

# CMPUT 206 W21

Getting Started with Colab, Numpy and Scikit-image



UNIVERSITY OF  
ALBERTA

# Lab Assignments

- There will be **no official labs** at the designated times (neither in-person nor remote)
- There are **6 lab assignments**, and each will be due **2 weeks** after it is released.
- Each assignment will be released with an explanatory video by the TA.
- All submissions must be uploaded on eclass
- You are allowed a **maximum of 4 days / 96 hours** for late submissions over all the assignments combined.
  - late hours used by each student will be tracked and any assignments submitted after this maximum will receive no credit.
- Students need to **write their own code**.
- TAs may ask students randomly to explain their code, procedure, etc.
  - Any students that are unable to provide satisfactory explanations will not get any credit for it.

# Documentation

- Each assignment will have 2/3 of its marks for the code and remaining 1/3 for the accompanying documentation
- You have the option to provide either in-code documentation as comments or submit a separate PDF file structured like a report (or both)
- Documentation needs to reflect both:
  - high-level purpose of every function and class you write and how it is accomplished
  - details of every low-level library function call used to implement this functionality within the function or class.
- It also needs to document, in as much detail as possible, all the online research or offline discussions you needed to do to come up with your solution
  - Citations are required to any websites / discussion forums / in-person discussions you used
- Documentation must make it **perfectly clear to the marking TA that your submission is your own work** and not plagiarized
  - Copying code snippets from online sources is allowed if it is cited
  - Copying code or documentation from other students is **not allowed**
- More details along with sample documentation will be posted on eclass

# TA Meetings

- There will be **4 zoom meetings** with the TA scheduled for each assignment – one towards the end of the week it is released and three in the following week
- There will be a **discussion forum** corresponding to each meeting where you'll be required to post any questions that might potentially be discussed in the meeting.
- The TAs will try their best to resolve all questions on the forum itself but if any question remains unresolved by the meeting, it will be discussed during the meeting itself.
- Anyone can attend any meeting but **only questions marked as unresolved** on the forum will be answered during the meeting
- Each forum will allow a **maximum of 10 unresolved questions** after which it will be locked for further posts.

# TA Meetings – Sharing Screen

- Any request to share your screen during a meeting needs to be justified in the forum before it is allowed
- We'd like to reduce redundancy by allowing multiple students to attend each discussion which cannot be done if someone is sharing their screen
- If you need to show some part of the provided code or instructions:
  - please post a screenshot of the same on the forum instead
- If you need to share your own code for help with debugging:
  - Code level debugging will **not be allowed** during TA meetings.
  - You'll instead need to ask more general questions about the libraries or errors involved that may help you debug your code yourself.

# Contents

- Google Colab
- Python Basics
- Numpy
- Scikit-image

# Google Colab

# Google Colab

- Cloud based runtime environment for executing Python code
- Supports GPU, TPU and CPU acceleration
- VM like environment - allows installing python (pip) and non-python (apt) packages
- Jupyter Notebooks
  - Interactive python wrapper
  - Cells – code and markdown text
- Real time collaboration just like Google Docs
- Overview



# Google Colab - Overview

# Google Colab – Data Handling

# Python Basics

# Basic data types

Like most languages, Python has a number of basic types including integers, floats, booleans, and strings. These data types behave in ways that are familiar from other programming languages.

**Numbers:** Integers and floats work as you would expect from other languages:

```
x = 3
print(type(x)) # Prints "<class 'int'>"
print(x)       # Prints "3"
print(x + 1)    # Addition; prints "4"
print(x - 1)    # Subtraction; prints "2"
print(x * 2)    # Multiplication; prints "6"
print(x ** 2)   # Exponentiation; prints "9"
x += 1
print(x) # Prints "4"
x *= 2
print(x) # Prints "8"
y = 2.5
print(type(y)) # Prints "<class 'float'>"
print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
```

