

コンピュータリテラシ発展 ～Pythonを学ぶ～

第 2 回：Pythonを始めよう

情報学部 情報学科 情報メディア専攻

清水 哲也 (shimizu@info.shonan-it.ac.jp)

今回の授業内容

今回の授業内容

- Pythonの特徴
- [Google Colaboratory](#) の使い方
- Pythonのきほん
- ある条件で処理を分ける
- オブジェクトの扱い
- 課題

Pythonの特徴

Pythonの特徴

Pythonは世界的に人気のあるプログラミング言語であり、その特徴は以下の通りです.

1. シンプルで読みやすい構文: Pythonはコードが読みやすく設計されています.

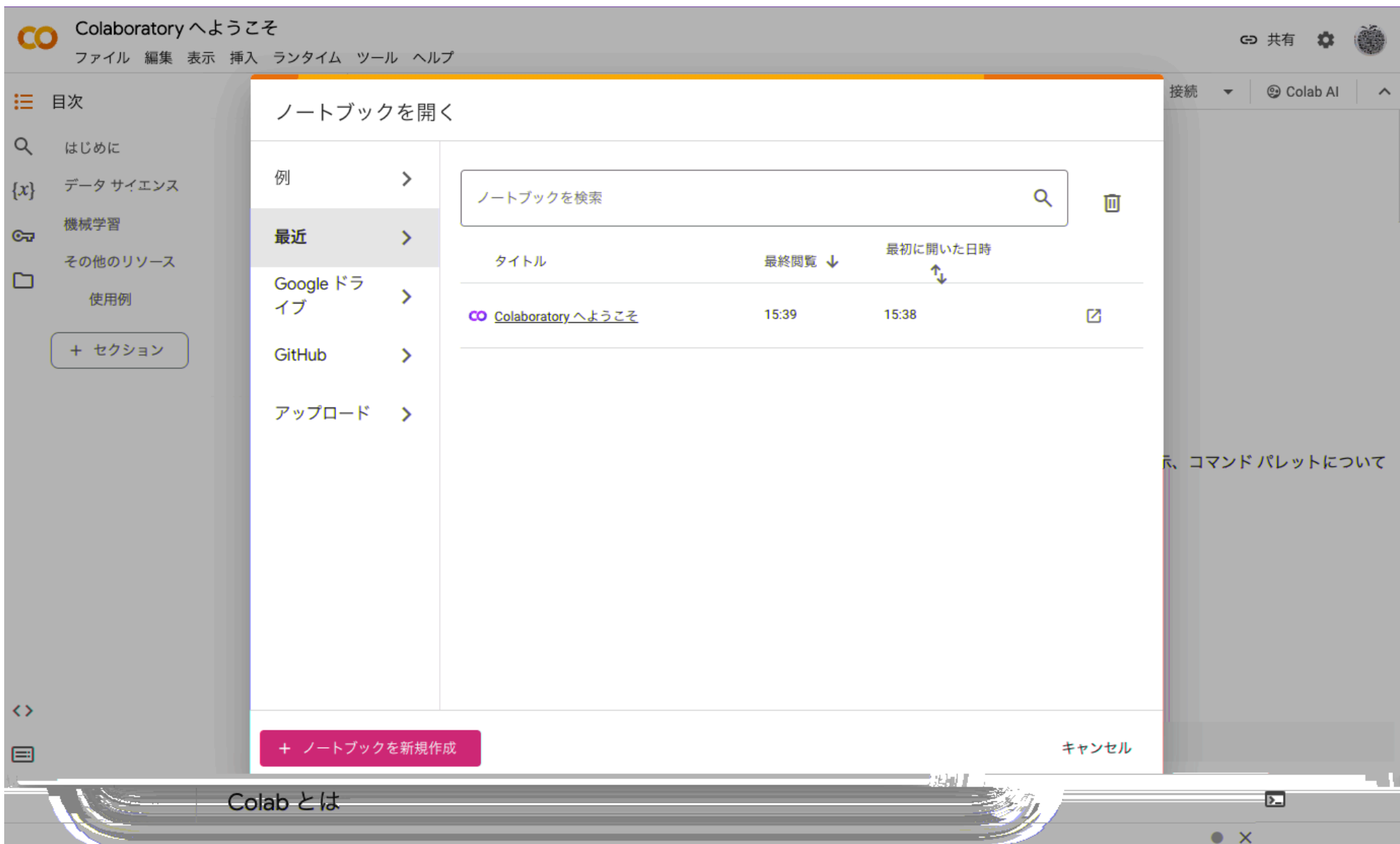
グラフィカル
用するための
インタプリタ型
これにより、開発

Google Colaboratory の使い方

Google Colaboratory の使い方

- Pythonの開発環境として[Google Colaboratory](#)を使います
- Google Colabはクラウドに用意された[Jupyter Notebook](#)環境でブラウザさえあれば無料で利用可能です
- 大学のGoogleアカウント（@sit.shonan-it.ac.jp）でログインしておけば使用したファイルはGoogle Drive上に保存されます

ノートブックの作成



Colaboratory へようこそ

共有 設定

接続 Colab AI

ノートブックを開く

ノートブックを検索

例	最終閲覧 ↓	最初に開いた日時 ↑
最近		
Google ドライブ		
Colaboratory へようこそ	15:39	15:38
GitHub		
アップロード		

+ ノートブックを新規作成

キャンセル

Colab とは

ノートブックの作成

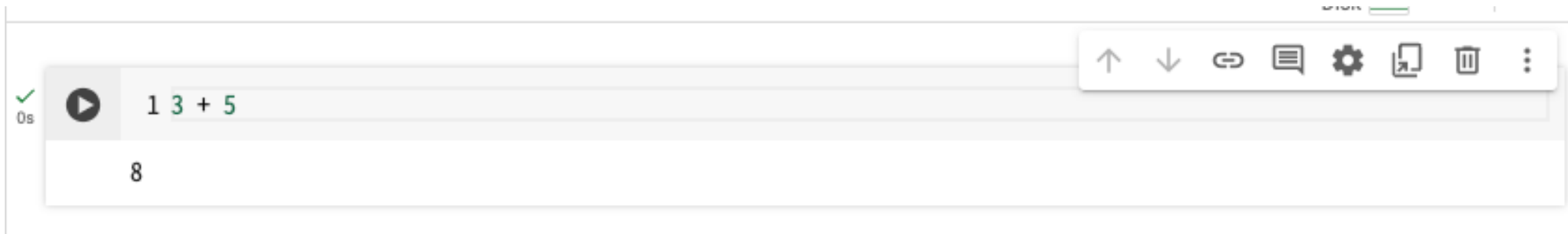
The screenshot shows the Google Colab interface. At the top, the title bar displays 'Untitled0.ipynb' with a star icon. Below it, a menu bar includes 'ファイル' (File), '編集' (Edit), '表示' (View), '挿入' (Insert), 'ランタイム' (Runtime), 'ツール' (Tools), and 'ヘルプ' (Help), followed by the message 'すべての変更を保存しました' (All changes saved). On the right side of the title bar are icons for 'コメント' (Comments), '共有' (Share), and '設定' (Settings).

Below the title bar, there are buttons for '+ コード' (Code) and '+ テキスト' (Text). The main workspace area contains a large gray box with a play button icon and the text 'コーディングを開始するか、AI で生成します。' (Start coding or generate with AI). This area is highlighted with a red border. A red arrow points from the text 'これがファイル名です' (This is the file name) to the title bar. Another red arrow points from the text 'ここを「セル」と呼びます' (Call this a 'cell') to the gray box. The bottom right corner of the interface shows a close button (X) and a refresh button (circular arrow).

