AIエンジニア研修

Python基礎

Pythonの基礎

AI開発に向けて、Pythonの基本を学ぶ

1. データ型と変数
2. 数値と文字列
    - 算術演算
    - 文字列操作
3. 条件分岐
    - if文
4. リスト
5. その他のシーケンスと繰り返し
    - range
    - tuple
    - for文
6. 辞書型
7. 処理の定義と利用
    - 関数
    - モジュール
8. クラウドAPIの利用
    - AI API体験（A3RT）

Python とは

Python

https://www.python.org/

- すばやく効果的にシステムを開発できるように作られた
汎用プログラミング言語
- スクリプト言語で開発しやすく、誰が書いても同じような書き方になる
- ライブラリが豊富で、AI・機械学習でよく使われている
- 2系と3系があり、互換性がないので注意
※ 基本的には3系の使用推薦

Colaboratory とは

Colaboratory（通称Colab）

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

- 機械学習/ディープラーニングの教育・研究促進を目的として
  Googleが提供するクラウドツール
- ブラウザ上から、無償でJupyter NotebookやGPUなどの
  機械学習環境が使用可能
- 機械学習の主要ライブラリがあらかじめインストールされている

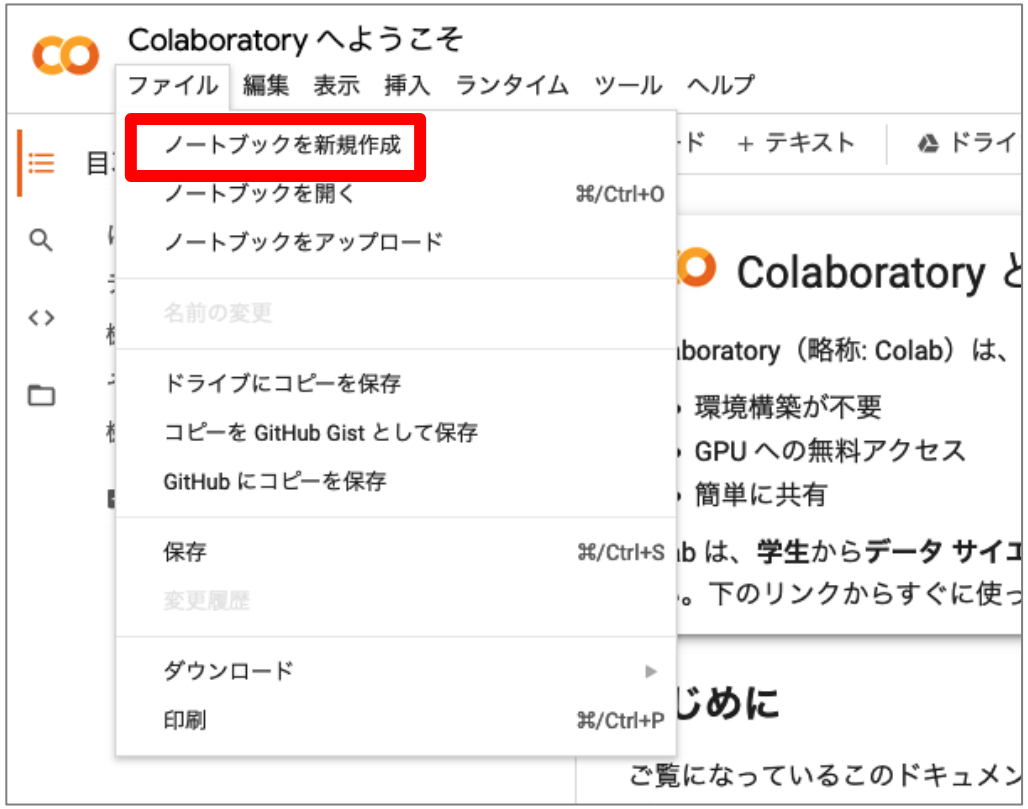
実行環境準備

実行環境準備 | Colabへのアクセス

1. Colabへアクセスする。
   https://colab.research.google.com/
   （Google等で「Colab」と検索してもよい）

実行環境準備 | ノートブックの作成

1. 「ファイル」をクリック
2. 「ノートブックを新規作成」をクリック



3. ファイル名を修正
   例：HelloWorld.ipynb

実行環境準備 | 動作確認

1. 以下を入力

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

2. 再生ボタンをクリック
   もしくは Shift + Enter

3. 以下が表示される

```
Hello World!
```

※作成したファイルはGoogle Drive マイドライブ上の「Colab Notebooks」フォルダ内に保存されます

Pythonの基本

- 1行ごとに実行内容（コード）を記述する。

```
print('Hello World!’)
x = 4
```

- ブロック（コードのまとまり）は
インデント（字下げ）で表す。

```
if x > 0:
  print('OK')
```

Pythonの基本記法と間違いやすいポイント

基本記法

1. セミコロンは不要

✖
```
x = 4;
y = 2;
print(x + y);
```

○
```
x = 4
y = 2
print(x + y)
```

2. 中括弧の代わりにインデントでブロックを管理
※ Colaboratory は空白スペース2つが基本

✖
```
if(x > 0) {
  print('OK');
}
```

○
```
if(x > 0):
  print('OK')
```

間違いやすいポイント

1. スペースが全角文字になっている
2. 閉じカッコが足りない
3. インデントがずれている

プログラムがうまく
動かない場合はチェック！

データ型と変数

データ型

Pythonではさまざまなデータの型（タイプ）が利用できる。

型	意味	例
int	整数	`1`
float	実数	`1.5`
str	文字列	`'Deep Learning'`
bool	真偽	`True, False`
list	配列	`[1, 2, 3]`
tuple	配列（書き換え不可能）	`(1, 2, 3)`

