

# 第8章 画像データの処理

画像を数値データとして扱うことで大量に効率よく処理する方法

# 画像データの処理

Pythonのライブラリを使って画像を数値データとして扱うことで、画像ファイルを大量に効率よく処理できる

数値データとして処理する目的の1つに機械学習モデルへの入力がある

PillowやNumPyを使って画像データを処理する便利な機能を紹介する

# Pillowを使った画像の加工

08-01-pillow.ipynb

# Pillowを使った画像の加工

Pillowライブラリの機能でさまざまな画像の加工が行える

次の加工方法について解説する

- サムネイル化
- クロップ（切り取り）
- グレースケール（モノクロ）化

# サムネイル化

SNSやWeb記事でよく見かける、縮小した画像をサムネイルという

```
from PIL import Image

with Image.open("data/sample.jpg") as im:
    im.thumbnail((128, 128))
    im.save("data/thumb-test.jpg")
```



- サムネイルの縦横比は、元の画像と同じになる
- サムネイルのサイズをタプルで(128, 128)と宣言している
- これは、幅128ピクセル以内、高さ128ピクセル以内という指示
- 「最大で128ピクセルまでを許容する」という意味なので、今回使った横長の画像からは、高さが128ピクセルより小さいサムネイルが生成される
- data/thumb-test.jpgに画像を保存

# サムネイル化

## サムネイル画像表示をする

Google Colabで実行する場合に画像をノートブック内で表示する設定

```
from PIL import ImageShow  
  
ImageShow.register(ImageShow.IPythonViewer, order=0)
```

保存した画像 data/thumb-test.jpgを読み込んで表示

```
with Image.open("data/thumb-test.jpg") as thumb:  
    print(thumb.size)  
    thumb.show()
```

(128, 85) 横:128 縦:85 縦横比はオリジナルを保つ



# クロップ（切り取り）

- 画像を切り取ることをクロップという
- 画像を扱う機械学習モデルには正方形の画像を入力としているものが多く、前処理として元の画像の情報をできるだけ残すよう画像の中央を基準に正方形にクロップすることがよくある
- クロップするcrop()メソッドは、画像を切り取る「左」「上」「右」「下」の座標位置を、この順に指定する必要がある
- 左上が0で、右方向、下方向に座標が大きくなる



# クロップ（切り取り）

```
def calc_cropping_box(image):  
    """  
    クロップする四隅の座標を計算する関数  
    """  
    x, y = image.size  
    print(f"{x=}, {y=}")  
    if x > y: # 幅が大きい（横長）場合の処理：高さが正方形の1辺の長さ  
        left = (x - y) // 2  
        upper = 0  
        right = left + y  
        lower = y  
    else: # 高さが大きい（縦長）場合の処理：幅が正方形の1辺の長さ  
        left = 0  
        upper = (y - x) // 2  
        right = x  
        lower = upper + x  
    box = (  
        left,  
        upper,  
        right,  
        lower,  
    ) # 切り取る四隅の座標  
    print(f"{box=}")  
    return box
```

画像を正方形に切り取る為に、切り取る四隅の座標計算する  
calc\_cropping\_box()関数を定義

元の画像が横長の場合と縦長の場合  
とに処理を分ける必要がある

// 割り算（切り捨て）



# クロップ（切り取り）

切り取る画像の座標を計算し、画像をcropped-test.jpgに切り出す

```
with Image.open("data/sample.jpg") as im:  
    box = calc_cropping_box(im)  
    cropped = im.crop(box) # boxの座標で切り出し  
    cropped.save("data/cropped-test.jpg")
```

```
x=800, y=533  
box=(133, 0, 666, 533)
```

画像をcropped-test.jpgを読み込んで表示

```
with Image.open("data/cropped-test.jpg") as cropped:  
    x, y = cropped.size  
    print(f"{x=}, {y=}")  
    cropped.show()
```



x=800, y=533



x=533, y=533

# グレースケール（モノクロ）化

カラー画像をモノクロ画像に変換することをグレースケール化という  
多くの機械学習モデルでは、画像をモノクロで扱う  
たとえば、物体認識に色情報を使わない

画像オブジェクトのconvert()メソッドの第1引数modeに“L”を渡すことで、  
カラー画像をグレースケール化できる

```
with Image.open("data/sample.jpg") as im:  
    im_gray = im.convert("L")  
    im_gray.save("data/gray-test.jpg")
```