# Basic data types

**Booleans:** Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols (`&&`, `||`, etc.):

```
t = True
f = False
print(type(t)) # Prints "<class 'bool'>"
print(t and f) # Logical AND; prints "False"
print(t or f)  # Logical OR; prints "True"
print(not t)   # Logical NOT; prints "False"
print(t != f)  # Logical XOR; prints "True"
```

# Basic data types

**Strings:** Python has great support for strings:

```
hello = 'hello'    # String literals can use single quotes
world = "world"    # or double quotes; it does not matter.
print(hello)       # Prints "hello"
print(len(hello))  # String length; prints "5"
hw = hello + ' ' + world # String concatenation
print(hw)          # prints "hello world"
hw12 = '%s %s %d' % (hello, world, 12) # sprintf style string formatting
print(hw12)        # prints "hello world 12"
```

String objects have a bunch of useful methods; for example:

```
s = "hello"
print(s.capitalize()) # Capitalize a string; prints "Hello"
print(s.upper())      # Convert a string to uppercase; prints "HELLO"
print(s.rjust(7))     # Right-justify a string, padding with spaces; prints "  hello"
print(s.center(7))    # Center a string, padding with spaces; prints "  hello  "
print(s.replace('l', '(ell)')) # Replace all instances of one substring with another;
                                # prints "he(ell)(ell)o"
print('  world '.strip()) # Strip leading and trailing whitespace; prints "world"
```

# String formatting

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    # old way
    print('A %s has %d legs' % (animal, legs))
    # newer way
    print('A {} has {:d} legs'.format(animal, legs))
    # newest way
    print(f'A {animal} has {legs} legs')

print('%.2f' % 0.25678) //0.26
print('%4d' % 2) //0002
print('%4i' % 10) //0010

print('{:.2f}'.format(0.25678)) //0.26
print('{:-4d}'.format(2)) // 2
print('{:04d}'.format(10)) //0010
```

# Basic data types

## Exercise



# Containers - List

A list is the Python equivalent of an array, but is resizable and can contain elements of different types:

```
xs = [3, 1, 2]      # Create a list
print(xs, xs[2])    # Prints "[3, 1, 2] 2"
print(xs[-1])       # Negative indices count from the end of the list; prints "2"
xs[2] = 'foo'       # Lists can contain elements of different types
print(xs)           # Prints "[3, 1, 'foo']"
xs.append('bar')     # Add a new element to the end of the list
print(xs)           # Prints "[3, 1, 'foo', 'bar']"
x = xs.pop()        # Remove and return the last element of the list
print(x, xs)        # Prints "bar [3, 1, 'foo']"
```

# Containers – List Slicing

**Slicing:** In addition to accessing list elements one at a time, Python provides concise syntax to access sublists; this is known as *slicing*.

