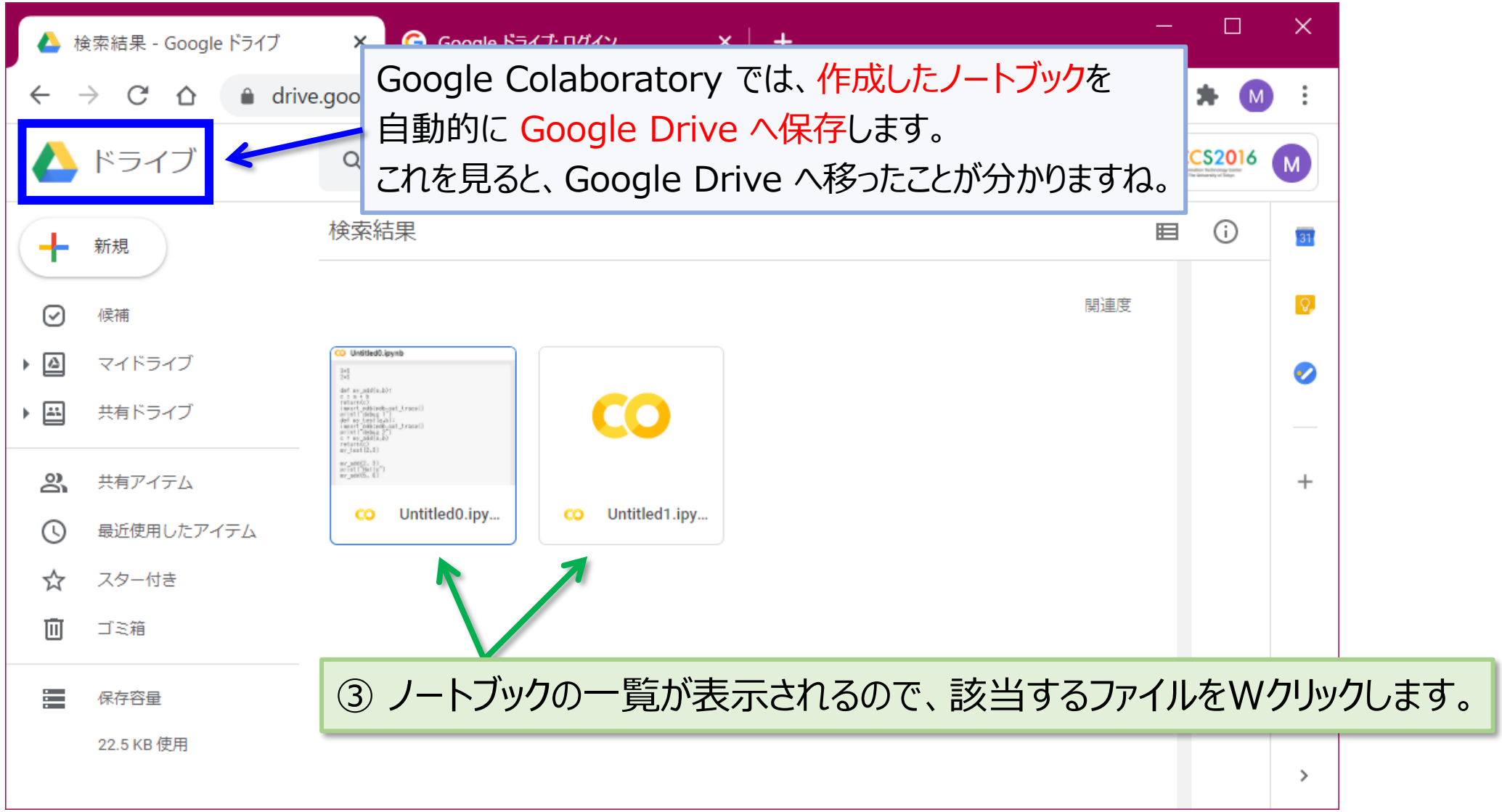
ノートブックを開く（2）



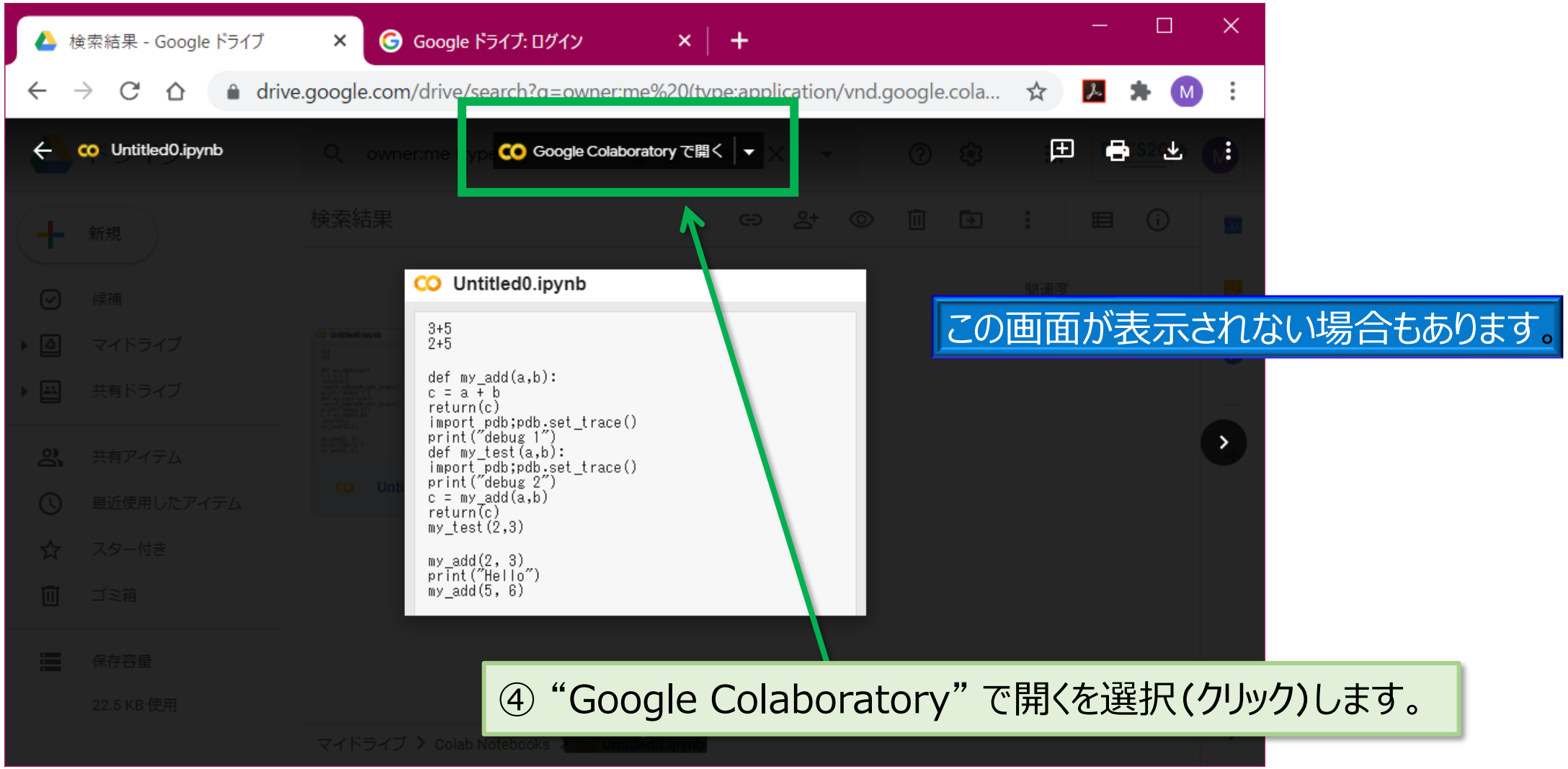
① まずは、先ほどの一覧をキャンセルし、適当なファイルを開きます。

② 次に左上にある ∞ アイコンをクリックします。

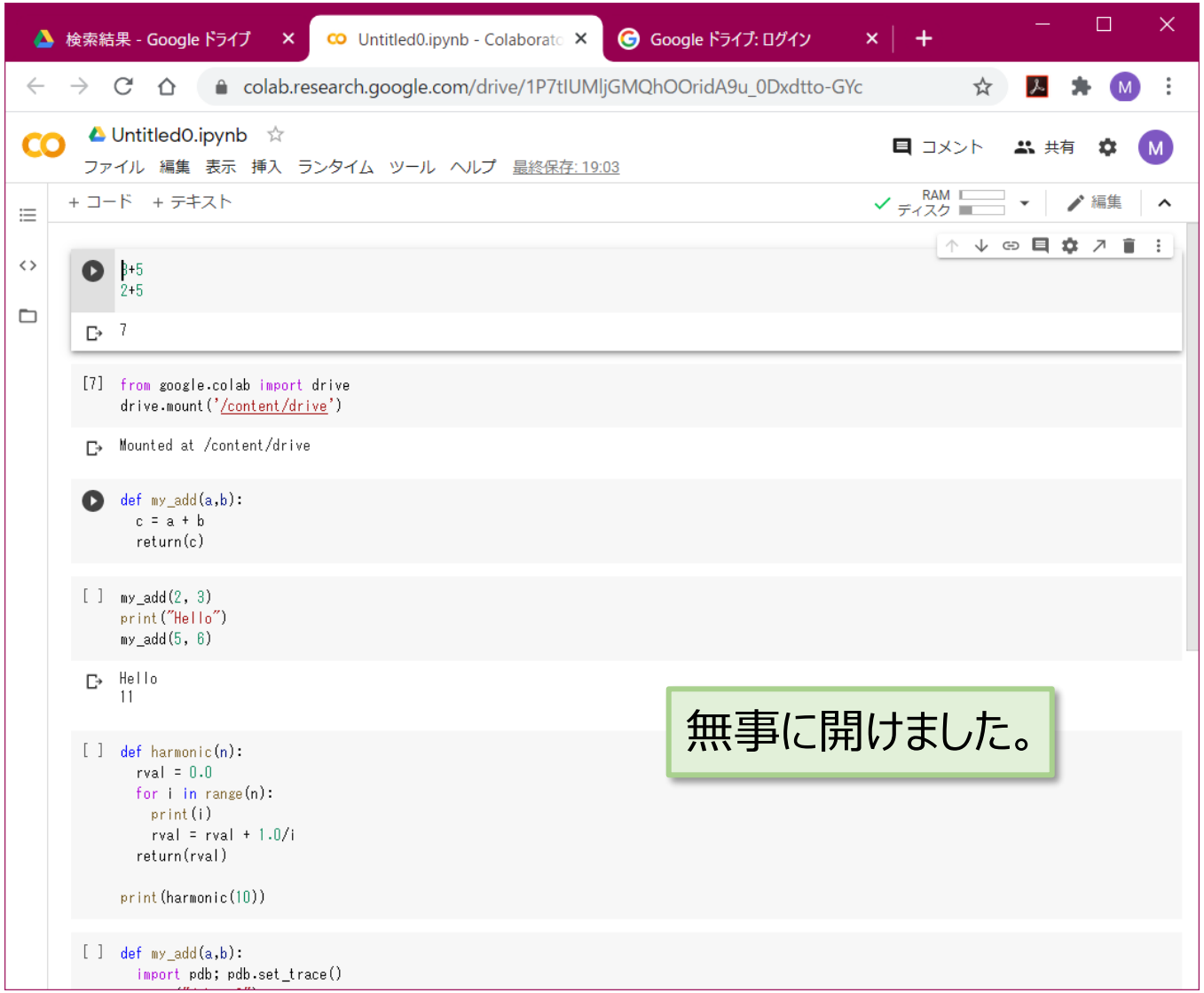
ノートブックを開く（3）


検索結果 - Google ドライブ
Google ドライブ: ログイン
drive.goo
ドライブ
Google Colaboratory では、作成したノートブックを
自動的に Google Drive へ保存します。
これを見ると、Google Drive へ移ったことが分かりますね。
新規
候補
マイドライブ
共有ドライブ
共有アイテム
最近使用したアイテム
スター付き
ゴミ箱
保存容量
22.5 KB 使用
検索結果
関連度
Untitled0.ipy...
Untitled1.ipy...
③ ノートブックの一覧が表示されるので、該当するファイルをWクリックします。