ここで指定したモードLは、8ビットのモノクロである  
フルカラー(トゥルーカラー)のモードRGBは、赤緑青の3色をそれぞれ8ビット計  
1677万色で表現する  
ほかには、RGBAやCMYKなどのモードも存在する



# NumPyを使った画像データ処理


08-02-numpy.ipynb

# NumPyを使った画像データ処理

画像データをNumPy配列に変換して扱っていると、下記のような処理をコード上で行える。  
NumPyとOpenCVを使った処理を紹介する

- 画像とNumPy配列の間の変換
- 拡大／縮小
- 切り取り／貼り付け
- 分割（n等分）
- 回転（90度単位）
- 反転
- 複製／貼り合わせ
- 次元の操作
- 配列の形式（HWCとCHW）の変換
- 色（チャンネル）の積み重ね／変換

# 画像データ



**USC Viterbi**  
School of Engineering

**Signal and Image Processing Institute**  
Ming Hsieh Department of Electrical and Computer Engineering

## The USC-SIPI Image Database

The USC-SIPI image database is a collection of digitized images. It is maintained primarily to support research in image processing, image analysis, and machine vision. The first edition of the USC-SIPI image database was distributed in 1977 and many new images have been added since then.

The database is divided into volumes based on the basic character of the pictures. Images in each volume are of various sizes such as 256x256 pixels, 512x512 pixels, or 1024x1024 pixels. All images are 8 bits/pixel for black and white images, 24 bits/pixel for color images. The following volumes are currently available:

<a href="#">Textures</a>	Brodatz textures, texture mosaics, etc.
<a href="#">Aerials</a>	High altitude aerial images
<a href="#">Miscellaneous</a>	The mandrill, peppers, and other favorites
<a href="#">Sequences</a>	Moving head, fly-overs, moving vehicles

## File Format and Names

**Note:** It is the database user's responsibility to figure out how to read the images into whatever computer they will be using, and how to access the files within application programs. USC-SIPI does not have the resources to provide assistance in these areas. If you are in doubt about whether or not you will be able to read the images on your computer, please check with your system managers and show them the description of the database and the image formats.

All images in the database are currently stored in TIFF format. Some information about the TIFF format is available from [Wikipedia](#) and from [Adobe Systems](#). The "libtiff" library of C functions for reading and writing TIFF images is available from <https://gitlab.com/libtiff/libtiff>. The "netpbm" collection of image format conversion programs (<http://netpbm.sourceforge.net/>) can convert between TIFF and many different formats.

Previous versions of the database that were only distributed on magnetic tape used a raw binary format for the data instead of the TIFF format. The TIFF files in the current edition can be converted back to a raw binary format using the TIFF software available at the above sites.

A sample C program for converting an image from TIFF to raw format on a Unix system is available ([tiff2raw.c](#)). This program should convert the color (24 bits/pixel) and grayscale (8 bits/pixel) images in the database into the raw binary format files. This program requires the "libtiff" library available at the site mentioned above. Note that while we believe this program works properly on the TIFF images in the database, it has NOT been tested on other TIFF images and may not convert them correctly.

Many of the images in the database have numerical filenames such as 4.2.03. These relate to an numbering scheme that was used with an earlier edition of the database that was released in 1981.

南カリフォルニア大学の画像データベース  
<https://sipi.usc.edu/database/database.php>  
画像処理研究者間で画像を共有するための画像データベース

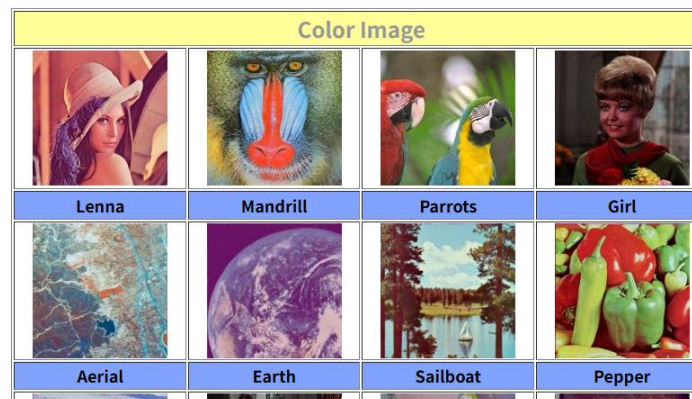
出典不明のものが大半であり、著作権を侵害している可能性もあると言われている

参考：日本のサイト  
神奈川工科大学  
[http://www.ess.ic.kanagawa-it.ac.jp/app\\_images\\_j.html](http://www.ess.ic.kanagawa-it.ac.jp/app_images_j.html)