変数

データを格納する入れ物。

```
x = 4
```

名前（識別子）にはアルファベット（大文字・小文字）、アンダースコア、半角数字が利用可能。

型	意味	例
int	整数	`x = 1`
float	実数	`x = 1.5`
str	文字列	`x = 'Deep Learning'`
bool	真偽	`x = True, y = False`
list	配列	`x = [1, 2, 3]`
tuple	配列（書き換え不可能）	`x = (1, 2, 3)`

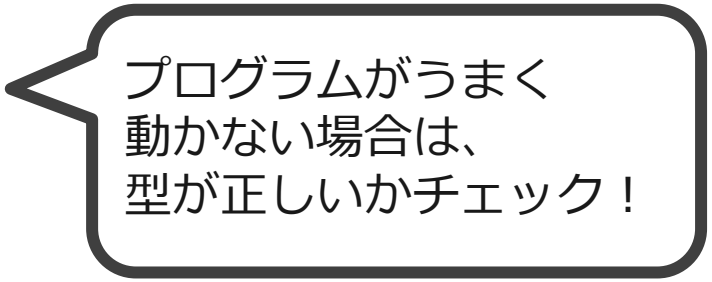
データ型と変数

Pythonは型を宣言しない **動的型付け** の言語。
代入された値から型が推定される。

型	意味	例
int	整数	`x = 1`
float	実数	`x = 1.5`
str	文字列	`x = 'Deep Learning'`
bool	真偽	`x = True, y = False`
list	配列	`x = [1, 2, 3]`
tuple	配列（書き換え不可能）	`x = (1, 2, 3)`

型の調査方法

```
> type('おはよう')
str
```



コメント

Pythonコード中に実行対象としないコメントを記述できる。

- 一行コメント（#以降）

```
# これはコメント
print('OK') # これもコメント
```

```
# print('Hello World!’)
x = 4
```

- 複数行のコメント（”””～”””）

```
"""
コメント
コメント
"""
```

データ型 | 演習

以下の値を持つ変数を定義し、型と値をそれぞれ出力しましょう。

1. 整数：年齢
2. 実数：靴のサイズ
3. 文字列：出身地
4. 真偽値：男性かどうか
5. 配列：好きな食べ物
6. タプル：姓と名

解答例

```
x = ['グミ', 'GPU', 'ジム']
print(type(x), x)
```

数値と文字列

数値型

数値をあつかうデータ型。

整数（int）

```
x = 4
y = 25
```

実数（float）

```
x = 3.5
y = 2.25
```

算術演算

演算子	意味	例	解
+	加算	5 + 3	8
-	減算	5 - 2	3
*	乗算	3 * 2	6
/	除算	8 / 2	4
//	除算（小数点以下切り捨て）	17 // 3	5
%	余剰	17 % 3	2
**	累乗	2 ** 3	8

解の型に注意！

- 数式中に float型 が含まれている場合、解が整数でも float型 となる
- int型 同士の除算の場合、解が整数でも float型 となる

5 + 2 = 7
int int int

5 + 2.0 = 7.0
int float float

4 / 2 = 2.0
int int float

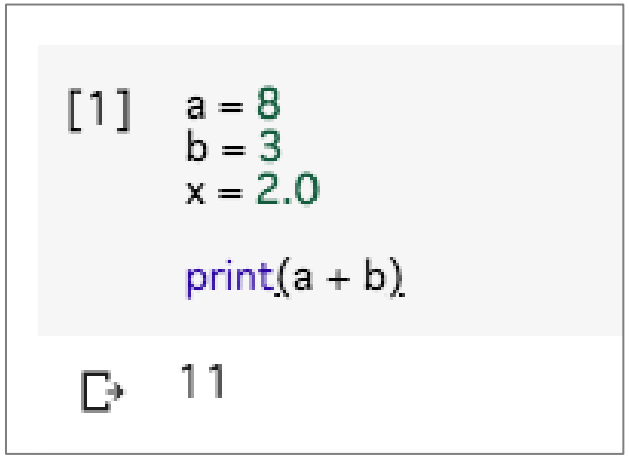
算術演算 | 演習

以下の変数を定義し、式を出力しましょう。
実行する際に、解の型が何になるか予想しましょう。

```
a = 8
b = 3
x = 2.0
```

1. a + b
2. a + x
3. a – b
4. b * x
5. a / x
6. a / b
7. a // b
8. a % b
9. b ** a