```
nums = list(range(5))      # range is a built-in function that creates a list of integers
print(nums)               # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])          # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"
print(nums[2:])            # Get a slice from index 2 to the end; prints "[2, 3, 4]"
print(nums[:2])           # Get a slice from the start to index 2 (exclusive); prints "[0, 1]"
print(nums[:])            # Get a slice of the whole list; prints "[0, 1, 2, 3, 4]"
print(nums[:-1])          # Slice indices can be negative; prints "[0, 1, 2, 3]"
nums[2:4] = [8, 9]        # Assign a new sublist to a slice
print(nums)               # Prints "[0, 1, 8, 9, 4]"
```

Syntax:  
**a[start:end:step]**

Num skip between elements

Inclusive Start from 0

exclusive

Can be negative or omitted

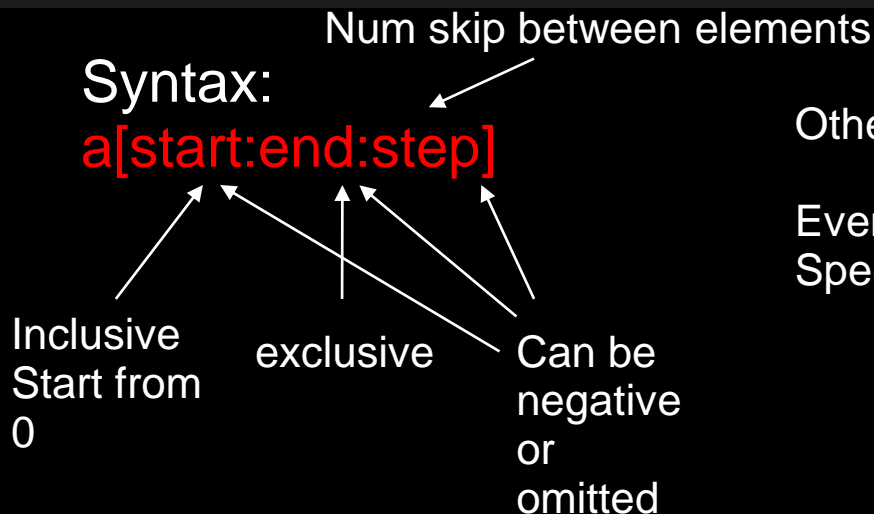
**Negative start, end:** returns the  $n^{\text{th}}$  element from the right-hand side of the list

**Omitted:** start = 0, end = len(a), step=1

# Containers – List Slicing

**Slicing:** In addition to accessing list elements one at a time, Python provides concise syntax to access sublists; this is known as *slicing*.

```
nums = list(range(5))      # range is a built-in function that creates a list of integers
print(nums)               # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])          # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"
print(nums[2:])            # Get a slice from index 2 to the end; prints "[2, 3, 4]"
print(nums[:2])           # Get a slice from the start to index 2 (exclusive); prints "[0, 1]"
print(nums[:])            # Get a slice of the whole list; prints "[0, 1, 2, 3, 4]"
print(nums[:-1])          # Slice indices can be negative; prints "[0, 1, 2, 3]"
nums[2:4] = [8, 9]        # Assign a new sublist to a slice
print(nums)               # Prints "[0, 1, 8, 9, 4]"
```



Other examples:

Even index elements: `a[::2]`

Special case - reverse list: `a[::-1]`

# Containers – List Looping

**Loops:** You can loop over the elements of a list like this:

```
animals = ['cat', 'dog', 'monkey']  
for animal in animals:  
    print(animal)  
# Prints "cat", "dog", "monkey", each on its own line.
```

If you want access to the index of each element within the body of a loop, use the built-in `enumerate` function:

```
animals = ['cat', 'dog', 'monkey']  
for idx, animal in enumerate(animals):  
    print('#%d: %s' % (idx + 1, animal))  
# Prints "#1: cat", "#2: dog", "#3: monkey", each on its own line
```

# Containers – List Comprehension

**List comprehensions:** When programming, frequently we want to transform one type of data into another. As a simple example, consider the following code that computes square numbers:

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)  # Prints [0, 1, 4, 9, 16]
```

You can make this code simpler using a **list comprehension**:

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)  # Prints [0, 1, 4, 9, 16]
```

List comprehensions can also contain conditions:

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)  # Prints "[0, 4, 16]"
```



# Triangle

Write python code to generate triangle of stars in increasing order.

```
*  
**  
***  
****  
*****  
*****  
*****  
*****  
*****  
*****
```

```
print('*\n**\n***\n****\n*****\n*****\n*****\n*****\n*****\n*****')
```



# Triangle

Write python code to generate triangle of stars in increasing order.

```
*  
**  
***  
****  
*****  
*****  
*****  
*****  
*****  
*****  
*****
```

```
n = 100  
print( '\n'.join( '*' * i for i in range(1,n) ) );
```

# Containers – List

## Exercise



# Container - Others

- Dictionary: Stores (key, value) pairs. A map from keys to values
- Set: unordered collection of distinct elements
- Tuple: immutable ordered list of values