ノートブックを開く（4）

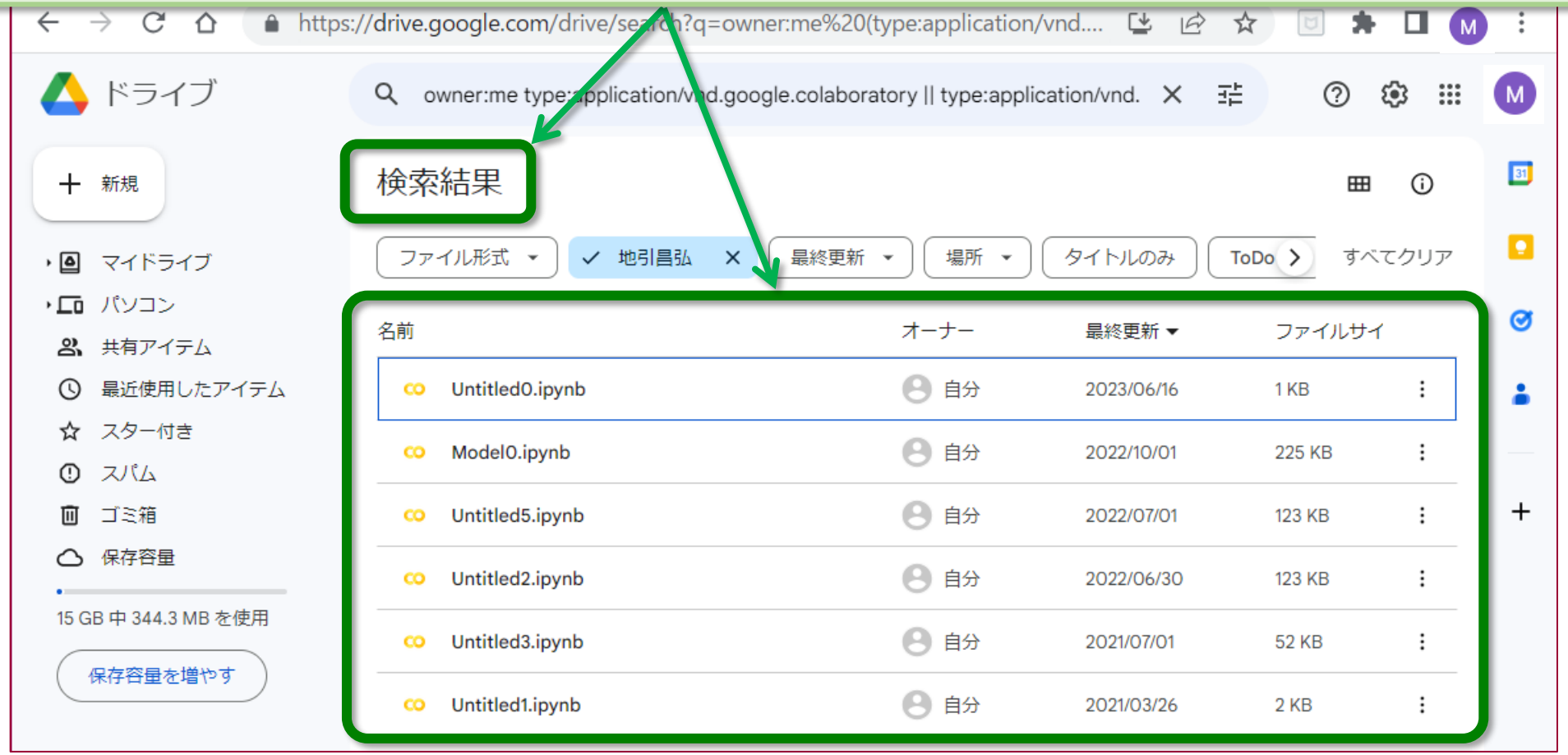

検索結果 - Google ドライブ
Google ドライブ: ログイン
drive.google.com/drive/search?q=owner:me%20(type:application/vnd.google.cola...
Untitled0.ipynb
Google Colaboratory で開く
新規
検索結果
候補
マイドライブ
共有ドライブ
共有アイテム
最近使用したアイテム
スター付き
ゴミ箱
保存容量
22.5 KB 使用
Untitled0.ipynb
3+5
2+5
def my_add(a,b):
c = a + b
return(c)
import pdb;pdb.set_trace()
print("debug 1")
def my_test(a,b):
import pdb;pdb.set_trace()
print("debug 2")
c = my_add(a,b)
return(c)
my_test(2,3)
my_add(2, 3)
print("Hello")
my_add(5, 6)
この画面が表示されない場合もあります。
④ “Google Colaboratory” で開くを選択(クリック)します。

ノートブックを開く（5）


検索結果 - Google ドライブ
Untitled0.ipynb - Colaborat
Google ドライブ: ログイン
colab.research.google.com/drive/1P7tIUMljGMQhOOridA9u_0Dxdtto-GYc
Untitled0.ipynb
ファイル 編集 表示 挿入 ランタイム ツール ヘルプ 最終保存: 19:03
コメント
共有
+ コード + テキスト
RAM
ディスク
編集
0+5
2+5
7
[7] from google.colab import drive
drive.mount('/content/drive')
Mounted at /content/drive
def my_add(a,b):
c = a + b
return(c)
[ ] my_add(2, 3)
print("Hello")
my_add(5, 6)
Hello
11
[ ] def harmonic(n):
rval = 0.0
for i in range(n):
print(i)
rval = rval + 1.0/i
return(rval)
print(harmonic(10))
[ ] def my_add(a,b):
import pdb; pdb.set_trace()
無事に開けました。

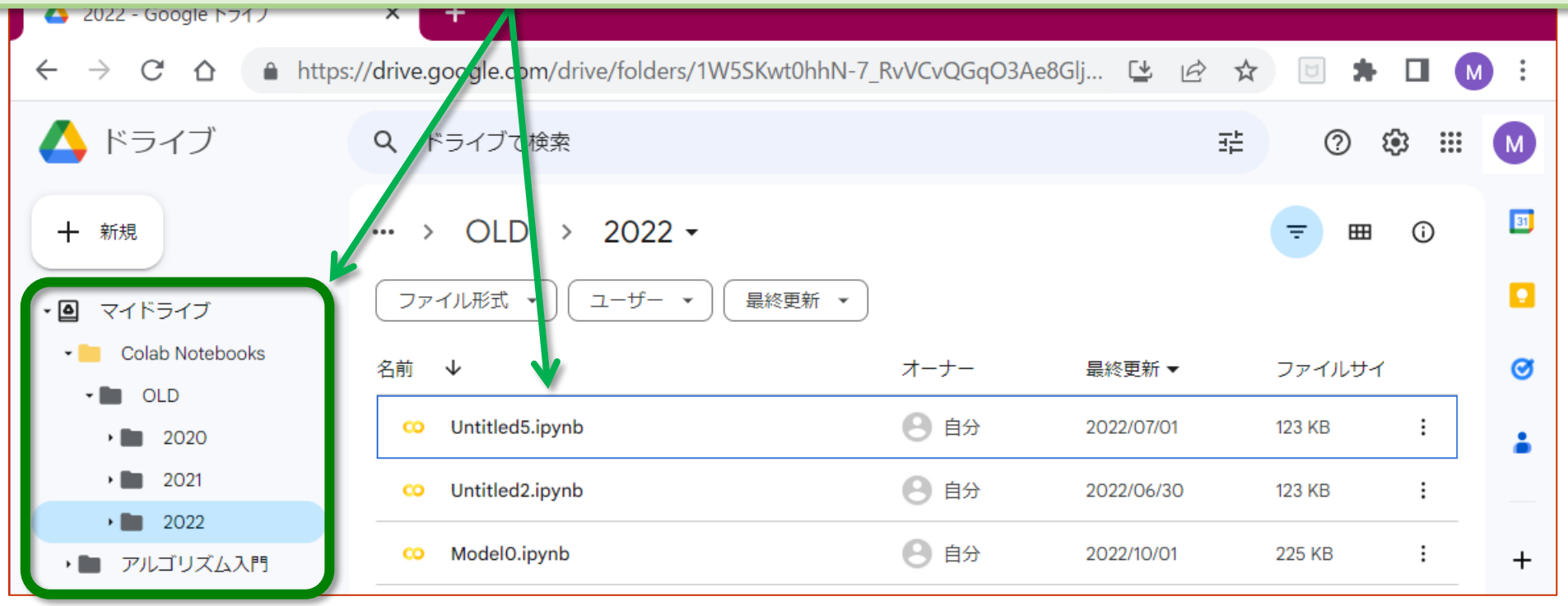
重要!!（1）

とは言え、ここに一覧されるノートブックは、Google Drive 内にある全ノートブックの検索結果です。
当然、ノートブックが増えて来ると、ズラズラと一覧され、お目当てのノートブックを探すのも大変です。



重要!!（2）

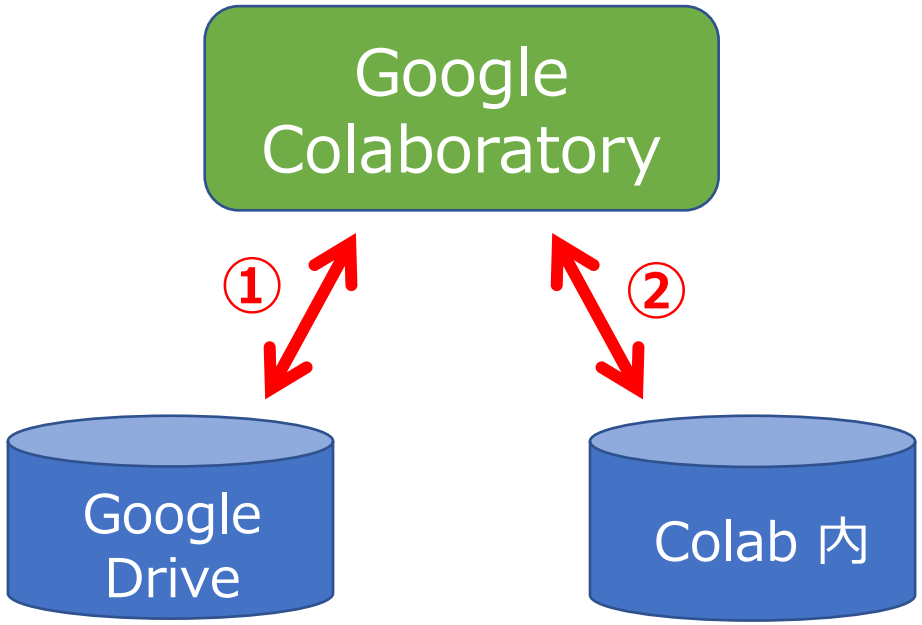
保存先は Google Drive なので、別途 Google Drive にログインして、“Colab Notebooks” フォルダ以下にサブ フォルダを作成し、適宜ノートブックを分類して整理しましょう。
Colab からノートブックを開く時は、前スライドにある検索結果一覧ではなく、左側に表示される ファイル システムを辿ってお目当てのノートブックを指定できるようになりましょう (重要なスキルです)。
Colab のメニューは、Google Drive をブラウザで開いた時と同じなので、操作も難しくないはずです。



データ ファイルの扱い

Google Colaboratory のファイル管理イメージ

Google Colaboratory では、扱うファイルを下記①,②のどちらかに保存します。
プログラム ファイルは、特に準備をしなくても①に保存されますが、データ用のファイルを保存する場合は、どちらに保存しても事前の準備や保存後の扱いが少々面倒です。



①にあるデータ ファイル(プログラム ファイルではない)を Colab からアクセスするには、Google Drive をマウント(一つのファイル システムとして接続)する必要があります。
以降のスライドで説明しますが、現状ではマウント処理を簡便に行なえない場合があるという欠点があります。
②の方はマウントが不要で、より直感的に扱えますが、保存したファイルは、一定時間後に削除されてしまうという欠点があります。