解答例



文字列

文字列、テキストをあつかうデータ型。
ダブルクォートかシングルクォートでくくる。

```
text1 = "これは文字列"
text2 = 'Hello'
```

文字列の連結

「+」演算子で文字列を連結できる

```
"Hello" + "World"
```

```
text1 = "Hello"
text2 = 'Python'

x = text1 + " " + text2
```

データ型の変換

文字列型でないものとは連結できない。

```
x = 30
print("私は" + x +"歳です。")
```

エラー

文字列型への変換

```
str(30)
```

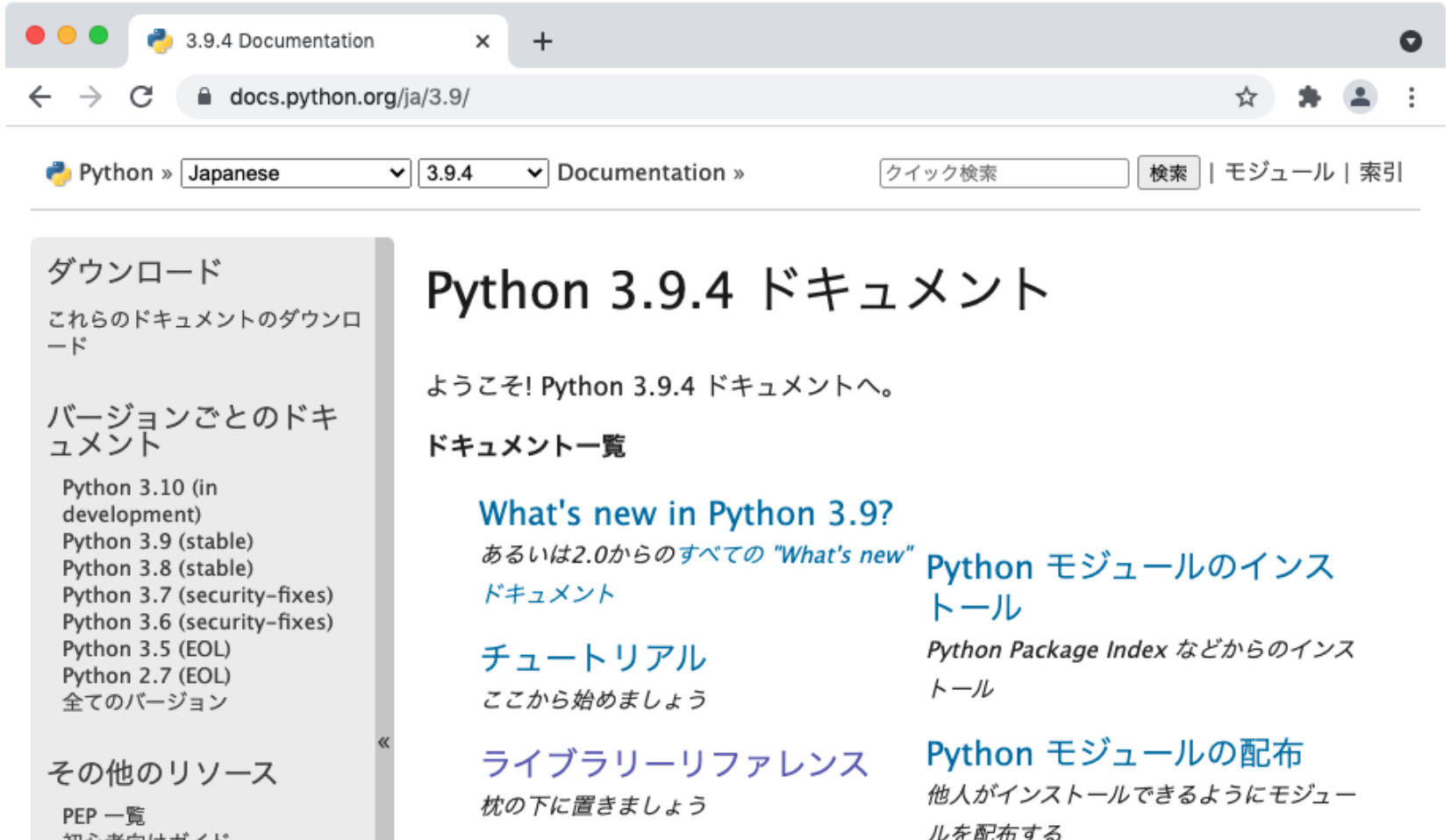
整数型への変換

```
int('30')
```

APIドキュメントの参照

言語の仕様や可能な処理の内容を確認できる。

https://docs.python.org/ja/

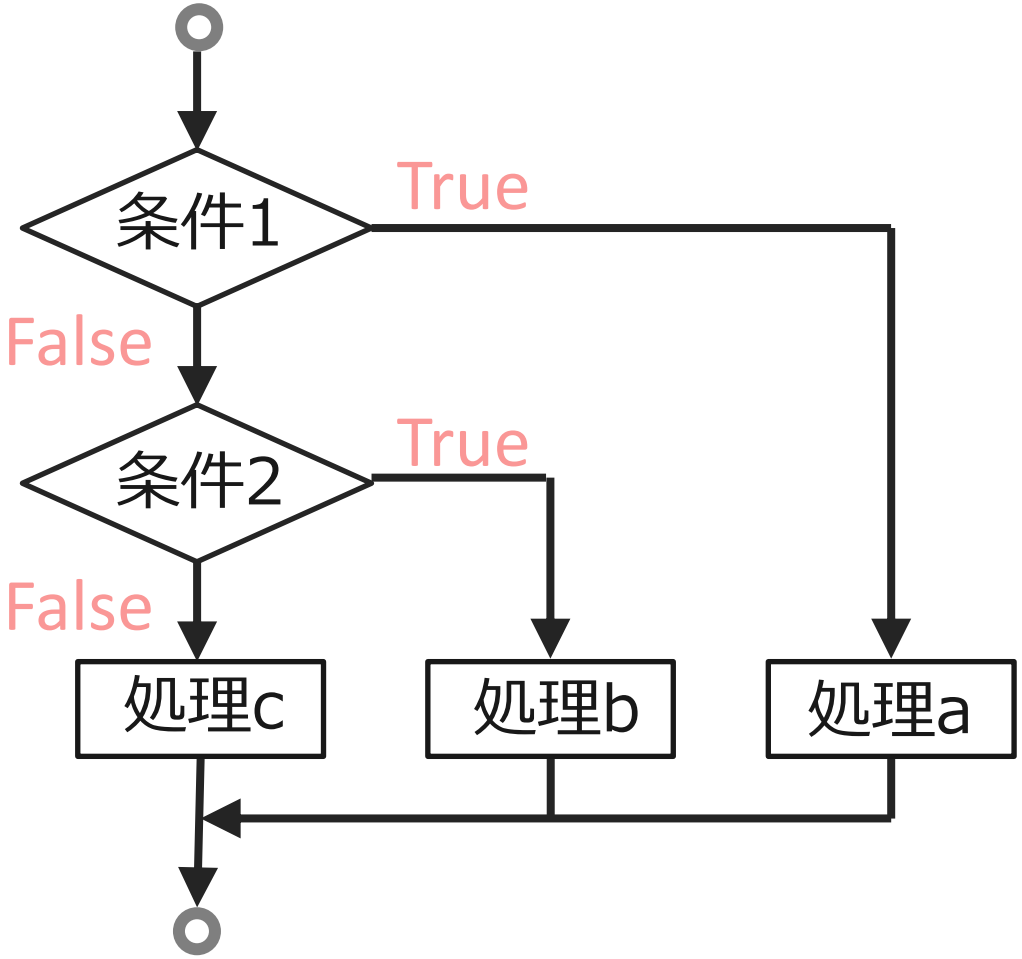


条件分岐

条件分岐

条件によって処理を分岐させる。（条件式は真偽値を返す式）

```
if 条件式1:
    処理内容a
elif 条件式2:
    処理内容b
else:
    処理内容c
```



elif は、なくても複数あってもよい。
else は、なくてもよい。

if 文

- 条件分岐は **if** / **elif** / **else**
- if の後に カッコ( ) は不要
- 条件の後に コロン :
- 条件内記述はインデントを下げる

```
age = 14

# 年齢に応じて料金を出力
if age < 12:
  print('500円')
elif age < 16:
  print('800円')
else:
  print('1000円')
```

型	例	意味