Colabを試してみる

プログラムを書いて実行してみる

1. セルに「`3 + 5`」と入力します
2. セルの左側にある「三角ボタン」をクリックします（初回は少し時間がかかります）
3. 計算結果が表示されればOK



Google Colabの使い方

基本的には以下の繰り返しです

- 「セル」にプログラムを入力する
- 「三角ボタン」を押す
- 結果を得る

Pythonのきほん

データの性質

- 扱うデータの種類によってデータ型が異なります

データの性質

データ型	概要	例
<code>str</code>	文字列	<code>'abc'</code> , <code>"hello world"</code>
<code>int</code>	整数	<code>1</code> , <code>-500</code>
<code>float</code>	浮動小数点	<code>1.23</code>
<code>bool</code>	真偽値	<code>True</code> もしくは <code>False</code>
<code>list</code>	リスト	<code>[1, 2, 3]</code>
<code>tuple</code>	タプル	<code>(1,2,3)</code> , <code>1,2,3</code>
<code>dict</code>	辞書	<code>{'a':1, 'b':2}</code>

オブジェクトと関数

オブジェクト

- オブジェクトは「データ型」や「値」といった要素で構成されます
- 「データ型」はデータを性質に応じて分類したものです
- 「値」はデータそのもののことです
- 例： `'hello'` というオブジェクトは、データ型がstr型で、値がhelloとなります
- 参考： <https://docs.python.org/ja/3/reference/datamodel.html>

関数

- 入力値を与えると何らかの処理を実行し出力値を返すものです
- 例： `print()` 関数

Pythonでの計算

参考：算術演算子（教科書:表2-2）（Colabの場合 `print()` がなくてもOK）

+：加算（足し算）

足し算を行うときには演算子「**+**」を利用します.

```
print(10 + 30)
```

-：減算（引き算）

引き算を行うときには演算子「**-**」を利用します.

```
print(10 - 30)
```


Pythonでの計算

* : 乗算 (掛け算)

掛け算を行うときには演算子「*」を利用します。「x」ではないので注意してください。

```
print(10 * 30)
```

** : べき乗

べき乗 (n 乗, a^n) の計算を行うときには演算子「**」を利用します。

```
print(10 ** 3)
```

Pythonでの計算

/ : 除算 (割り算)

割り算を行うときには演算子「/」を利用します。「÷」ではないので注意してください。

```
print(10 / 3)
```

// : 除算 (小数点以下切り捨て)

小数点以下を切り捨てた割り算を行うときには演算子「//」を利用します。割り算の整数の商を求めます。

```
print(10 // 3)
```

Pythonでの計算

% : 剰余 (除算した余り)

割り算の余りを求める場合は演算子「%」を利用します.

```
print(10 % 3)
```

divmod 関数 : 除算&剰余

Pythonには割り算を行ったときの小数点以下を切り捨てた答え (割り算の整数の商) とその余りを同時に求める関数が用意されています.

```
print(divmod(10, 3))
```

数値演算子の優先順位

数値の演算には優先順位があります.

演算子による優先順位

演算子による優先順位の考え方は通常の数学と同じです.

```
print(10 + 5 * 2)
```

() 括弧による優先順位の変更

一般的な数学と同じで () 括弧でくくることで演算子の演算優先順位を任意に変更することができます.

```
print((10 + 5) * 2)
```

異なるデータ型同士の計算

計算可能なのは整数型だけではなく文字列も計算可能です

文字列同士を結合したり，同じ文字列を繰り返し表示したりすることも可能です。

文字列を結合する場合

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

同じ文字列を繰り返す場合

```
print('Hello' * 5)
```

何でもできるわけではないです。

```
print('1' + 2)
```

基本計算

2)

-と

オブジェクトを操作する

文字列オブジェクト内の文字をすべて大文字にする.

```
print('hello world!'.upper())
```

文字列オブジェクトは「hello world!」でそのオブジェクトを操作する命令が「upper()」です.