## 標準画像／サンプルデータ

### Standard Image / Sample Data

種々の画像処理手法の性能を比較するには、同じ画像に対して行う必要があります。  
そこで以下のような画像を標準として使い、種々の画像処理との比較検討に使っています。



# 画像データセット

近年、画像を用いた機械学習では「著作権」「倫理」を重視した公的データセットの利用が強く推奨されている

ちょっとしたフィルタ処理のサンプルを見せたい場合でも、出典が明確そうなデータセットから一枚使うのが良く、下記がオススメ

- [ImageNet](#)
- [ALOT](#)
- [Flickr-Faces-HQ Dataset \(FFHQ\)](#)
- [AFHQ \(Animal Faces-HQ\)](#)

# 画像をNumPy配列に変換

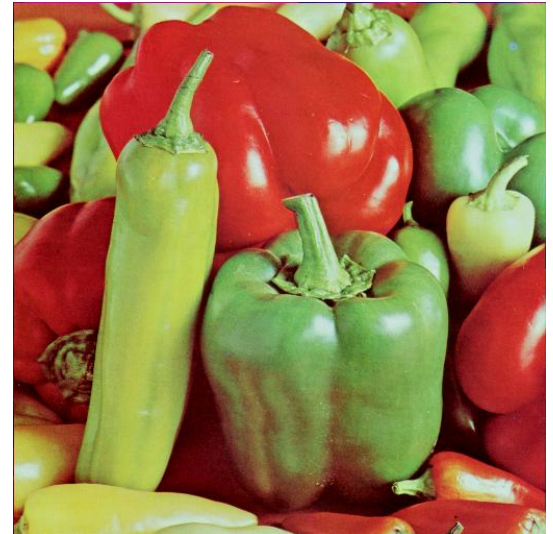
画像データを数値データに変換することで、データ分析や機械学習に使える。NumPyの配列ndarrayに変換し、データの中身を確認する

Pillowを使ってデータを読み込み

```
from PIL import Image
import numpy as np

im = Image.open("data/4.2.07.tiff")
print(im.format)
print(im.size)
print(im.mode)
```

TIFF  
(512, 512)  
RGB



peppers: 4.2.07.tiff

TIFFフォーマット、解像度が幅512ピクセル×高さ512ピクセル RGB画像



# 画像をNumPy配列に変換

画像は3次元の行列で表現できる。この数値データをもとに物体検出など機械学習に使える

```
im_array = np.array(im)
print(type(im_array))
```

<class 'numpy.ndarray'>

NumPy配列に変換

```
print(im_array.shape)
```

(512, 512, 3)

3次元のデータ「縦／横／各色」の情報がある

```
print(im_array[0, 0, :])
```

[101 0 0] -> R=101, G=0, B=0

1次元目が高さ（縦）

左上角のピクセルのRGBデータを確認

```
print(im_array[256, :, :])
```

```
[[131  0  0]
 [188 60 51]
 [189 76 56]
 ...
 [204 48 46]
 [192 38 36]
 [205 46 44]]
```

256行目にあるピクセルのRGBデータの確認

```
print(im_array[256, 128, :])
```

[176 192 80]

256行目、128列目にあるピクセルのRGBデータの確認



# NumPy配列を画像に変換

- 数値データ（NumPy配列）を画像（Imageオブジェクト）に変換するには、PIL.Imageモジュールのfromarray()関数を使う
- この関数はPILのImageオブジェクトを返す

```
Image.fromarray(im_array)
```



# 拡大／縮小

NumPyだけを使って画像の拡大や縮小をするのは大変なので、Imageオブジェクトのまま処理することもある

また、NumPy配列の状態で処理をする必要があればOpenCVライブラリを使う

# 拡大／縮小

拡大も縮小も、変換後の画像サイズをピクセル数（配列の大きさ）の  
タプル(幅（横）, 高さ（縦）)として第2引数dsizeに指定する

```
import cv2 as cv

# 幅128ピクセル、高さ256ピクセルに変換
im_resized1 = cv.resize(im_array, (128, 256))
Image.fromarray(im_resized1)
```