==	a == b	aとbが等しい
!=	a != b	aとbが等しくない
>	a > b	aがbより大きい
>=	a >= b	aがbより大きいまたは等しい
<	a < b	aがbより小さい
<=	a <= b	aがbより小さいまたは等しい

if 文

- 複数条件をつなげたい場合は **and** / **or** を使用
  ※ && / || は使用出来ない

15歳以上 かつ 18歳以下 を対象とするとき

```
age = 16

if 15 <= age and age <= 18:
  print('割引対象です。')
```

15歳 以下 もしくは 60歳以上 を対象とするとき

```
age = 62

if age <= 15 or 60 <= age:
  print('割引対象です。')
```

if 文 | 演習

以下の課題を if 文 を用いて実装しましょう。

1. 任意の数値 x に対して以下を出力
   3で割り切れるとき「3の倍数です。」
   それ以外のとき「3の倍数ではありません。」

2. 任意の数値 year が閏年かどうかを出力
   閏年の条件は以下
   ・4で割り切れる
   ・4でも100でも割り切れるとき、閏年でない
   ・4でも100でも400でも割り切れるとき、閏年

解答例

```
[35]  x = 17

      if x % 2 == 0:
        print('偶数です。')
      else:
        print('奇数です。')
```

```
奇数です。
```

リスト

リスト

型の異なる要素を格納できる入れ物。

```
# 1次元リスト
x = [1, 2, 3, 1]

# 2次元リスト
y = [[1, 2, 3] , [4, 5, 6]]
```

```
# 型の異なる要素
z = [3, 3.14, '円周率']

# 空のリスト
e = []
```