「upper()」をメソッドといいます. メソッドは, データ型と密接に関係しています. なので, 整数型のオブジェクトはupper()メソッドを実行できせ

同じオブジェクトを使いまわす

- 同じオブジェクトを何度も使う場合はオブジェクトに名前をつけます
- オブジェクトの名前 = 「変数」

```
hi = 'hello'  
print(hi)
```

一度実行しましょう

```
hi = 'world'  
print(hi)
```


ある条件で処理を分ける

条件を判定する

条件分岐で条件を判定するためには、次のような **比較演算子** を使います

記号	意味
<	左辺が右辺より小さい場合にTrue
<=	左辺が右辺以下の場合にTrue
>	左辺が右辺より大きい場合にTrue
>=	左辺が右辺以上の場合にTrue
!=	左辺と右辺が一致しない場合にTrue
==	左辺と右辺が一致する場合にTrue

条件を判定する

- 比較演算子を使って判定してみる

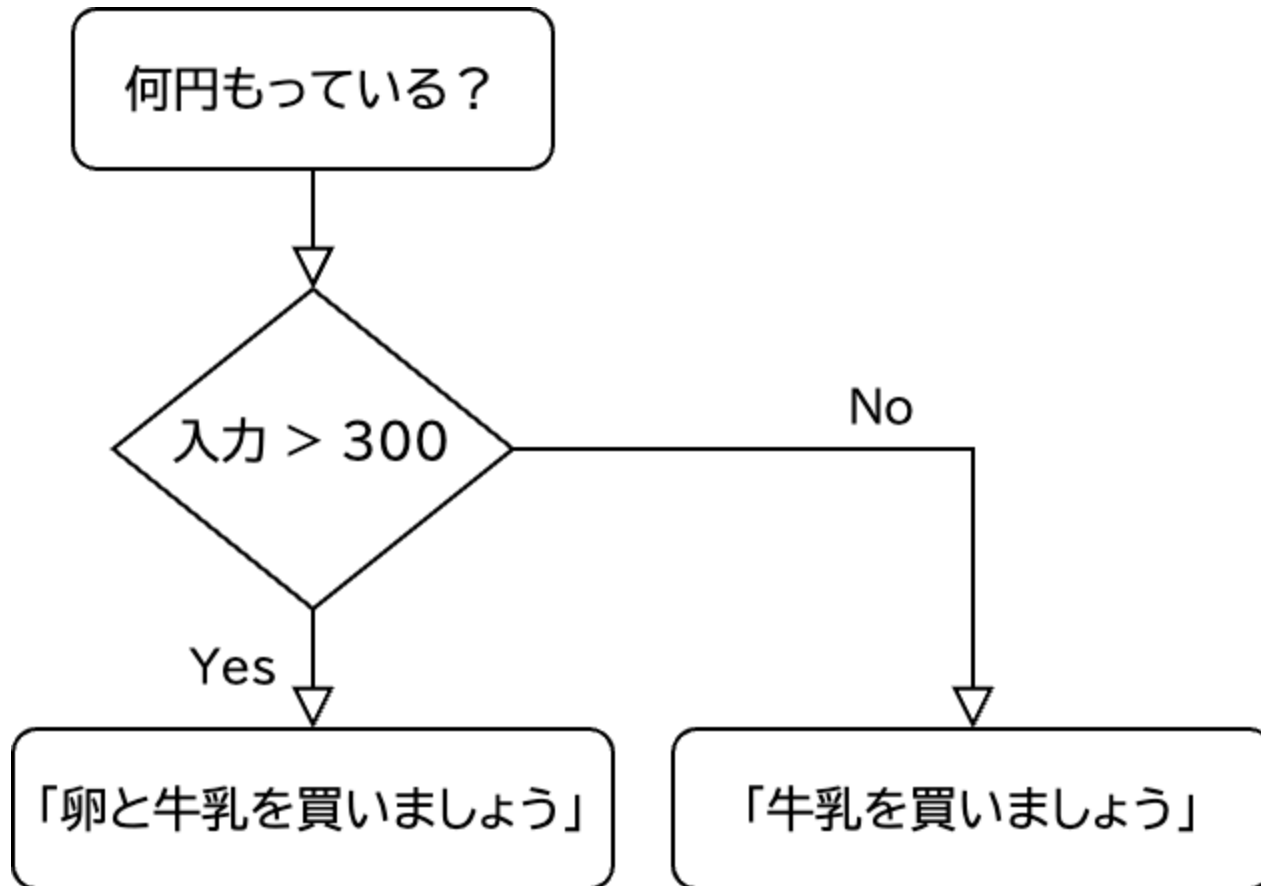
```
print(1 < 2)
```

```
print(1 > 2)
```