# 拡大／縮小

変換倍率を引数fxとfyに指定できます。ただし、引数dsizeにNoneを渡す必要がある

```
#横方向に0.25倍、縦方向に0.5倍  
im_resized2 = cv.resize(im_array, dsize=None, fx=0.25, fy=0.5)  
Image.fromarray(im_resized2)
```

拡大や縮小に伴うピクセルの補間方法は、デフォルトでは線形補間  
cv.INTER\_LINEARが使われる

OpenCVの公式ドキュメントでは、表8-1の補間方法が推奨されています。補間方法は引数interpolationに指定します。



# 拡大／縮小

- 拡大や縮小に伴うピクセルの補間方法は、デフォルトでは線形補間 `cv.INTER_LINEAR` が使われる
- 補間方法は引数 `interpolation` に指定する

```
im_resized3 = cv.resize(  
    im_array,  
    (128, 256),  
    interpolation=cv.INTER_AREA,  
)  
Image.fromarray(im_resized3)
```



表 8-1 拡大や縮小に伴うピクセルの補間方法

変換種類	補間方法
拡大	<code>cv.INTER_CUBIC</code> または <code>cv.INTER_LINEAR</code>
縮小	<code>cv.INTER_AREA</code>

それぞれの補間方法の詳細とほかの補間方法については、公式ドキュメントを参照のこと

# 拡大／縮小

# 縦に3倍、横に2倍

```
im_repeated = im_array.repeat(3, axis=0).repeat(2, axis=1)  
Image.fromarray(im_repeated)
```





# 切り取り／貼り付け

NumPy配列をスライスして画像の一部を切り取れる。また、スライスした配列を元の配列に代入して置き換えることで画像を貼り付けられる

```
# 中央右のピーマンを縦330、横280の長方形に切り取る  
im_cropped = im_array[170:500, 160:440, :]  
Image.fromarray(im_cropped)
```



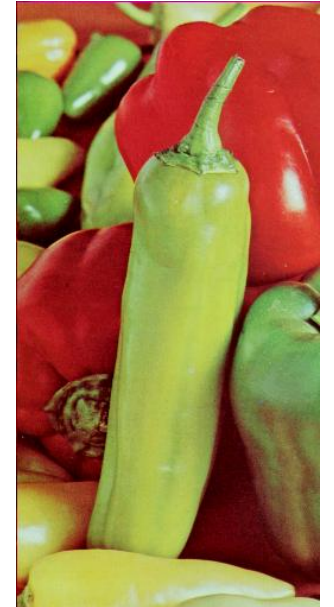
```
# 左上隅に貼り付け  
im_pasted = im_array.copy()  
im_pasted[0:330, 0:280, :] = im_cropped  
Image.fromarray(im_pasted)
```



# 分割（n等分）

- 画像を等サイズに分割するには、`split()`関数を使う。
- 分割数は第2引数`indices_or_sections`に、分割方向は引数`axis`に指定する
- 分割数は分割方向のサイズ（ピクセル数）の約数である必要がある
- 分割方向（引数`axis`）は、縦（行方向）が0、横（列方向）が1

```
# 左右に2等分
im_left, im_right = np.split(im_array, 2, axis=1)
Image.fromarray(im_left) # 左
```



左右に2等分

```
# 上下左右に2等分（4分割）
im_split_list = []
for part in np.split(im_array, 2, axis=0):
    im_split_list.extend(np.split(part, 2, axis=1))

Image.fromarray(im_split_list[3]) # 右下
```



上下左右に2等分（4分割）



# 90度単位での回転

- 画像を90度単位で回転するには、`rot90()`関数を使う。
- デフォルトの回転は反時計回りに90度で、回転回数を引数`k`に整数で渡す
- `k`に負の整数を渡すと時計回りになる

```
# 反時計回りに270°回転  
im_rotated270 = np.rot90(im_array, k=3) # k=-1でも同じ  
Image.fromarray(im_rotated270)
```



# 反転

- 画像を反転するには、flip()関数を使う
- 反転方向は引数axisに指定し、上下方向は0、左右方向（鏡と同じ）は1を指定する
- 上下左右に同時に反転するには、axis=(0, 1)と指定する

```
# 上下反転  
im_flipped_vertical = np.flip(im_array, axis=0)  
Image.fromarray(im_flipped_vertical)
```

```
# 左右反転  
im_flipped_horizontal = np.flip(im_array, axis=1)  
Image.fromarray(im_flipped_horizontal)
```

```
# 上下左右反転（180°回転と同じ）  
im_flipped_both = np.flip(im_array, axis=(0, 1))  
Image.fromarray(im_flipped_both)
```