リストの基本操作

各要素へのアクセス（インデックスによるアクセス）

```
x = [1, 2, 3]
print(x[0]) # 1

y = [[1, 2, 3] , [4, 5, 6]]
print(y[1][2]) # 6

# 負のインデックス（末尾から）
print(x[-1]) # 3
```

リストの長さの取得方法

```
x = [1, 2, 3]
len(x) # 3
```

意図した通りにデータが
取得・定義出来ているか、
簡易的にチェックできる

リストの基本操作

各要素の更新

```
x = [1, 2, 3]
x[0] = 4
print(x[0]) # 4

y = [[1, 2, 3] , [4, 5, 6]]
y[1][2] = 7

# 負のインデックス（末尾から）
x[-1] = 10
```

リストの基本操作

要素の追加：append

```
x = [1, 2, 3]
x.append(5)
```

要素の挿入：insert(挿入位置, 要素)

```
x = [1, 2, 3]
x.insert(0,5)
```

要素の削除：del

```
x = [1, 2, 3]
del x[1]
```

要素が含まれているか？：in

```
x = [0, 1, 2, 0, 3]
0 in x
```

要素が含まれていないか？：not in

```
x = [0, 1, 2, 0, 3]
0 not in x
```

15歳、20歳、25歳、30歳 を対象とするとき

```
age = 20

if age in [15, 20, 25, 30]:
  print('割引対象です。')
```

リストの基本操作

リストの結合

```
x = [1, 2, 3]
y = [4, 5, 6]
x + y
```

リストのソート

```
x = [1, 4, 2, 6, 5]
x.sort()
x.sort(reverse=True) # 逆順
```

リストのソート（元のリストを変更しない）

```
x = [1, 4, 2, 6, 5]
y = sorted(x)
z = (x, reverse=True) # 逆順
```

リスト | 演習

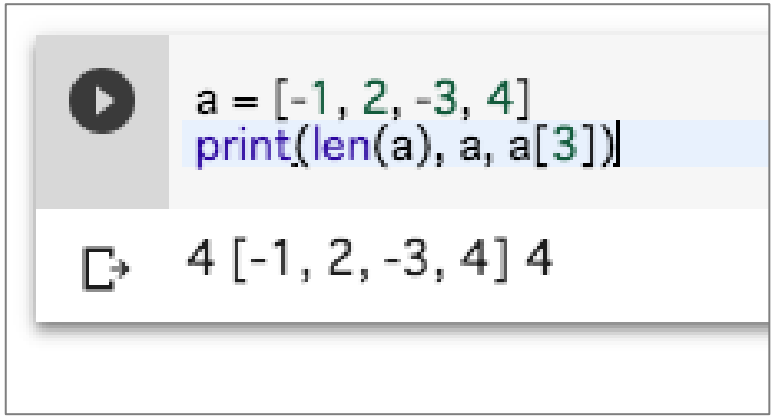
以下のリストを定義し、リストの長さ・リスト全体・指定された要素 を出力しましょう。

1. [23, 470, 78, 0, -12]
   出力する要素：23

2. [‘月’, ‘火’, ‘水’, ‘木’, ‘金’, ‘土’, ‘日’]
   出力する要素：金

3. [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
   出力する要素1：[4, 5, 6]
   出力する要素2：9

解答例



以下の課題を if 文 を用いて実装しましょう。

1. 以下を休診日としてリストで保持
   休診日：[日曜, 水曜, 金曜]
   任意の曜日 day が上記リストに含まれれば「休診日です。」と出力

スライシング

リスト等の特定の範囲を切り取る機能。

[始まりの要素番号：終わりの要素番号：何要素ごとに取得するか]
※ 各属性は省略可

スライス位置	0	1	2	3	4
リスト	|	1 |	2 |	3 |	4 |
スライス位置(マイナス)	-4	-3	-2	-1	

```
x = [1, 2, 3, 4]

x[:]
# すべての要素
# [1, 2, 3, 4]

x[:3]
# 要素番号 2まで
# [1, 2, 3]
```

```
x = [1, 2, 3, 4]

x[1:]
# 要素番号 1から全て
# [2, 3, 4]

x[1:4:2]
# 要素番号 1から3
# 2要素ごとに取得
# [2, 4]
```

```
x = [1, 2, 3, 4]

x[-2:]
# 後ろから2番目以降
# [3, 4]

x[::-1]
# 全要素
# 後ろから1要素ごとに取得
# [4, 3, 2, 1]
```