- 比較演算子の結果は「**True**」か「**False**」が返ってきます
- 「**True**」か「**False**」を真理値と呼びます

条件に応じて処理をする

簡単な条件分岐による処理をしてみましょう



条件に応じて処理をする

条件分岐を行うために必要 `if` 文

```
if 条件式:  
    条件式がTrueのときの処理  
else:  
    条件式に当てはまらなかったときの処理
```

「もし～ならば」という意味です

条件式には比較演算子などを使い判定をおこないます

「`else`」は「それ以外」という意味で条件式にあてはまらない場合の処理を書きます

Pythonでは「インデント(空白)」がとても大切です

条件に応じた処理をする

Colabの場合「インデント」は自動ではいりません

(インデントは一般的に半角スペース4つ分ですが変更も可能です)
適切な「インデント」をつけないとエラーとなります

```
if x<2:  
    print('xは、2より小さい')
```



インデント

(半角スペース4つ分)



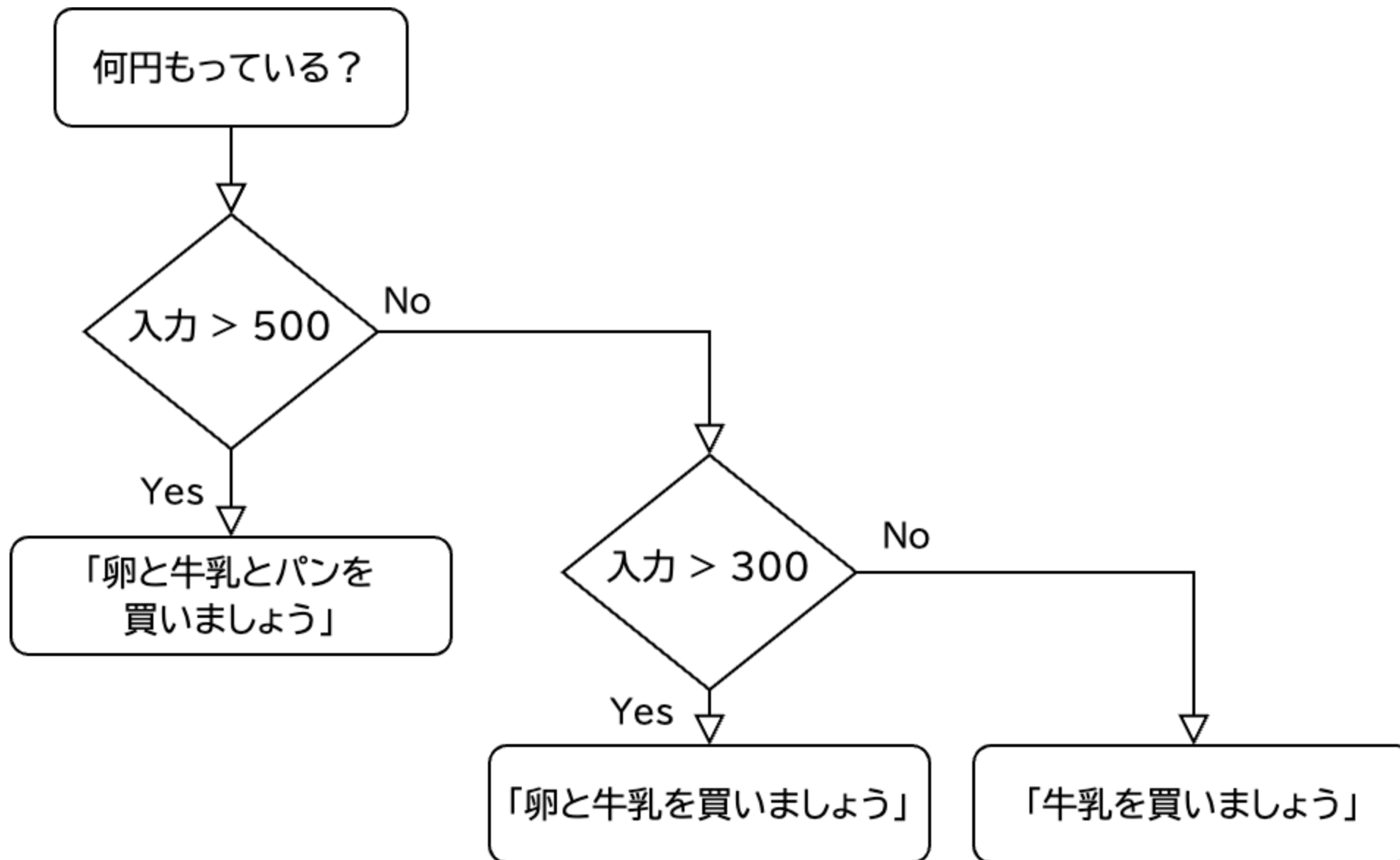
ブロック

条件に応じた処理をする

それでは、実際にP.28の図のプログラムを組み立ててみましょう

```
money = 400
if money > 300:
    print('卵と牛乳を買いましょう')
else:
    print('牛乳を買いましょう')
```

条件に応じた処理をする



条件に応じた処理をする

複数の条件式で判定を行う場合は「`elif`」(`else if` の略) を使って判定をおこないます.

```
if 条件式1
    処理1
elif 条件式2
    処理2
elif 条件式3
    処理3
:
else:
    その他処理
```

P.32の図のプログラムをしてみてください

オブジェクトの扱い

中身をあとから変更できるリスト型

- オブジェクトの集まりを扱うために「データ構造」という仕組みがあります
- その一つである「リスト(**list**)型」があります
- リスト型は「`[]`」で囲んで扱います

リスト型の例

```
[1]  
[1, 2, 3]  
['apple', 'orange', 'banana']
```

中身をあとから変更できるリスト型

- `[]` の中にオブジェクトを並べます
- オブジェクトが複数ある場合は「`,` (カンマ)」で区切ります
- リストの中身のオブジェクトを「要素」と呼びます