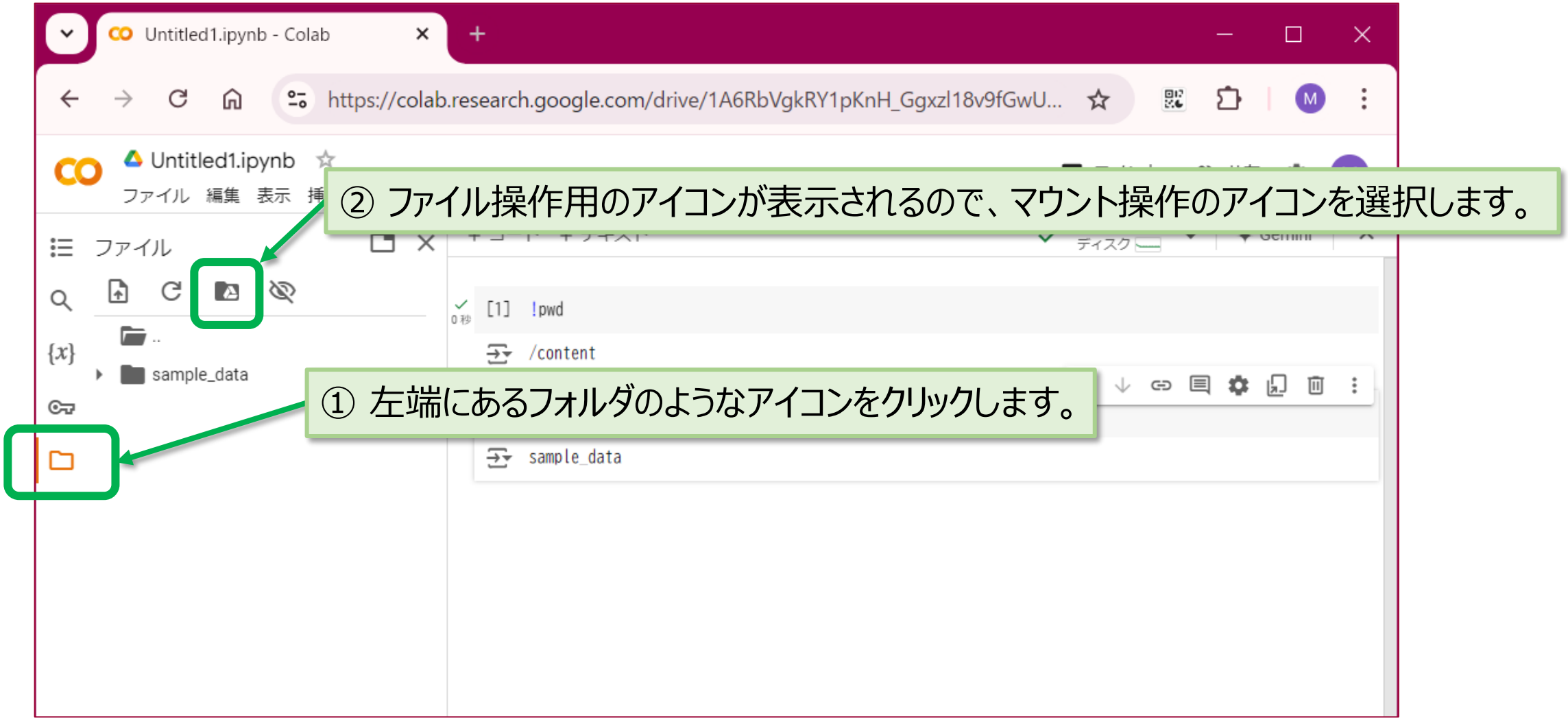
Google Drive のマウント（1）

- 以下では、Colab 上の Python コードがアクセスするデータ ファイルの保存先として、Google Drive を利用する手順を説明します。

- 次スライド以降の手順により、Colab に Drive をマウントすることで、Python コードから Drive 内のデータ ファイルへアクセスできます。

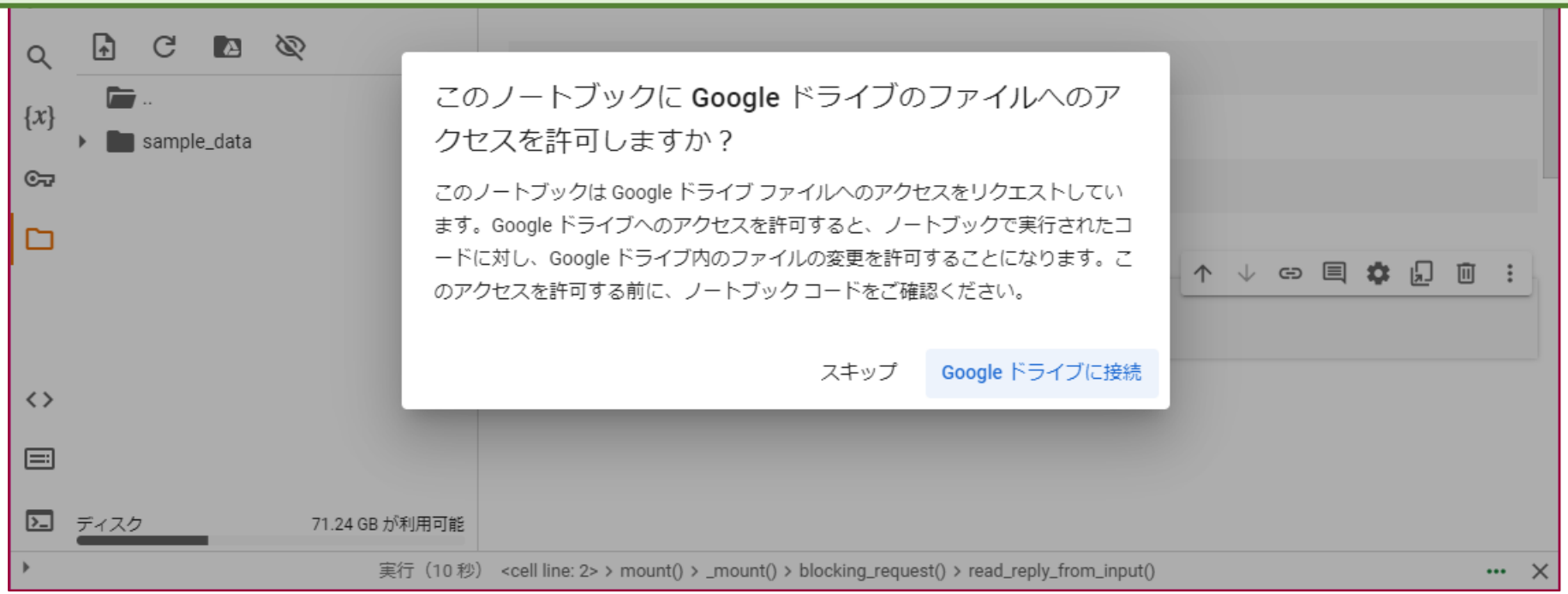
- 但し、Python コード内にファイルのパス名を正しく記載しないと入力/出力ができないことに注意して下さい。
  - 詳細は、“【重要】データ ファイルへのアクセス” スライドを参照
  - パス名の利用に慣れましょう → これができないと今後何もできない…

Google Drive のマウント（2）



Google Drive のマウント（3）

このような感じでGoogle Drive のマウント処理が自動的に始まった場合は、
“Google Drive のマウント（６）” スライド以降を参照して下さい。
自動的に始まらなかった場合は次スライドへ。

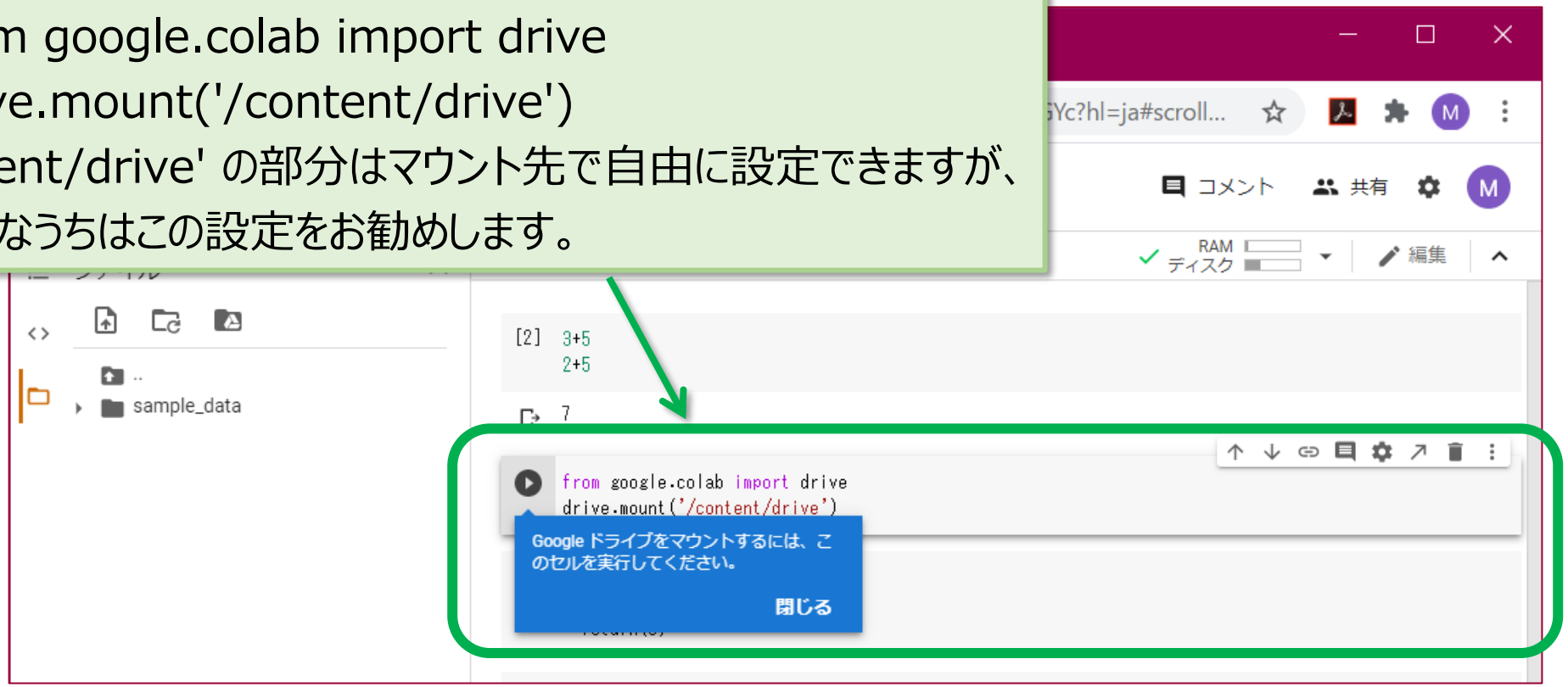


Google Drive のマウント（4）

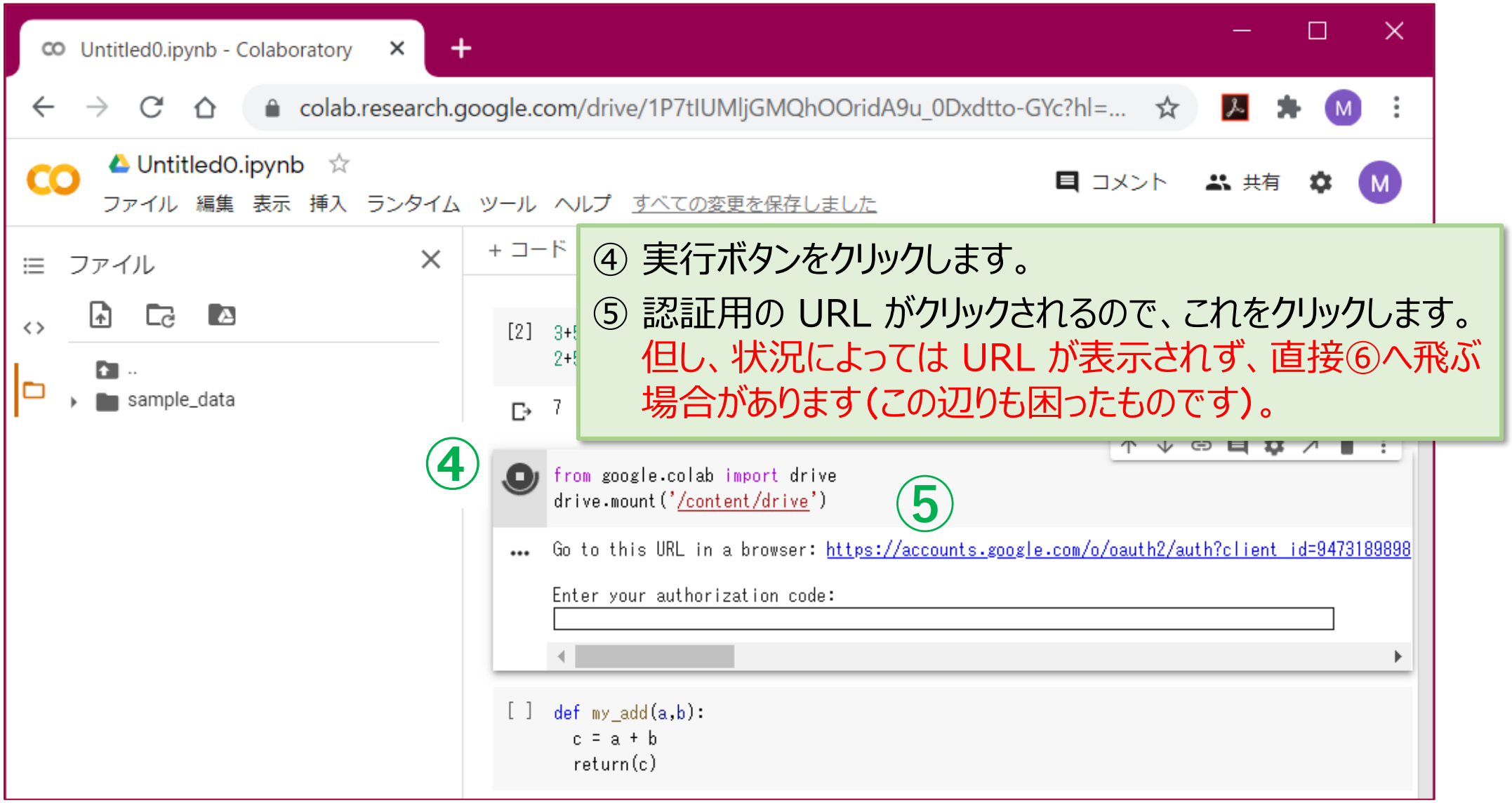
③ 前スライドのようにマウント処理が自動的に開始されず、
マウントにに必要なプログラムの実行が表示された場合は、
そのセルを実行します。
参考: マウントに必要なプログラム