# Functions and Classes

The syntax for defining classes in Python is straightforward:

```
class Greeter(object):

    # Constructor
    def __init__(self, name):
        self.name = name # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, %s!' % self.name.upper())
        else:
            print('Hello, %s' % self.name)

g = Greeter('Fred') # Construct an instance of the Greeter class
g.greet()           # Call an instance method; prints "Hello, Fred"
g.greet(loud=True)  # Call an instance method; prints "HELLO, FRED!"
```

# Functions and Classes

Python functions are defined using the `def` keyword. For example:

```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))
# Prints "negative", "zero", "positive"
```

We will often define functions to take optional keyword arguments, like this:

```
def hello(name, loud=False):
    if loud:
        print('HELLO, %s!' % name.upper())
    else:
        print('Hello, %s' % name)

hello('Bob') # Prints "Hello, Bob"
hello('Fred', loud=True) # Prints "HELLO, FRED!"
```

# Numpy

# Array

- Grid of values, all of the same type
- The number of dimensions is the rank of the array
- The shape of an array is a tuple of integers giving the size of the array along each dimension.

# Array – Array Creation

- Initialize numpy array from list

```
a = np.array([1, 2, 3], dtype=np.float32)
```



```
import numpy as np
```

- 2D, 3D,... array from nested list:

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

# Numpy data types

- *np.uint8*
- *np.int32*
- *np.int64*
- *np.float16*
- *np.float32*
- *np.float64*
- ...

# Array – Array Creation

- Other:
  - *np.zeros((4, 4), dtype=np.uint8)*
  - *np.ones*
  - *np.full((4, 4), 5)*
  - *np.eye*
  - *np.random.random((4, 4))*
  - *np.ones\_like*
  - *np.zeros\_like(x)*
  - *np.full\_like*



# Array Indexing – Integer Indexing

- Access elements using square bracket:

```
import numpy as np

a = np.array([1, 2, 3])    # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]]) # Create a rank 2 array
print(b.shape)            # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0]) # Prints "1 2 4"
```

# Array Indexing - Integer indexing

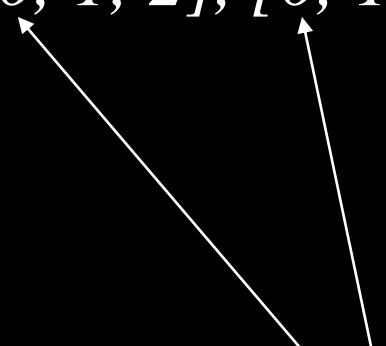
You can specify which element of array A to access using 2, 3, 4 one dimensional arrays depending whether A is 2D, 3D, 4D... array

Example:

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

```
print(a[[0, 1, 2], [0, 1, 0]])
```

```
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))
```



# Array Indexing– Boolean Indexing

- Use a boolean array B that has the same shape as array A to index array A.
- $A[B] \rightarrow$  elements in A where the same location in B equal True will be indexed
- Example: Find all element in array A that is greater than 2 and assign them to -1:
  - $A[A > 2] = -1$

# Array Indexing – Slicing

- Similar to python list
- For multidimensional array:
  - Specify a slice for each dimension
  - If a dimension is omitted, it gets all elements of that dimension
  - Rank of output array is the same as input array

# Array Indexing – Slicing

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])    # Prints "2"
b[0, 0] = 77      # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])    # Prints "77"
```

# Array Indexing – Slicing

Mixing integer indexing and slice indexing: For each integer indexing, rank of output matrix will be decreased by 1.

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape) # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape) # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape) # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape) # Prints "[[ 2]
#                               [ 6]
#                               [10]] (3, 1)"
```

Mixed indexing. Output will have shape (4)

Slice indexing. Output will have shape (1, 4)