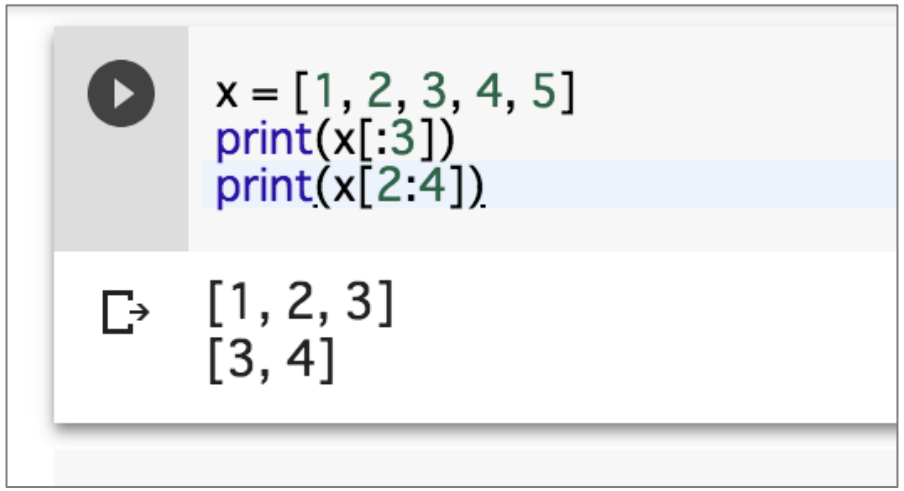
スライシング | 演習

以下のリストを定義し、スライシングを用いて値を出力しましょう。

リスト：1から20までのリスト

1. 出力値：1から15まで
2. 出力値：5から17まで
3. 出力値：1から20まで3要素ごとに

解答例



その他のシーケンスと繰り返し

レンジ

連続した整数のデータを作成可能。
書き換え不可能な range型 になるので、リストとして扱いたい場合は変換が必要。

range([始まりの数値,] 終わりの数値 [, 増加する量])

※ [ ]内はオプション
※ 終わりの数値 -1 まで生成

```
range(5)
# 0, 1, 2, 3, 4

range(5, 8)
# 5, 6, 7

range(10, 20, 3)
# 10, 13, 16, 19
```

リストへの変換

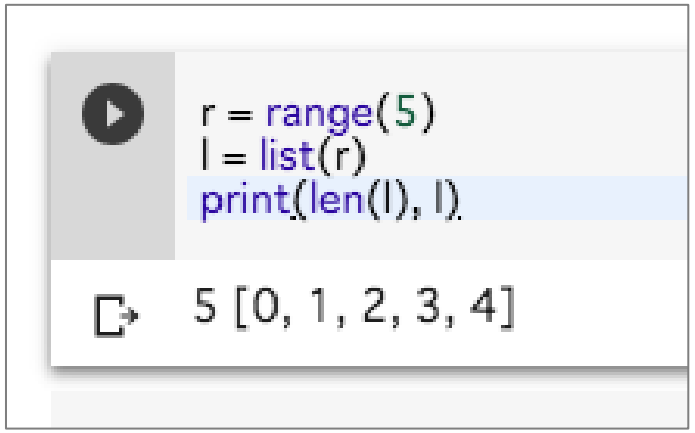
```
r = range(5)
l = list(r)
# [0, 1, 2, 3, 4]
```

レンジ | 演習

以下のレンジを定義し、リストへ変換しましょう。
その後、リストの長さ・リスト全体 を出力しましょう。

1. 0～15
2. 100～120
3. 10～50, 5ずつ増加

解答例



タプル

書き換え不可な配列。
配列と同じように要素へアクセス出来るが、要素に再代入することは出来ない。

```
# 1次元タプル
x = (1, 2, 3)

# 2次元リスト
y = ((1, 2, 3) , (4, 5, 6))
```

使用用途

- 処理結果など変更不要のデータ
- 順序に意味があり変更されたくないデータ
- 複数値を辞書型のキーにしたい場合
  （辞書型については後述）

要素が１つのタプル

(1) と記入すると数式で使用される ( ) と認識されてしまう。
要素が１つのときはカンマをつけて区別する。

```
a = (1)
type(a) # int

b = (1, )
type(b) # tuple
```

タプル | 演習

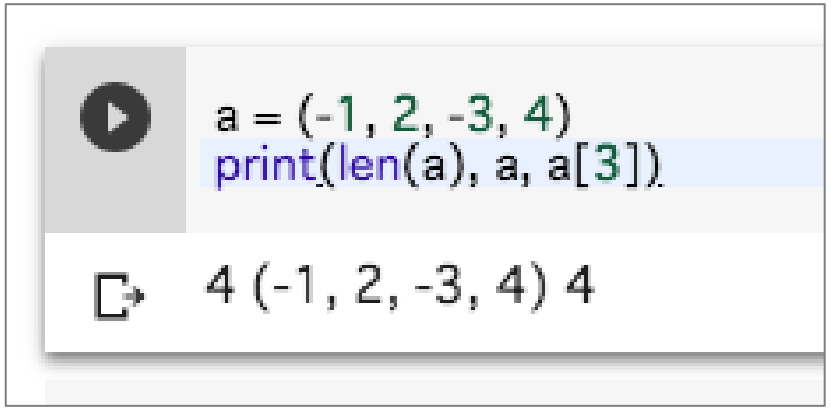
以下のタプルを定義し、タプルの長さ・タプル全体・指定された要素 を出力しましょう。