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

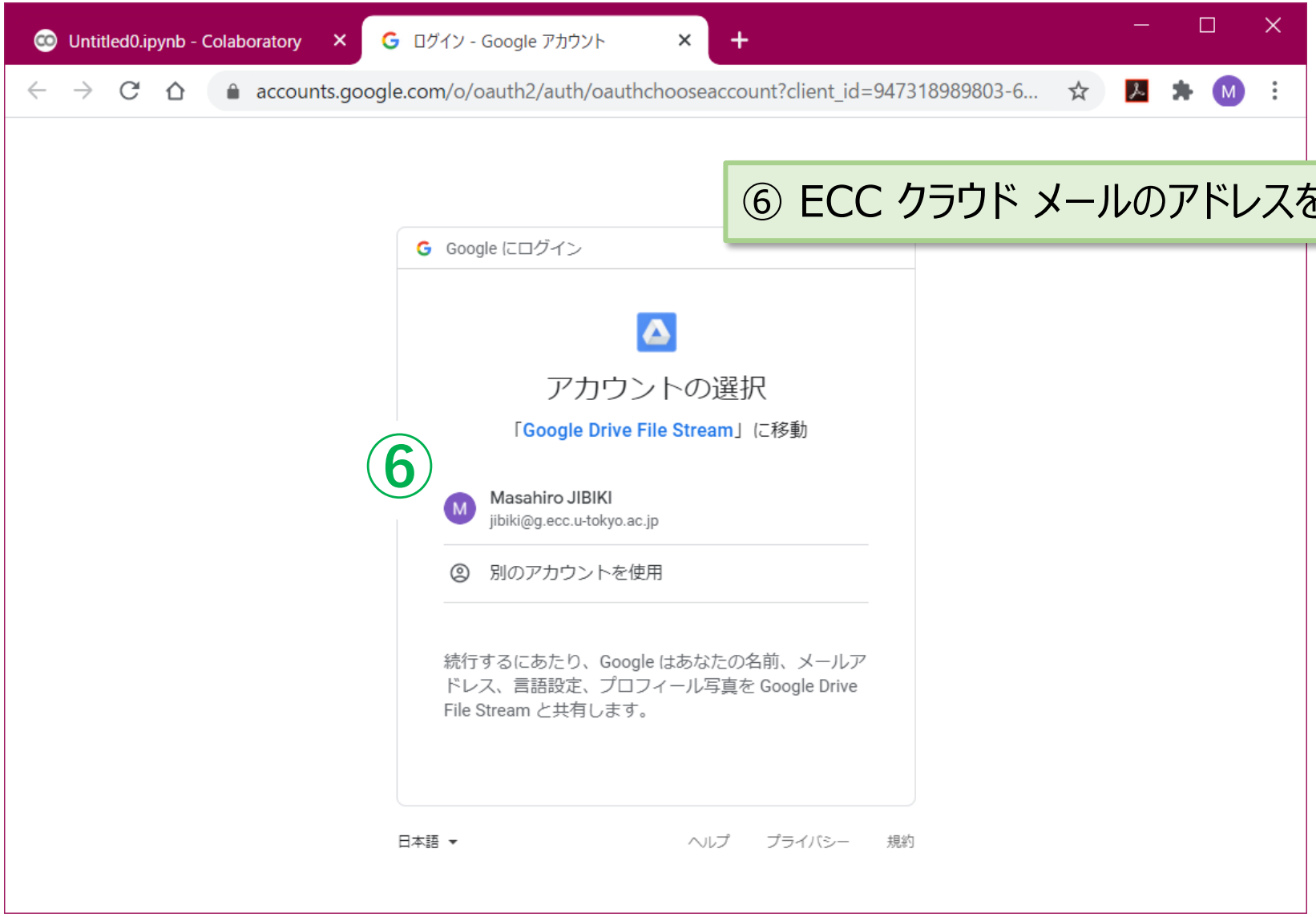
'/content/drive' の部分はマウント先で自由に設定できますが、
不慣れなうちはこの設定をお勧めします。