# 上下反転



# 左右反転



# 上下左右反転（180°回転と同じ）

# 反転

- 反転方向axisを指定しないと、色（チャンネル）の値も順序が変わってしまうため、色合いがおかしくなる
- これは、axisに2を指定した場合も同じ

```
# 反転方向の指定なし  
im_flipped_wrong = np.flip(im_array)  
Image.fromarray(im_flipped_wrong)
```



# 複製／貼り合わせ

- 同じ画像を繰り返し並べて新たに1つの画像にするには、`tile()`関数を使う
- 複数の画像を並べて1つの画像にするには、`concatenate()`関数を使う
- `tile()`関数で同じ画像を繰り返す回数は、引数`reps`にタプル(縦方向の回数, 横方向の回数, 1)を渡して指定する
- タプルの3つ目の1は必ず必要で、これは色（チャネル）の軸方向には要素を複製しないことを表している

```
# 縦方向に3回、横方向に2回複製。  
im_tiled = np.tile(im_cropped, reps=(3, 2, 1))  
Image.fromarray(im_tiled)
```





# 複製／貼り合わせ

- 複数の画像を並べる場合には、並べる方向のNumPy配列のサイズが同じである必要がある（縦方向なら列数、横方向なら行数）。
- 並べる画像のNumPy配列をリストとして concatenate()関数の第1引数に指定し、並べる方向は引数axisに指定する。縦方向が0、横方向が1

```
# 縦方向に並べる
im_concatenated_vertical = np.concatenate(
    [im_array, im_flipped_vertical],
    axis=0,
)
Image.fromarray(im_concatenated_vertical)
```

```
# 横方向に並べる
im_concatenated_horizontal = np.concatenate(
    [im_array, im_flipped_horizontal],
    axis=1,
)
Image.fromarray(im_concatenated_horizontal)
```



# 縦方向に並べる



# 横方向に並べる

# 次元の操作

- 画像のNumPy配列は、縦（高さ）、横（幅）、色（チャンネル）からなる3次元配列であるが、機械学習モデルによっては1次元配列を入力値の前提としているものもある
- 機械学習モデルの入出力データを処理する中で、軸の追加や削除が必要になることもある
- `ravel()` : 多次元の配列を1次元に変換（参照を返すため、`flatten()`メソッドより高速）
- `expand_dims()` : 引数`axis`に指定した位置に軸を追加（挿入）
- `squeeze()` : 引数`axis`に指定した位置の長さ1の軸を削除

# 次元の操作

ravel()

例として、Rチャンネル（赤）を抽出し、その2次元配列im\_2dを1次元配列に変換する。flatten()メソッドと同じ結果となる

```
im_2d = im_array[:, :, 0] # Rチャンネルを抽出
print(f"{im_2d.shape=}")
print(f"{np.ravel(im_2d).shape=}") # 要素の数は512x512
print(np.array_equal(np.ravel(im_2d), im_2d.flatten())) # 同じ結果であることを確認
```

```
im_2d.shape=(512, 512)
np.ravel(im_2d).shape=(262144,)
True
```

```
im_order = im_2d[0:2, 0:2] # 配列の左隅の2行2列
print(im_order)
print(f'行を優先: {np.ravel(im_order, order="C")}') # C言語形式、デフォルト
print(f'列を優先: {np.ravel(im_order, order="F")}') # FORTRAN言語形式
```

```
[[101 140]
 [123 191]]
行を優先: [101 140 123 191]
列を優先: [101 123 140 191]
```

# 次元の操作

`expand_dims()`

例として、2次元配列`im_2d`に軸を追加します。追加する位置を引数`axis`に指定する

```
im_3d_last = np.expand_dims(im_2d, axis=-1) # 最後に追加  
im_3d_first = np.expand_dims(im_2d, axis=0) # 先頭に追加  
im_3d_1 = np.expand_dims(im_2d, axis=1) # 1番目に追加
```

```
print(f"{im_3d_last.shape=}")  
print(f"{im_3d_first.shape=}")  
print(f"{im_3d_1.shape=}")
```

```
im_3d_last.shape=(512, 512, 1)  
im_3d_first.shape=(1, 512, 512)  
im_3d_1.shape=(512, 1, 512)
```