1. (23, 470, 78, 0, -12)
   出力する要素：23

2. (‘月’, ‘火’, ‘水’, ‘木’, ‘金’, ‘土’, ‘日’)
   出力する要素：金

3. ((1, 2, 3), (4, 5, 6), (7, 8, 9))
   出力する要素1：[4, 5, 6]
   出力する要素2：9

解答例



繰り返し | for文

- **for** 変数 **in** シーケンス

  ※ シーケンス：順番に並んだデータのこと

```
fruits = ['バナナ', 'リンゴ']
for fruit in fruits:
  print(fruit)

# バナナ
# リンゴ
```

Step.1　**最初の要素を参照**

['バナナ', 'リンゴ']

- fruit = 'バナナ'
- 処理

Step.2　**次の要素を参照**

['バナナ', 'リンゴ']

- fruit = 'リンゴ'
- 処理

Step.3　**次の要素がなくなったら終了**

繰り返し | for文

- **range** でループ回数を指定

  （**range**の要素を繰り返すと解釈できる）

```
for i in range(5):
  print(i)

# 0
# 1
# 2
# 3
# 4
```

Step.1　**最初の要素を参照**

0 1 2 3 4

- i = 0
- 処理

Step.2　**次の要素を参照**

0 1 2 3 4

- i = 1
- 処理

…

Step.6　**次の要素がなくなったら終了**

繰り返し | for文

- **enumerate** で要素とインデックスの両方を参照可能

  「何番目」が「何」と両方必要な際に便利

```
fruits = ['バナナ', 'リンゴ']
for index, fruit in enumerate(fruits):
  print(index, fruit)

# 0 バナナ
# 1 リンゴ
```

繰り返し | 演習

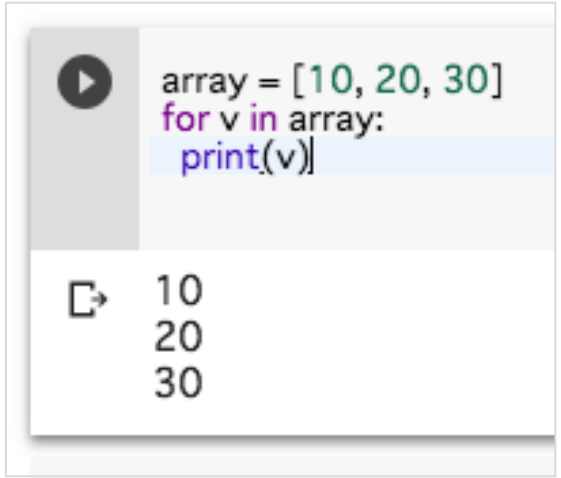
以下の課題を for文 を用いて実装しましょう。

1. 月曜から日曜までの全曜日を持つリストを作成
   全ての曜日を順番に出力

2. 1000から2000までの年数のうち、閏年の年のみ出力
   閏年の条件は以下
   ・4で割り切れる
   ・4でも100でも割り切れるとき、閏年でない
   ・4でも100でも400でも割り切れるとき、閏年

3. 任意の名前のリストを作成
   インデックス番号と名前を両方出力

解答例



辞書型

辞書型

キーと値のペアのリストを保持できる。
（値に名前をつける）

```
x = {
   'key_int': 123,
   'key_str': 'Hello Dict!'
}
```

辞書型の基本操作

各要素へのアクセス

```
x['key_int']
# キー「key_int」の値を取得
# 123
```

各要素の更新・追加

```
x['key_int'] = 245
x['key_new'] = 500
```

辞書型の基本操作

keys で全キー、**values** で全値、**items** で全キーと値の組み合わせを取得出来る。
※ items　ではキーと値が タプル型 となって取得

```
x.keys()
# dict_keys(['key_int', 'key_str'])
```

```
x.values()
# dict_values([123, 'Hello Dict!'])
```

```
x.items()
# dict_items([('key_int', 123), ('key_str', 'Hello Dict!')])
```

辞書 | 演習

以下のキーと値を持つ辞書型データを作成しましょう。
その後、各キーを用いて全ての値を出力しましょう。

1. キー：name
   値：あなたの氏名

2. キー：age
   値：あなたの年齢

3. キー：birthplace
   値：あなたの出身地

解答例

```
[12]  your_info = {
          'name': 'AI Engineer',
          'age': '32',
          'birthplace': '東京'
      }
      print('名前:', your_info['name'])
```

```
名前: AI Engineer
```

処理の定義と利用

処理の定義と利用

関数

処理のまとまりに名前をつけて再利用できるようにしたもの。

モジュール

複数の関数などを別ファイルにまとめ、再利用できるようにしたもの。

関数

処理のまとまりを「関数」として定義することで、再利用することが容易になる

- 関数は **def** で定義
- 引数に デフォルト値 を設定出来る
- 返り値は タプル型 で複数値返せる

```
def add(a, b):
  sum = a + b
  return sum

result = add(2, 7)
print(result)
# 9
```

```
def greet(name, greeting='Hello'):
  print(greeting + ', ' + name)

greet('Andy')
# Hello Andy

greet('Jerry', 'Good morning')
# Good morning, Jeryy
```

```
def getInfo():
  return 'Rob', '51歳', '1967年生まれ'

info = getInfo()
print(info)
# ('Rob', '51歳', '1967年生まれ')
```