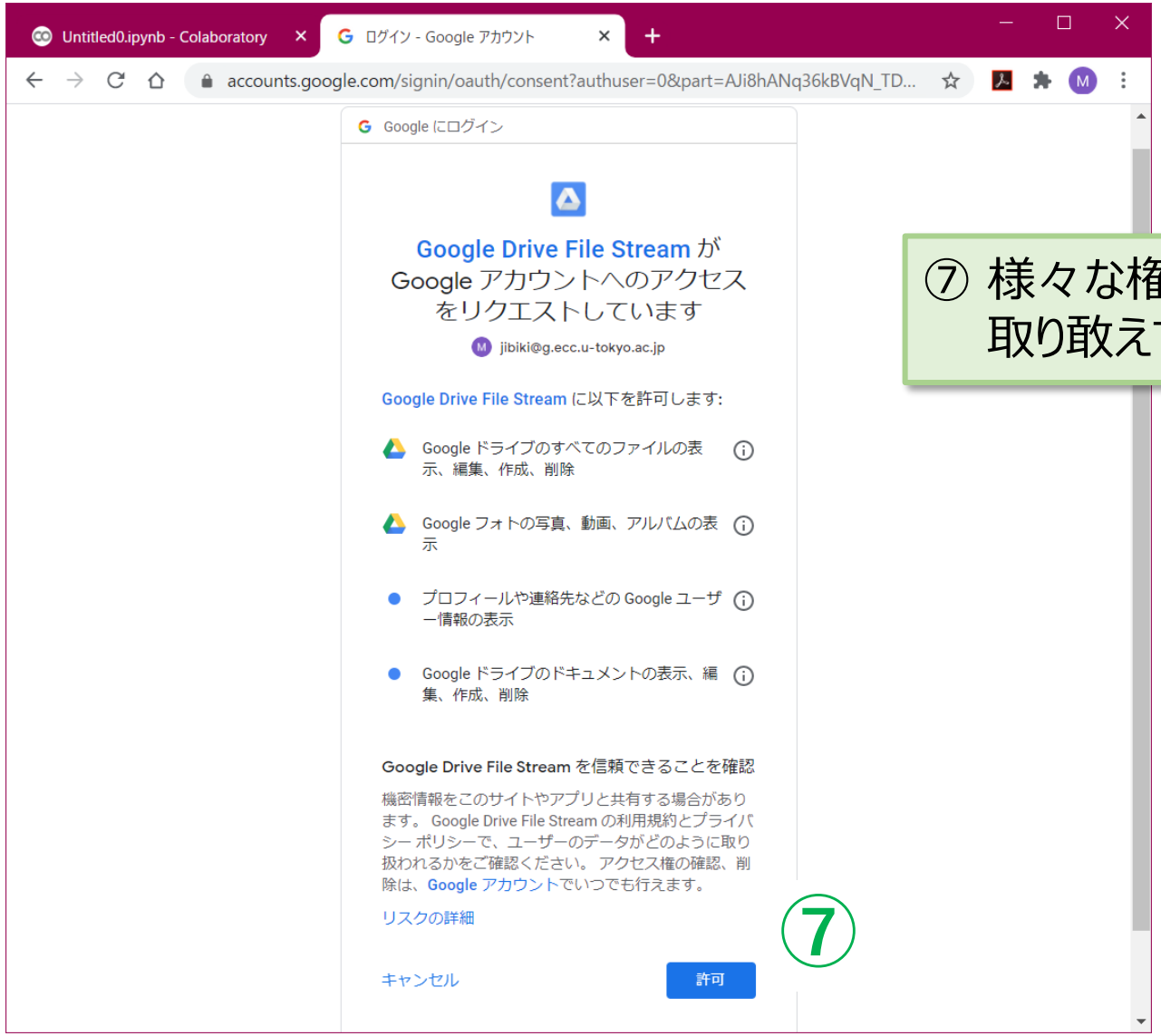
Google Drive のマウント（5）



Google Drive のマウント（6）

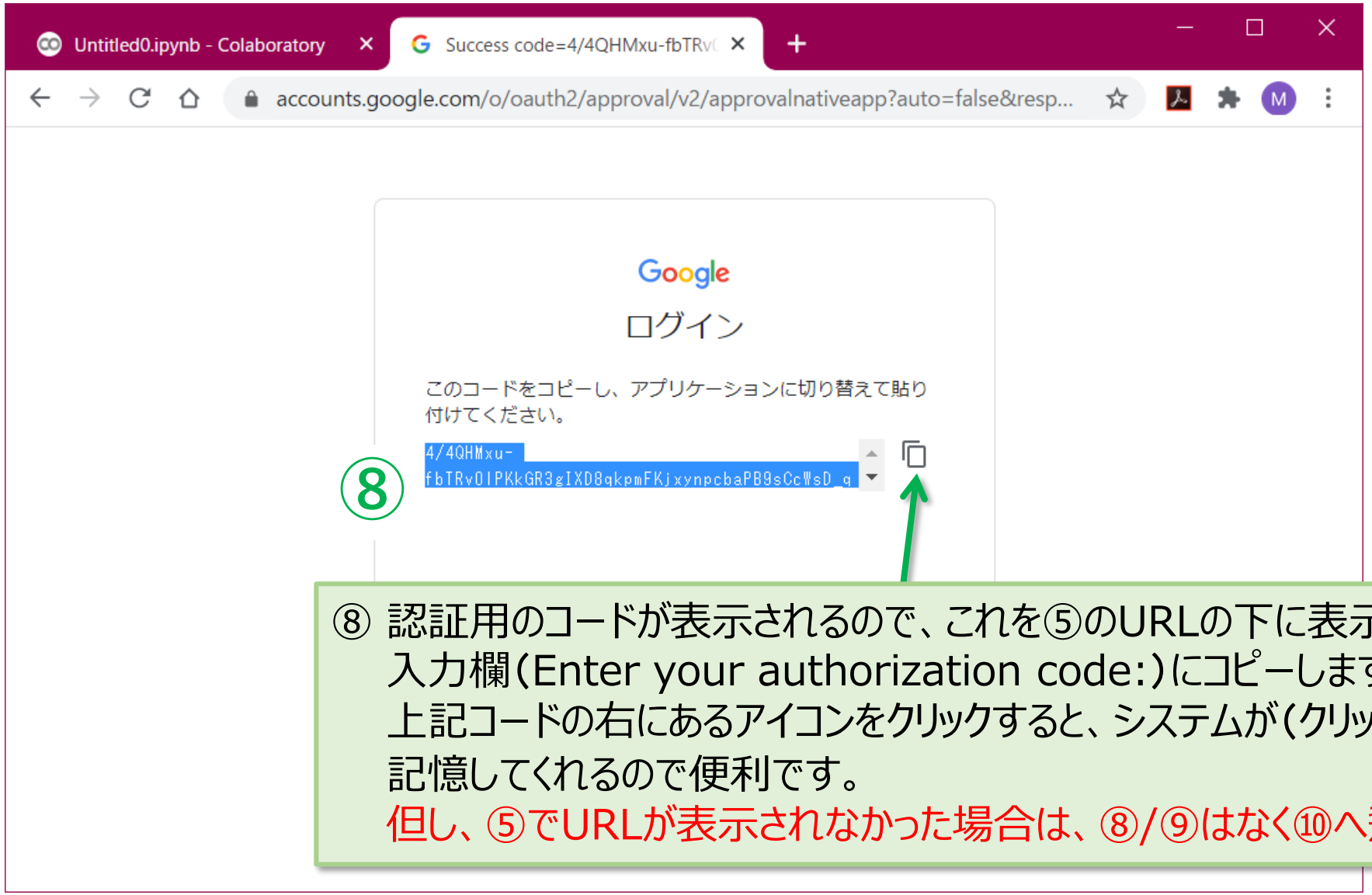


Google Drive のマウント（7）



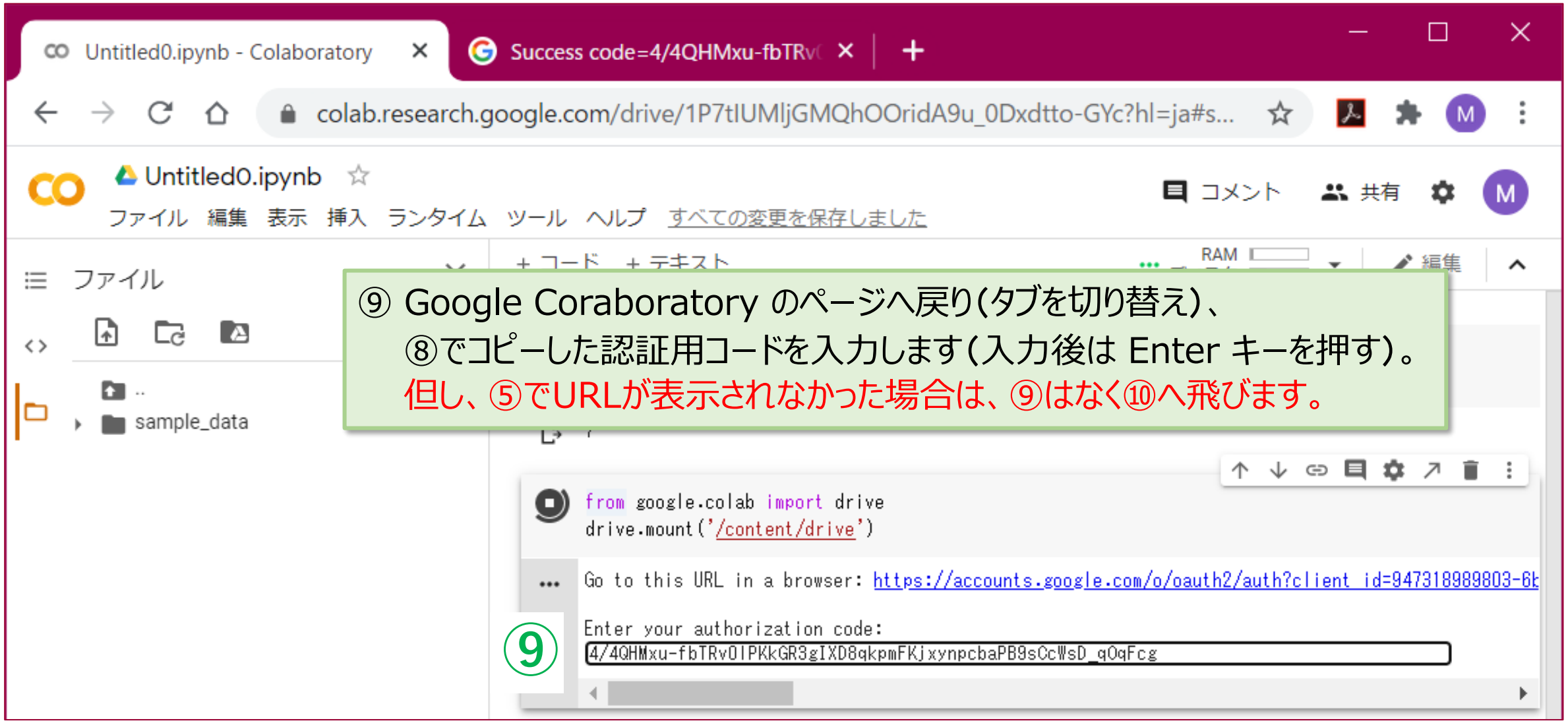
⑦ 様々な権限を求められますが、
取り敢えずこれらを全て許可しておきます。

Google Drive のマウント（8）



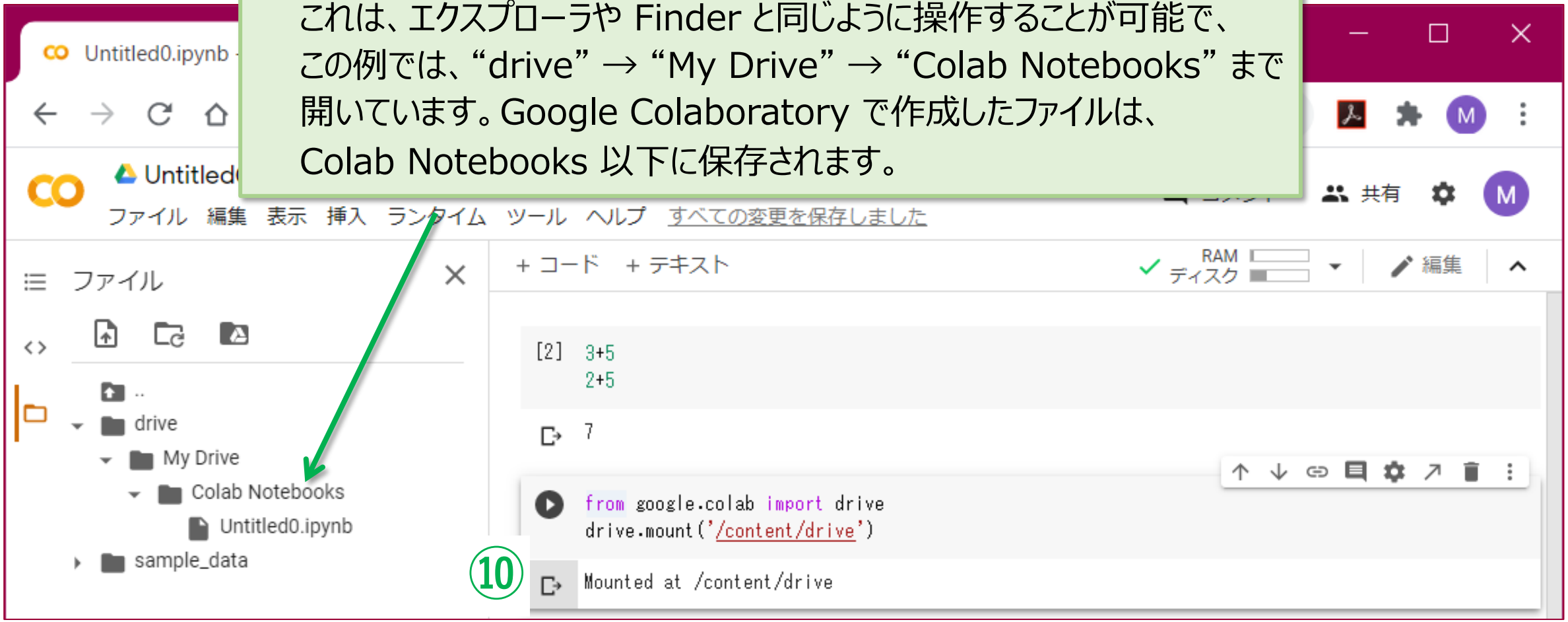
⑧ 認証用のコードが表示されるので、これを⑤のURLの下に表示された入力欄(Enter your authorization code:)にコピーします。
上記コードの右にあるアイコンをクリックすると、システムが(クリップ ボードに)記憶してくれるので便利です。
但し、⑤でURLが表示されなかった場合は、⑧/⑨はなく⑩へ飛びます。

Google Drive のマウント（9）



Google Drive のマウント（10）

⑩ 認証に成功すると、/content/drive にマウントしたと表示されます。
左側にあるファイル システム ビューに、“drive”が追加されています。
これは、エクスプローラや Finder と同じように操作することが可能で、
この例では、“drive” → “My Drive” → “Colab Notebooks” まで
開いています。Google Colaboratory で作成したファイルは、
Colab Notebooks 以下に保存されます。



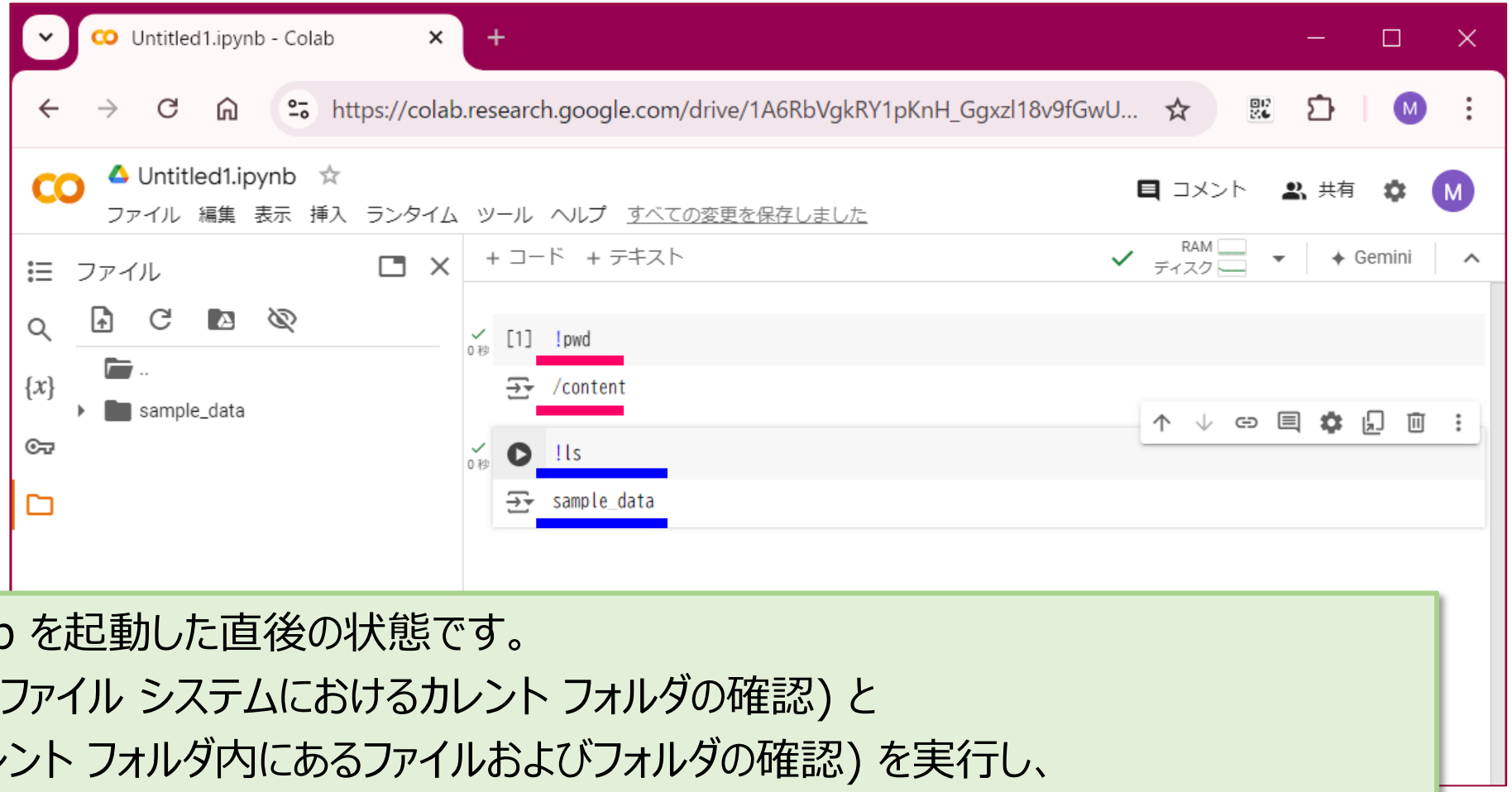
【重要】データ ファイルへのアクセス（1）

- 前のスライドでも述べましたが、ファイル システムやファイルのパス名を理解できていないと、プログラムを作成して実行し、その結果を利用することがほとんどできません。