# 次元の操作

`squeeze()`

例として、先ほど`im_2d`に軸を追加して作った3次元配列`im_3d_last`と`im_3d_1`から長さ1の軸を削除する  
削除する軸の位置は、引数`axis`に指定する

```
im_2d_squeezed_last = np.squeeze(im_3d_last, axis=-1)
im_2d_squeezed_1 = np.squeeze(im_3d_1, axis=1)
```

```
print(f"{im_2d_squeezed_last.shape=}")
print(f"{im_2d_squeezed_1.shape=}")
```

# 同じ結果であることを確認

```
print(
    np.array_equal(
        im_2d_squeezed_last,
        im_2d_squeezed_1,
    )
)
```

```
im_2d_squeezed_last.shape=(512, 512)
im_2d_squeezed_1.shape=(512, 512)
True
```

# HWCとCHWとの変換

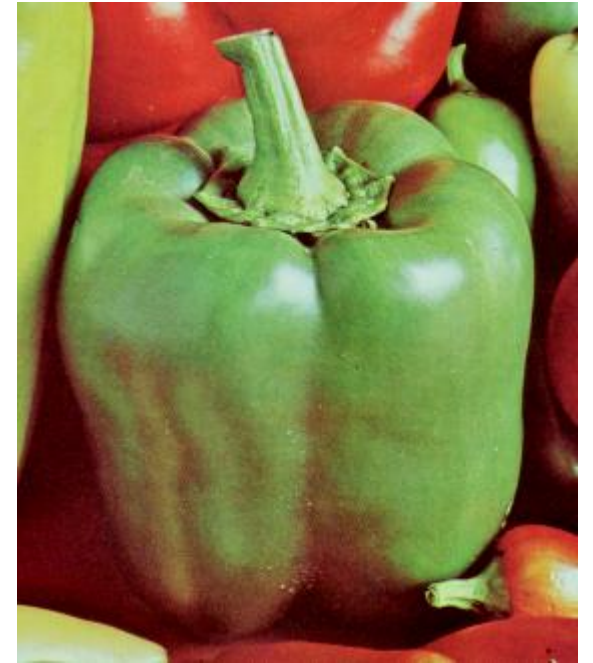
- ライブラリやモジュールが入力として想定する画像の配列の形式には2種類ある
  - HWC (高さ×幅×チャンネル)
  - CHW (チャンネル×高さ×幅)
- 複数のライブラリを組み合わせる画像処理を行う場合は、2つの形式の間の変換が必要がある
- この変換は、次の3つの関数または配列 (ndarray) のメソッドのいずれかを使う
- 例に示すように結果は同だが、過程は異なる
  
- `swapaxes()` : 指定した位置にある軸を入れ替え
- `moveaxis()` : 指定した位置にある軸を移動
- `transpose()` : 軸を指定した順序に並べ替え

# HWCからCHWへ

例として切り取り／貼り付けで切り取ったピーマンの画像の配列（HWC形式）を変換して、 $(330, 280, 3)$ を $(3, 330, 280)$ にする

```
im_hwc = im_cropped.copy() # 切り取ったピーマン  
print(im_hwc.shape)
```

$(330, 280, 3)$



# HWCからCHWへ

swapaxes()を使う場合、1番目と2番目の軸を入れ替えて(330, 3, 280)とし、さらに0番目と1番目を入れ替える

```
print(np.swapaxes(im_hwc, 1, 2).swapaxes(0, 1).shape)
```

(3, 330, 280)

moveaxis()を使う場合、2番目の軸を0番目に移動する

```
print(np.moveaxis(im_hwc, 2, 0).shape)
```

(3, 330, 280)

transpose()を使う場合、2番目、0番目、1番目の順に並べ替える

```
print(np.transpose(im_hwc, (2, 0, 1)).shape)
```