- 要素が何もなくてもOKです
- リストの中にリストを入れることもできます

リスト型の例

```
[1, 2, 3] # カンマで区切って複数のオブジェクトを扱う  
['apple', 'orange', 'banana'] # 文字列も同じように扱える  
[] # 要素が無い（空）のリストも作成できる  
[[1, 2], [3, 4]] # リストの中にリストをいれることもできる
```

リスト型のインデックス

- インデックスはリスト内の要素（オブジェクト）の位置を示します
- 「リスト型[インデックス]」という形で使います
- インデックスを使ってリスト内の要素を変更することもできます

インデックスの例

```
fruits = ['apple', 'orange', 'banana'] # リスト作成
print(fruits[0]) # 0番目の要素を表示
print(fruits[1]) # 1番目の要素を表示
fruits[1] = 'grape' # 1番目の要素を「grape」に変更
print(fruits) # リストの中身を確認
```

リスト型にオブジェクトを追加

- リストにオブジェクトを追加することもできます
- `append()` メソッドを使ってリストに新たにオブジェクトを付けることができます

`append()`の例

```
number = [1, 2, 3] # リスト作成
number.append(4) # 「number」リストに要素「4」を追加
print(number) # リストの中身を確認
```

リストからオブジェクトの取り出し

- リストからオブジェクトを取り出すときは「`pop()`」メソッドを使います
- 取り出したいオブジェクトのインデックスを指定して取り出します
- 取り出したあとはリストから取り出したオブジェクトはなくなります
- インデックスを指定しない場合は最後のオブジェクトが取り出されます

`pop()`の例

```
alphabet = ['A', 'B', 'C', 'D'] # リスト作成
char_C = alphabet.pop(2) # アルファベット「C」のインデックスを指定して取り出す
print(char_C) # 中身を確認
print(alphabet) # リストの中身を確認
char_last = alphabet.pop() # インデックスを指定しない場合最後のオブジェクトを取り出す
print(char_last) # 中身を確認
print(alphabet) # リストの中身を確認
```