- パス名の利用に慣れましょう → これができないと今後何もできない…

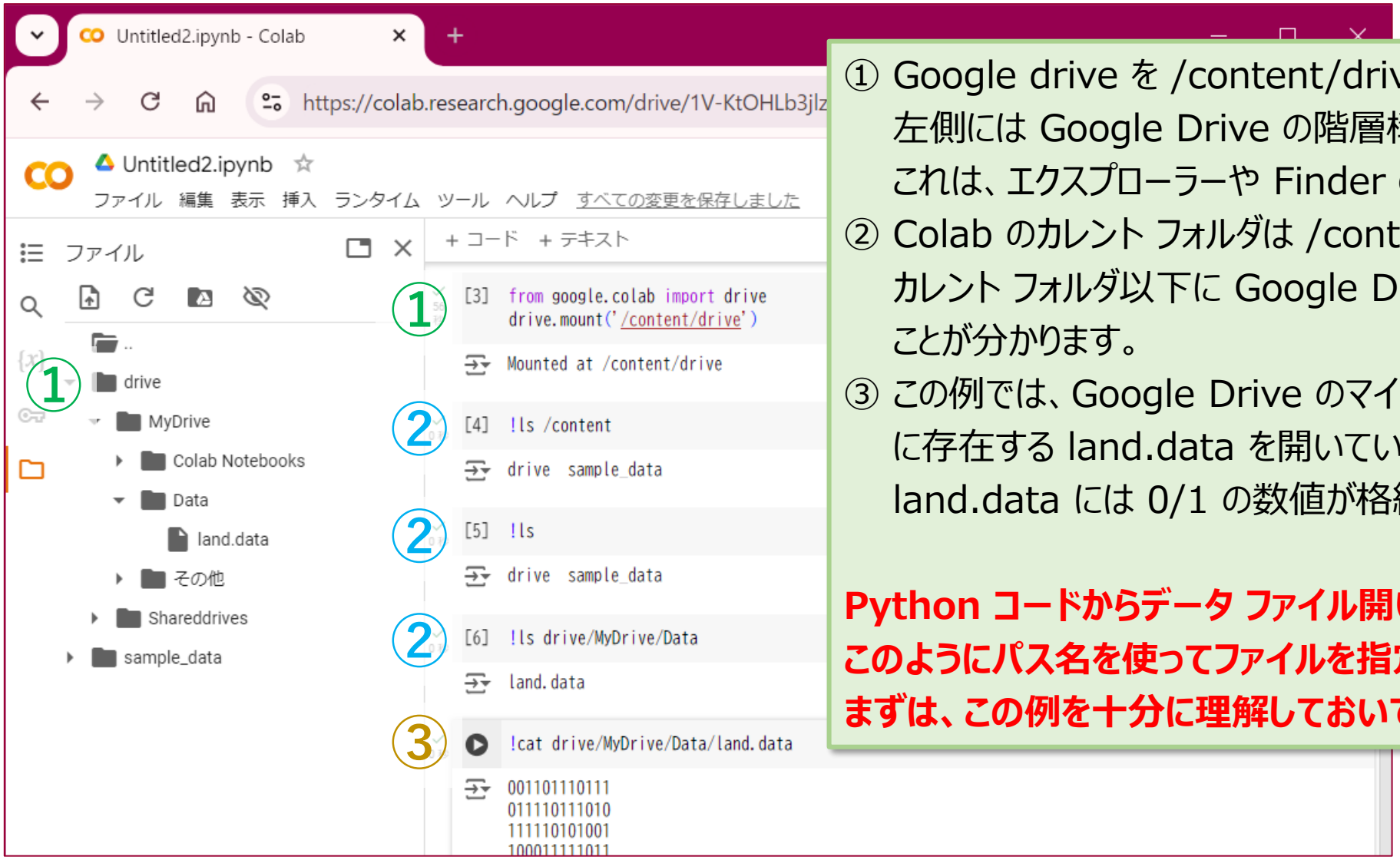
- これは、本講義だけの問題ではありません。

【重要】データ ファイルへのアクセス（2）



Google Colab を起動した直後の状態です。
pwd コマンド (ファイル システムにおけるカレント フォルダの確認) と
ls コマンド (カレント フォルダ内にあるファイルおよびフォルダの確認) を実行し、
Google Colab のカレント フォルダとその中身を確認します。
カレント フォルダは "/content" で、/content 内には sampe_data フォルダが存在しています。

【重要】データ ファイルへのアクセス（3）



① Google drive を /content/drive にマウントした状態です。
左側には Google Drive の階層構造が表示されています。
これは、エクスプローラーや Finder のように操作できます。

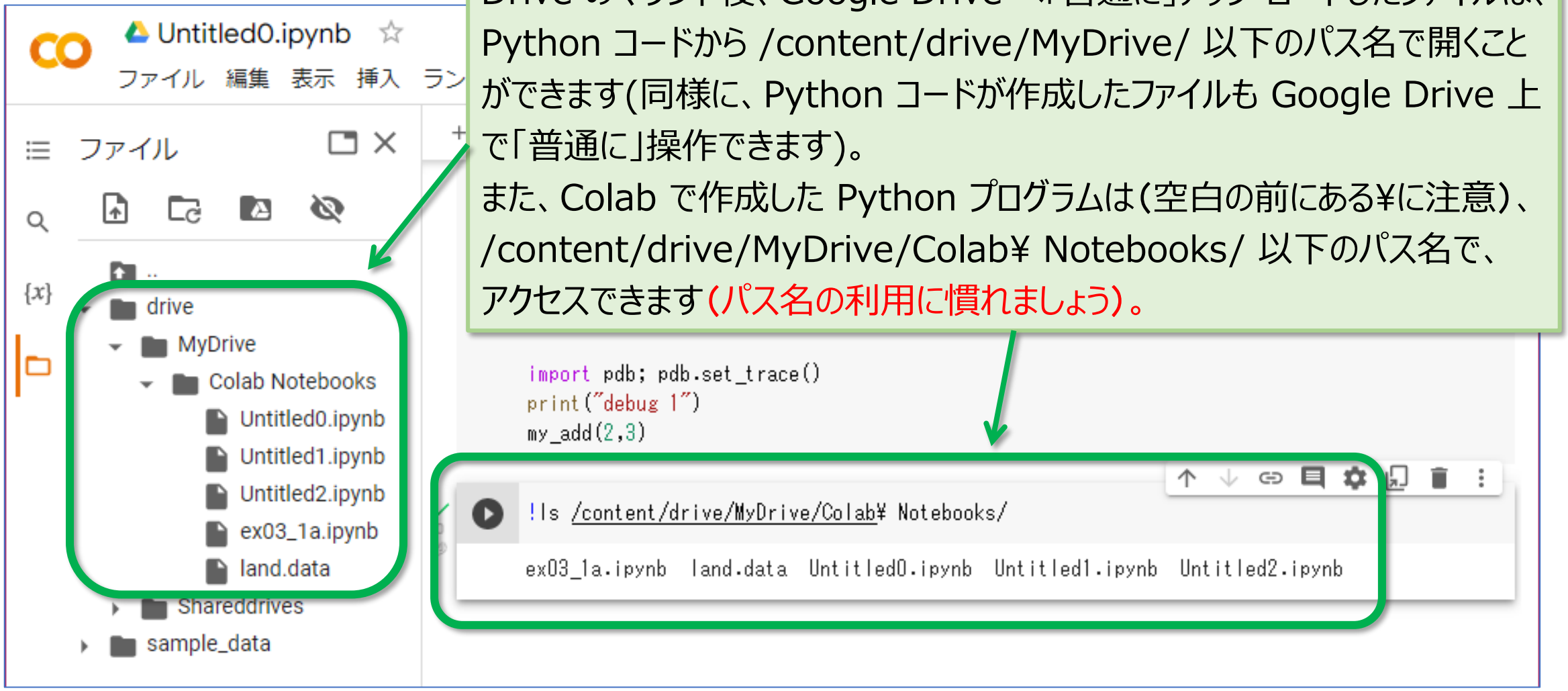
② Colab のカレント フォルダは /content なので、
カレント フォルダ以下に Google Drive がマウントされている
ことが分かります。

③ この例では、Google Drive のマイドライブにある Data フォルダ
に存在する land.data を開いています。
land.data には 0/1 の数値が格納されています。

Python コードからデータ ファイル開いたり結果を保存するには、
このようにパス名を使ってファイルを指定します。
まずは、この例を十分に理解しておいて下さい。

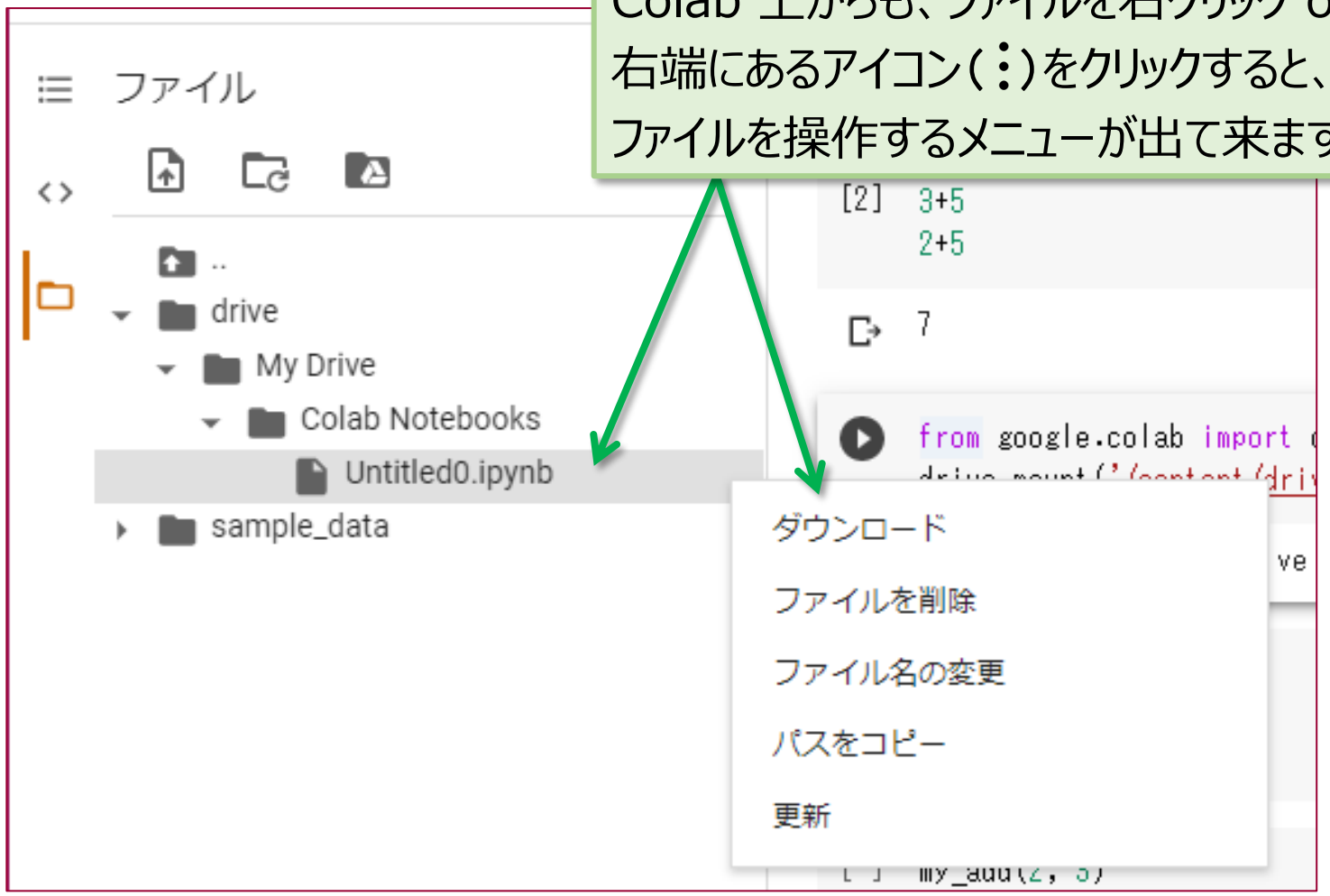
【重要】データ ファイルへのアクセス（4）

Drive のマウント後、Google Drive へ「普通に」アップ ロードしたファイルは、Python コードから /content/drive/MyDrive/ 以下のパス名で開くことができます(同様に、Python コードが作成したファイルも Google Drive 上で「普通に」操作できます)。
また、Colab で作成した Python プログラムは(空白の前にある¥に注意)、/content/drive/MyDrive/Colab¥ Notebooks/ 以下のパス名で、アクセスできます(パス名の利用に慣れましょう)。