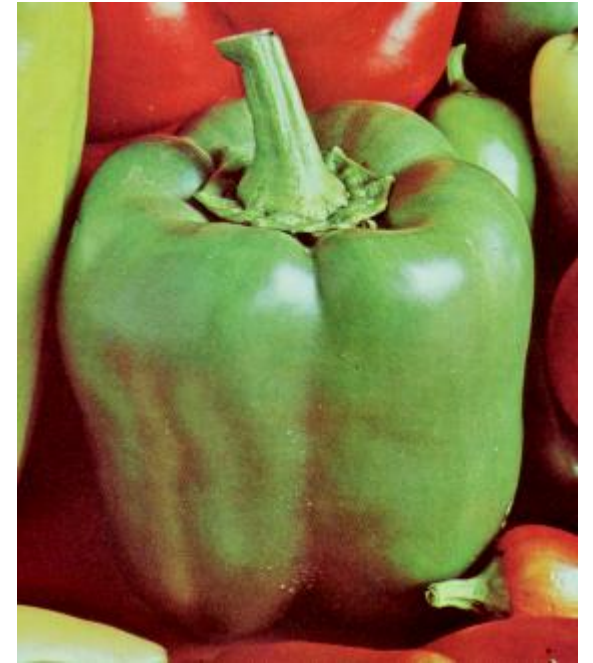
(3, 330, 280)

# CHWからHWCへ

先ほどと同じデータを使い、 $(3, 330, 280)$ を $(330, 280, 3)$ にする

```
# 準備 (CHWに変換しておく。)  
im_chw = np.transpose(im_hwc, (2, 0, 1))  
print(im_chw.shape)
```

$(3, 330, 280)$



# CHWからHWCへ

swapaxes()を使う場合、1番目と2番目の軸を入れ替えて(3, 280, 330)とし、さらに0番目と2番目を入れ替える

```
print(np.swapaxes(im_chw, 1, 2).swapaxes(0, 2).shape)
```

(330, 280, 3)

moveaxis()を使う場合、0番目の軸を2番目に移動する

```
print(np.moveaxis(im_chw, 0, 2).shape)
```

(330, 280, 3)

transpose()を使う場合、1番目、2番目、0番目の順に並べ替える

```
print(np.transpose(im_chw, (1, 2, 0)).shape)
```

(330, 280, 3)

# 色（チャネル）の積み重ね／変換

- 画像のNumPy配列は3次元だが、これをRGBの色（チャネル）ごとに2次元配列に分けて処理することがある
- 処理後の3つの2次元配列を1つの3次元配列にまとめるにはstack()関数を使い、引数axisに2を指定する
- なお、stack()関数は新しい軸を追加しそれに沿って積み重ねる

```
# チャネルごとに分割した配列から元の配列を再構成
im_stacked = np.stack(
    [
        im_array[:, :, 0],
        im_array[:, :, 1],
        im_array[:, :, 2],
    ],
    axis=2,
)
print(np.array_equal(im_stacked, im_array))
```

True

# 色（チャネル）の積み重ね／変換

- 色（チャネル）の順序には、RGB（赤、緑、青）とBGR（青、緑、赤）の2種類があり、場合によってそれらを互いに変換する必要がある
- NumPyだけでも処理ができ、OpenCVのcvtColor()関数も使える
- cvtColor()関数は、引数codeに対して下記を渡す
  - BGRへ変換する場合はcv.COLOR\_RGB2BGR
  - RGBへ変換する場合はcv.COLOR\_BGR2RGB

RGBからBGRへの変換は、次の2つのコードのように行う

```
# NumPyのみ
im_rgb = im_array.copy()
im_bgr_np = im_rgb[:, :, [2, 1, 0]] # 色（チャネル）の軸を並べ替える
```

このNumPyでの処理は、結果としては逆順に並べ替えているので、`im_rgb[:, :, ::-1]`または`np.flip(im_rgb, axis=2)`としても同じ結果を得る

```
# OpenCVのcvtColor関数
im_bgr_cv = cv.cvtColor(im_rgb, code=cv.COLOR_RGB2BGR)
print(np.array_equal(im_bgr_np, im_bgr_cv)) # NumPyと同じ結果であることを確認
```



# 色（チャネル）の積み重ね／変換

BGRからRGBへの変換は、次の2つのコードのように行う

```
# NumPyのみ
im_rgb_np = im_bgr_np[:, :, [2, 1, 0]] # 色（チャネル）の軸を並べ替える
print(np.array_equal(im_rgb_np, im_rgb)) # 元に戻ったことを確認
```

```
# OpenCVのcvtColor関数
im_rgb_cv = cv.cvtColor(im_bgr_cv, code=cv.COLOR_BGR2RGB)
print(np.array_equal(im_rgb_cv, im_rgb)) # 元に戻ったことを確認
```