リストの中のデータを確認する

- 「オブジェクト in リスト」の形でリストの中の要素が存在することを判定できます
- 「オブジェクト not in リスト」の形でリストの中に要素が存在しないことを判定できます

データ確認の例

```
Kanto = ['Tokyo', 'Kanagawa', 'Chiba', 'Saitama', 'Ibaraki', 'Tochigi', 'Gunma'] # リスト作成
if 'Gunma' in Kanto: # GunmaがKantoリスト内に存在するか判定
    print('Gunmaは関東地方です') # 処理
else: # それ以外
    print('Gunmaは関東地方ではありません') # 処理

if 'Yamanashi' not in Kanto: # YamanashiがKantoリスト内に存在しないか判定
    print('Yamanashiは関東地方ではありません') # 処理
else: # それ以外
    print('Yamanashiは関東地方です') # 処理
```


中身をあとから変更できないタプル型

- リスト型と同じくオブジェクトを複数格納することができ中の値を読み込むことができる
- リスト型との違いは、一度タプル型をつくと格納されている要素やその順番を一切変更することができない
- 「`()`」で囲むことでタプルをつくることができます

タプル型の例

```
() # 空のタプル型の生成
(1,2) # 複数のオブジェクトで構成されたタプル
(1,) # オブジェクトが1つの場合でもカンマをつける
1,2 # ()がなくてもタプル型になる
1, # ()がない場合でオブジェクトが1つの場合もカンマをつけることでタプル型になる
```

中身をあとから変更できないタプル型

- リスト型と同じように複数のオブジェクトを「`,`」で区切ります
- 1つの要素のときでも「`,`」をつければタプルとなります
- 「`()`」がなくても「`,`」で区切ればタプルとなります

タプル型の例

```
sample_tuple = (10, 20) # タプルの生成
x, y = sample_tuple # タプルの中身を展開してx, yに代入
print(x)
print(y)

sample_tuple2 = 'A', 'B', 'C' # タプルの生成
a, b, c = sample_tuple2 # タプルの中身を展開してx, yに代入
print(a)
print(b)
print(c)
```

キーと値をセットで扱う辞書型

- 辞書(dict)型も同じくオブジェクトを複数格納するためのものです

格納可能なオブジェクト (文字列, 数値, タプル)

辞書型はオブジェクト

特定のキーを指定してバリューを取り出します

{ }

辞書型は空の辞書型

「キー:バリュー」と表現します

複数のキーとバリューを格納する場合は「,」で区切ります

```
{ } # 空のdict型
{'tea' : 100} # キーとバリューを格納
{'tea' : 100, 'coffee' : 200} # 複数のキーとバリューを格納
```

辞書型でキーを指定する

- dict（辞書）型のバリュー（値）を参照するには「辞書名[キー]」で指定します
- キーを指定するとそれに対応するバリューが出力されます
- 辞書に新しいキー、バリューを追加するには「辞書名[追加キー]=追加バリュー」と書きます

dict型でキーの扱いの例

```
my_dict = {'tea' : 100, 'coffee' : 200} # dict型の作成
print(my_dict['tea']) # キーを指定して対応するバリューを表示

my_dict['milk'] = 300 # 新たにキーとバリューを追加
print(my_dict) # dict型の中身を確認
```

キーと値をセットで扱う辞書型

- 辞書の中のデータを確認したい場合
- 「期待するキー名 in 辞書名」で存在を確認できます
- リストの場合とほぼ同じです

セットで扱う例

```
my_dict = {'tea' : 100, 'coffee' : 200, 'milk' : 300} # dict型の作成  
'tea' in my_dict # my_dict内に'tea'があるか確認
```

課題

課題2-1

- Moodleにある「SCfCL-2nd-prac.ipynb」ファイルをダウンロードしてColabにアップロードしてください
- 課題が完了したら「File」>「Download」>「Download .ipynb」で「.ipynb」形式でダウンロードしてください
- ダウンロードした.ipynbファイルをMoodleに提出してください
- 提出期限は **4月25日(木) 20時まで** です