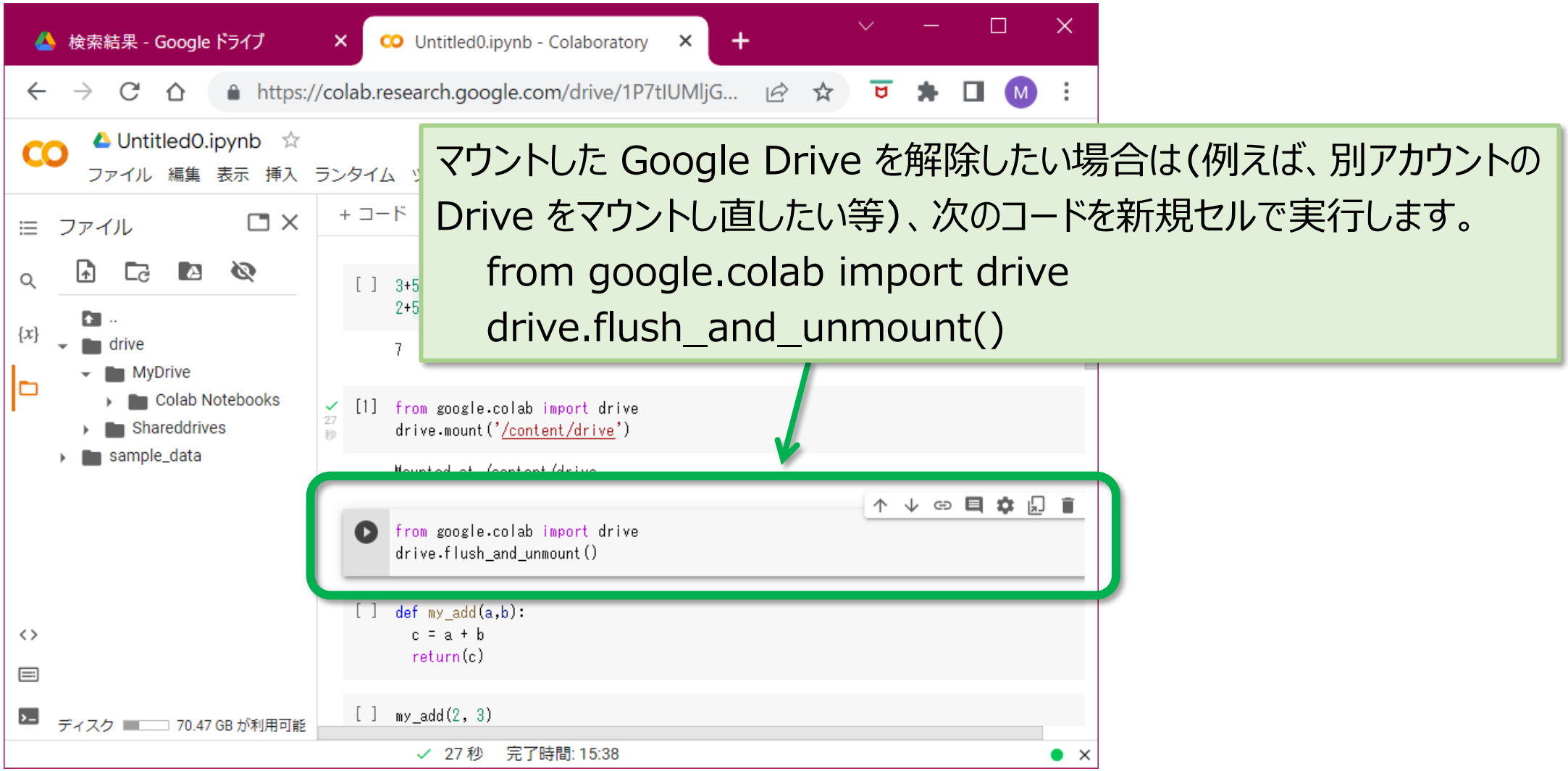


【重要】データ ファイルへのアクセス（5）

Colab 上からも、ファイルを右クリック or ファイル名の
右端にあるアイコン(⋮)をクリックすると、Google Drive の
ファイルを操作するメニューが出て来ます。



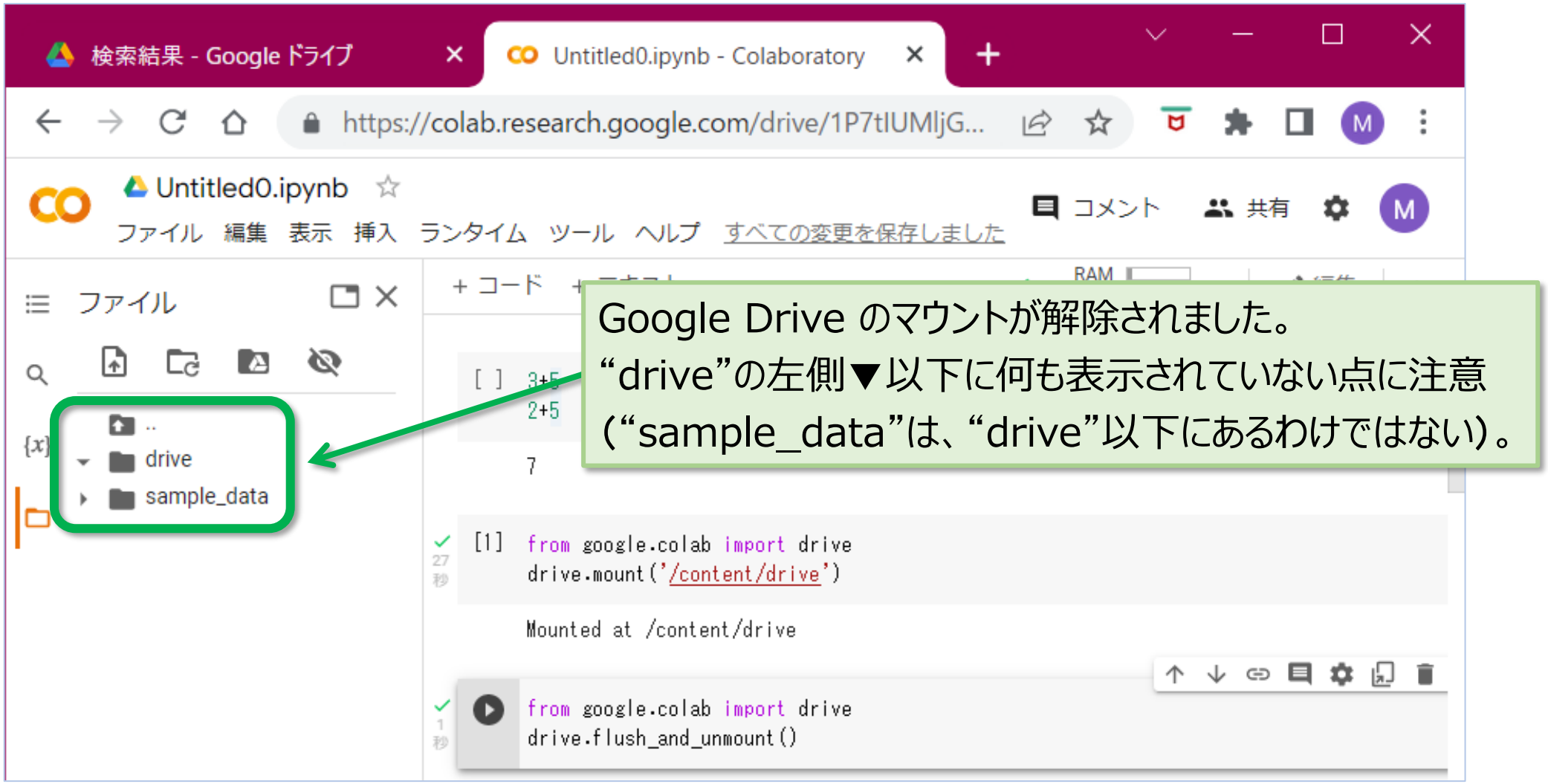
Google Drive のマウントを解除（1）



マウントした Google Drive を解除したい場合は（例えば、別アカウントの Drive をマウントし直したい等）、次のコードを新規セルで実行します。

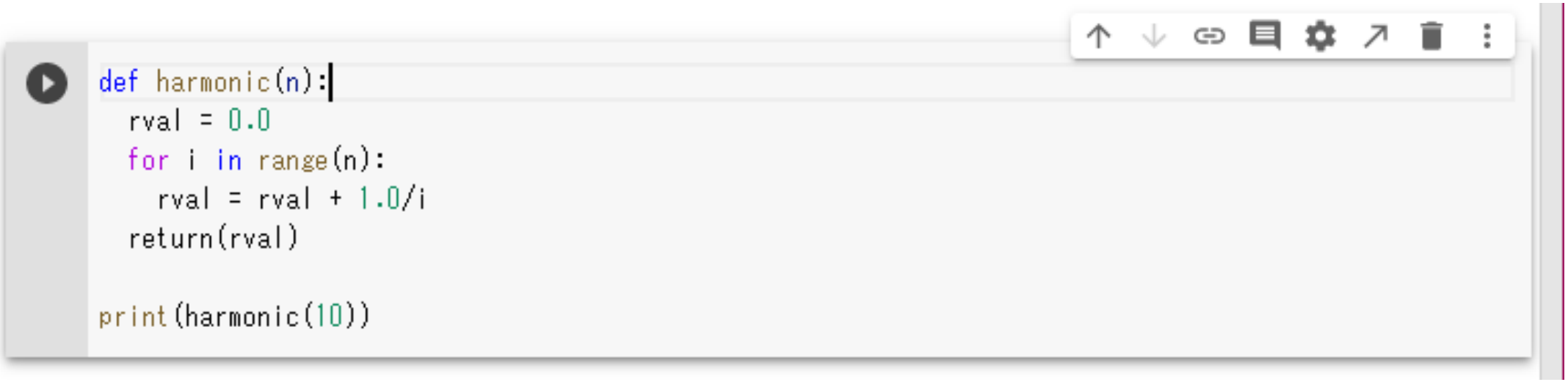
```
from google.colab import drive
drive.flush_and_unmount()
```

Google Drive のマウントを解除（2）



困った時の対応

エラー メッセージを確認しよう（1）



```
def harmonic(n):
  rval = 0.0
  for i in range(n):
    rval = rval + 1.0/i
  return(rval)

print(harmonic(10))
```

例えば、上のようなプログラムを作成したとします。各コードの意味は、講義が進むにつれ説明をしますが、取り敢えず上のプログラムでは、次の漸化式を計算すると考えて下さい。

$$a_{n+1} = a_n + 1.0/n \quad (a_0 = 0,\ n = 0, 1, \ldots, 9)$$

この場合、$n = 0$ の時に 0 による割り算が発生し、エラーとなることが予想されます。

実際に動かしてみましょう。

エラー メッセージを確認しよう（2）

```
def harmonic(n):
  rval = 0.0
  for i in range(n):
    rval = rval + 1.0/i
  return(rval)

print(harmonic(10))
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-3-f4409d6c49f5> in <module>()
      5   return(rval)
      6
----> 7 print(harmonic(10))

<ipython-input-3-f4409d6c49f5> in harmonic(n)
      2   rval = 0.0
      3   for i in range(n):
----> 4     rval = rval + 1.0/i
      5   return(rval)
      6

ZeroDivisionError: float division by zero
```

SEARCH STACK OVERFLOW

ZeroDivisionError が発生したと表示されました。

左の表示にある ----> の部分に注目して下さい。

これらの表示は上から見て行きます。意味は下記の通りです。

- まずは 7 行目にある print(harmonic(10)) を実行した。
- そこから harmonic 関数へ入り、4 行目の rval = rval + 1.0/i で止まった。