関数 | 演習

以下の関数を作成し、返り値を出力してみましょう。

1　機能：
　　任意の数値を2つ渡すと、4つの演算結果を返す関数
引数：
　　任意の数値 a, 任意の数値 b
返り値：
　　加算結果、減算結果、乗算結果、除算結果

出力例

```
[15]  result = arithmetic(10, 2)
      print(result)

      (12, 8, 20, 5.0)
```

2　機能：
　　任意の年を渡すと、閏年かどうかを判定する関数（閏年であれば真を返す関数）
引数：
　　任意の年 year
返り値：
　　閏年かどうか

モジュールのインポート

- **import** でモジュールをインポート
- **from** と **import** でモジュールから直接クラスや関数をインポート
- **as** を用いて別名をつける

```
import random
# random というモジュールをインポート

random.randint(0, 10)
# 0 から 10 の間でランダムな整数を出力
# 6
```

```
from math import cos
# math というモジュールから 関数 cos をインポート

cos(90)
# cos(90) を出力
# -0.4480736161291701
```

モジュールと関数 | 演習

以下の関数を作成し、返り値を出力してみましょう。

1　機能：
　　任意のリストを渡すと、ランダムにそのリストの要素を返す関数
引数：
　　任意のリスト
返り値：
　　リスト

2　機能：
　　実行するとじゃんけんの手（グー、チョキ、パー）をランダムに返す関数
引数：
　　なし
返り値：
　　じゃんけんの手
※ 1で作成した関数を使用すること。

クラウドAPIの利用

AI API体験「A3RT」

A3RT（アート） https://a3rt.recruit.co.jp/

リクルートが提供する機械学習API群。
リクルートグループ内へ展開するためにプロジェクト化
されたソリューションの総称。

Text Classification API

ディープラーニングを用いて文章の分類を行うAPI。
学習済みモデルでは、求人系の文章とどの職種の求人かというラベルを学習しており、
単語や文章を入力すると、どの職種に関連する文章か、カテゴリを予測してくれる。

※ 氏名・メールアドレス・住所・電話番号など、
　個人を特定しうる情報はアップロード禁止

A3RT　Text Classification API | 仕様

エンドポイント

URL	https://api.a3rt.recruit.co.jp/text_classification/v1/classify
メソッド	POST

リクエストパラメータ

パラメータ名	説明
apikey	APIキー。
model_id	モデルID。 ‘default’を指定することで、 学習済みモデルが使用出来る。
text	分類したい文章。 1000文字以内、500単語以内。

レスポンスフィールド

フィールド名		説明
status		処理ステータス。
message		ステータスメッセージ。
classes		結果。
	label	予測ラベル（カテゴリ）
	probability	確度

A3RT　Text Classification API | 例

モデル「default」に対して、
「システムの企画から開発・運用まで
幅広く関われます。」という文章を分類させる。

```
{
   “apikey”: “”,
   “model_id“: “default”,
   “text”: “システムの企画から開発・運用まで
            幅広く関われます。”
}
```

「ITエンジニア（システム開発・SE・インフラ）」
に「0.930484」の確度で分類される、
という結果が返る。

```
{
   “status”: 0,
   “message“: “ok”,
   “classes”:[
      {
         “label”: “ITエンジニア（システム開発・
                  SE・インフラ”,
         “probability”: 0.930484
      },
      …
   ]
}
```

A3RT　Text Classification API | 演習

以下のコードを実行し、A3RT API を体験してみましょう。

```
import requests
import pprint

# POST先のURLとAPI Key
url =
'https://api.a3rt.recruit.co.jp
/text_classification/v1/classif
y'
apikey = 'XXXXX'

# 分類させる文章
text = 'システムの企画から開発・運
用まで幅広く関われます。'
```

```
# リクエストパラメータ
params = {
    'apikey': apikey,
    'model_id': 'default',
    'text': text
}

# POST
response = requests.post(url,
params)

# 結果を取得
response_json = response.json()
```

```
# ステータスコードを出力
status =
response_json['status']
message =
response_json['message']
print(status, message)

# ステータスが ok のとき、結果を出
力
if status == 0:
    classes =
response_json['classes']

    # 辞書型データを整形して出力
    pprint.pprint(classes)
```

```
# 結果を見やすく整形
for result in classes:
    print(result['label'], result['probability'])
```

text 内の文章を変更し、
分類結果がどう変わるか試してみましょう。

付録

Fizz Buzz ゲームルール

- 1から順に整数を発言（出力）していく
- ただし
  - 3の倍数のときは、数字の代わりに「Fizz」と発言（出力）
  - 5の倍数のときは、数字の代わりに「Buzz」と発言（出力）
  - 3の倍数かつ5の倍数のときは、数字の代わりに「FizzBuzz」と発言（出力）

出力例

1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16 17 Fizz ...

上記のルールにのっとり、nまでの数字（文字列）を出力する
関数 fizzbuzz(n) を作成し、fizzbuzz(100)を実行せよ。