その下に、止まった理由を表示しています。

ZeroDivisionError: float division by zero

講義が進むにつれ説明をしますが、float は実数を意味します。

これらのメッセージを参考にして、プログラムを修正しましょう。

エラー メッセージを確認しよう（3）

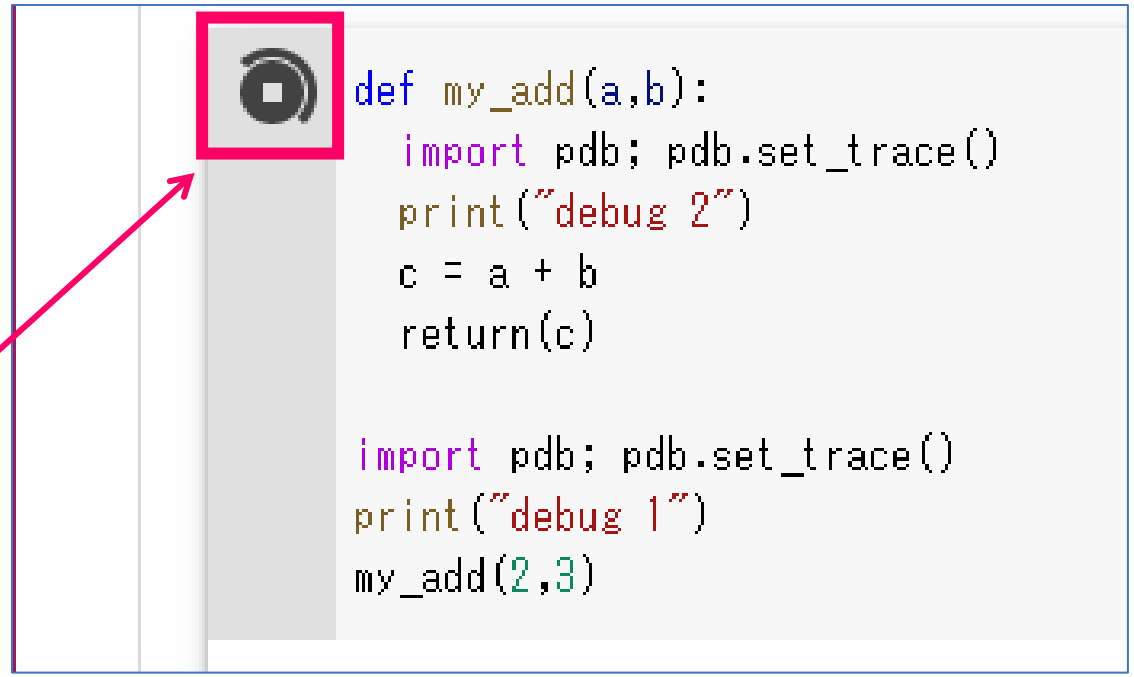
- 単純な誤りの原因として多いのは、以下の種類です。
  - インデントの整合が取れていない。
  - 複合文の “:” を忘れている。
  - 計数ループの範囲が合っていない。
  - 配列の上限/下限を超えている。

- 特に最後の二つについては、以降のスライド（デバッグの技法）を参考に、誤りを修正できるようにしましょう。

暴走してしまったら（1）

プログラムを実行したものの、いつまで経っても
結果が出ない/終了しない場合は、
プログラムが暴走している可能性があります。
このような場合は、プログラムを強制終了し、
後述する方法(デバッグの技法)を用いて
プログラムを修正します。

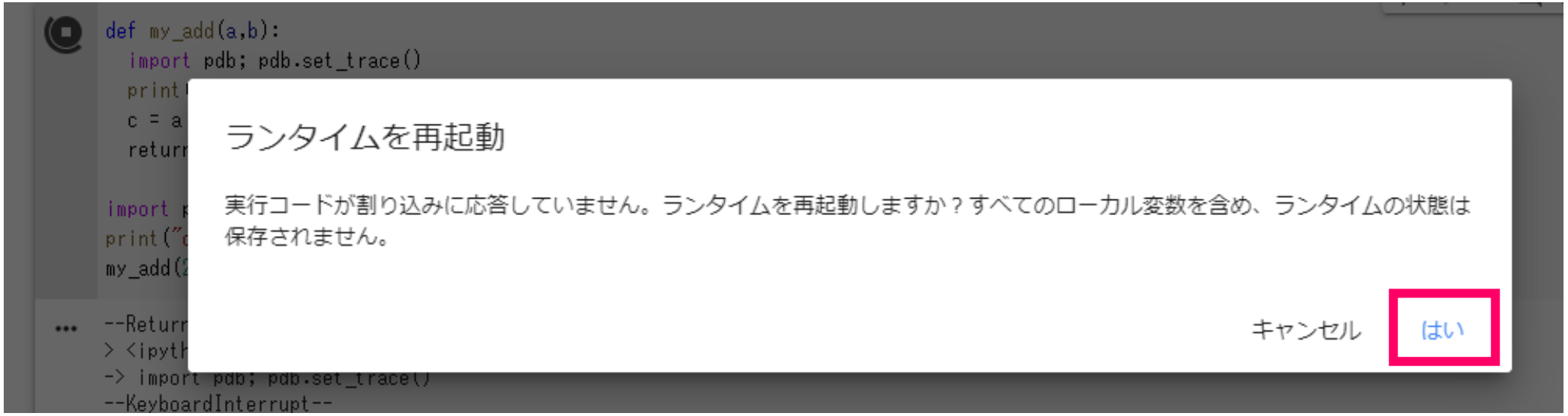
プログラムを強制終了するには、このアイコンを
クリックします(実行前の形と実行中の形との
違いに注意して下さい)。



強制終了されない場合は、何回かクリックして
みて下さい(どうしても終了しない場合は、
“ランタイム” タブより “実行を中断” を選択)。

暴走してしまったら（2）

プログラムの強制停止を行なうと、以下のようなメッセージが表示されることがあります。



ランタイムとは、Google Colaboratory 上で Python プログラムの実行を行なう存在です。
例えば、プログラムの実行箇所や、内部で利用するデータなどを管理します（これらは一例です）。
上記メッセージの意味は、プログラムの強制停止の要求に対してランタイムが応答しないので、
プログラムではなく、（と言うか、プログラムごと）ランタイムを強制再起動するという意味です。
まあ、こうなってしまったら仕方がないので、最終手段として「はい」を選びましょう。

デバッグの技法（1）

```
def harmonic(n):
  rval = 0.0
  for i in range(n):
    print(i)
    rval = rval + 1.0/i
  return(rval)

print(harmonic(10))
```

```
0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback
<ipython-input-4-11a967b1c677> in <module>()
      6   return(rval)
      7
----> 8 print(harmonic(10))

<ipython-input-4-11a967b1c677> in harmonic(n)
      3   for i in range(n):
      4     print(i)
----> 5     rval = rval + 1.0/i
      6   return(rval)
      7

ZeroDivisionError: float division by zero
```

SEARCH STACK OVERFLOW

プログラムのエラーを修正することをデバッグと呼びます。デバッグを行なうには、まずはそのエラー原因を突き止める必要があります。エラー原因の多くは、変数に予期しない値が入ってしまうことです。そのため、怪しげな部分で変数の値を確認することは、デバッグの基本と言えます。

この例では、計算をする直前で、print 文により計算に使う i の値を表示させています (print 文のより高度な使い方は、講義が進むにつれ説明します)。

エラーが検出される前に、i の値(0)が表示されていることを確認して下さい。

この技法は、基本中の基本なので(達人もこの技法を使います)、皆さんも必ず習得して下さい。

デバッグの技法（2）

```
def my_add(a,b):
  c = a + b
  return(c)

my_add(2,3)
```

次は、左のようなプログラムを考えてみます。このプログラムは、足し算をする関数 my_add を定義し、それを外部から my_add(2, 3) として呼び出しています（つまり、2 + 3 を計算しているわけです）。
my_add(2, 3) は、関数定義の外側にある点に注意して下さい。

```
def my_add(a,b):
  import pdb; pdb.set_trace()
  print("debug 2")
  c = a + b
  return(c)

import pdb; pdb.set_trace()
print("debug 1")
my_add(2,3)
```

このプログラムに対して、以下のコードを入れます（print 文は場所の確認用です）。
　import pdb; pdb.set_trace()
その後、このセルを実行すると、上記コードを入れた場所で実行が停止します。この技法は、停止時点での変数を確認するだけではなく、停止した場所から 1 行ずつ再開できるので、自分が意図した順番でコードが実行されているかどうかも確認できます（勿論、1 行進めた後に変数を確認することもできます）。但し現時点では、上記のコードを入れる場所は、関数外（左図で言えば print("debug 1")）よりも関数内（同、 print("debug 2")）の方が、混乱は少ないです（慣れてしまえば、どこでも可です）。ちょっと実行してみましょう。

デバッグの技法（3）

```
--Return--
> <ipython-input-7-4879edcff19c>(7)<module>
-> import pdb; pdb.set_trace()
(Pdb) l
  2         import pdb; pdb.set_trace()
  3         print("debug 2")
  4         c = a + b
  5         return(c)
  6
  7  -> import pdb; pdb.set_trace()
  8     print("debug 1")
  9     my_add(2,3)
[EOF]
(Pdb) n
> /usr/local/lib/python3.6/dist-packages/IPython/core/in
-> sys.excepthook = old_excepthook
(Pdb) l
2880                     self.hooks.pre_run_code_hook()
2881                     #rprint('Running code', repr(cod
2882                     exec(code_obj, self.user_global
2883                 finally:
2884                     # Reset our crash handler in pla
2885 ->                  sys.excepthook = old_excepthook
2886             except SystemExit as e:
2887                 if result is not None:
2888                     result.error_in_exec = e
2889                 self.showtraceback(exception_only=True)
2890                 warn("To exit: use 'exit', 'quit', or Ctrl-D.", stacklevel=1)
(Pdb)
```

前スライドのプログラムを実行した結果を表示します。
実行が停止したので、l コマンドで停止場所を確認しました。
確かに、import pdb; pdb.set_trace() の部分で止まっていますね。
但し、これは、関数外（左図で言えば print("debug 1")）にある
import pdb; pdb.set_trace() であることに注意して下さい。

次に、n コマンドで 1 行実行し、l コマンドで場所を確認してみました。
なんと、予想も付かない場所に行ってしまったようです。
これは、プログラムの実行を関数外で止めてしまったため、
利用者と対話的にやり取りするコードの中へ入ってしまったようです。
関数外では、Python の処理系は、利用者から入力があるかどうかを
確認しながら処理を進めるため、このようなコードが介在します。

デバッグの技法（4）

```
def my_add(a,b):
  import pdb; pdb.set_trace()
  print("debug 2")
  c = a + b
  return(c)

# import pdb; pdb.set_trace()
# print("debug 1")
my_add(2,3)
```

```
> <ipython-input-2-f13f7690921f>(3)my_add()
-> print("debug 2")
(Pdb) l
  1     def my_add(a,b):
  2         import pdb; pdb.set_trace()
  3  ->     print("debug 2")
  4         c = a + b
  5         return(c)
  6
  7     # import pdb; pdb.set_trace()
  8     # print("debug 1")
  9     my_add(2,3)
[EOF]
(Pdb) n
debug 2
> <ipython-input-2-f13f7690921f>(4)my_add()
-> c = a + b
(Pdb) print(a, b)
2 3
(Pdb) c
5
```

関数外にある import pdb; pdb.set_trace() を無効化して（先頭に # を付ける）再実行してみました。
l コマンドより、関数内の import pdb; pdb.set_trace() で止まっていることが分かります。

続いて、n コマンドで 1 行実行すると、予期した通り c = a + b の行に移動したことが分かります。この時、print(a, b) コマンドより、a と b には 2 と 3 が入っていることが確認できました。
その後、c コマンドで、次の import pdb; pdb.set_trace() に出会うまで実行させます。今回は、 import pdb; pdb.set_trace() に一度しか出会わない作りのため、プログラムが最後まで実行されて結果(5)が表示されました。講義が進むにつれ説明しますが、プログラム内に繰り返しの構文が入っている場合、例えば my_add 関数は複数回呼び出される可能性があり、 同じ import pdb; pdb.set_trace() に何度も出会うことがある点に注意して下さい。