# Array Indexing – Ellipsis (...)

- Indexing with unknown number of dimensions
- Ellipsis indicates a **placeholder** for the rest of the array dimensions not specified
- Think of it as indicating the full slice [:] for all the dimensions in the gap it is placed
  - $a[..., 0] : 3\mathbf{D} \rightarrow a[:, :, 0], 4\mathbf{D} \rightarrow a[:, :, :, 0]$
  - $a[0, ...] : 3\mathbf{D} \rightarrow a[0, :, :], 4\mathbf{D} \rightarrow a[0, :, :, :]$
  - $a[0, ..., 0] : 3\mathbf{D} \rightarrow a[0, :, 0], 4\mathbf{D} \rightarrow a[0, :, :, 0]$
  - $a[0, 1, ..., 0] : 4\mathbf{D} \rightarrow a[0, 1, :, 2], 5\mathbf{D} \rightarrow a[0, 1, :, :, 2]$
- $n\mathbf{D} \rightarrow$  However many colons in the middle make up the full number of dimensions

# Array Indexing – Exercise

- Create a  $5 \times 5$  array of random numbers between 1 and 10  $\rightarrow arr1$
- Create a  $6 \times 6 \times 3$  array with all 1, 2 and 3 in the respective channels  $\rightarrow arr2$
- Extract a  $3 \times 3$  block from the center of *arr1*
- Extract a  $3 \times 3$  block from bottom right corner of channel 2 of *arr2*
- Add the two blocks in such a way that:
  - *arr1* gets modified
  - *arr2* gets modified
  - neither gets modified



# Array Math

- $+$ ,  $-$ ,  $*$ ,  $/$
- $A + B$ ,  $A - B$ ,  $A * B$ ,  $A / B$
- *np.add(A, B)*, *np.subtract*, *np.multiply*, *np.divide*
- All are element-wise operations
- For matrix multiplication, use *np.dot* or *np.matmul*:
  - *A.dot(B)*
  - *np.dot(A, B)*
  - *np.matmul(A, B)*

# Array Math

- Other unary operations: sum, max, min, transpose...
- Can specify axis for some operations

Numpy provides many useful functions for performing computations on arrays; one of the most useful is `sum`:

```
import numpy as np

x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)      # Prints "[[1 2]
               #           [3 4]]"
print(x.T)    # Prints "[[1 3]
               #           [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)      # Prints "[1 2 3]"
print(v.T)    # Prints "[1 2 3]"
```

# Array Broadcasting

- Powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be *compatible* in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

# Array Broadcasting

- Example 1:
  - *image* = *np.random.random*((400, 400, 3))
  - *means* = *np.asarray*([0.1, 0.2, 0.3])
  - *print*(*image* - *means*)
  - Shape of *image*: (400, 400, 3)
  - Shape of *means*: (1, 400, 1)  $\leftarrow$  (3)
  - Broadcasted *means*: (400, 400, 3)
  - Broadcasted *means* will have
    - [:, :, 0] = 0.1
    - [:, :, 1] = 0.2
    - [:, :, 3] = 0.3

# Array Broadcasting

- Example 2:

```
>>> x = np.array([[1, 1, 1]])
>>> y = np.array([[1], [1], [1]])
>>> x.shape
(1, 3)
>>> y.shape
(3, 1)
>>> x - y
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

Be careful! This can cause unnoticeable bugs in your code

# Array Broadcasting – Exercise

- Create a  $5 \times 1$  array of random numbers  $\rightarrow arr3$
- Add *arr3* to each column of *arr1*
- Subtract *arr3* from each row of *arr1*
- Reset *arr2* to its original state in a single statement
- Create a  $3 \times 3$  array of all 2s  $\rightarrow arr4$
- Use *arr4* to add 2, 4, 6 and 8 respectively to the top left, top right, bottom right and bottom left  $3 \times 3$  corners of *arr2*

# Scikit-image

# Image operations - Read, Write, Resize

```
from skimage.io import imread, imsave
```

```
from skimage.transform import resize
```

```
im = imread('image.jpg')
```

```
im_resized = resize(im, output_shape=(500, 500))
```

```
imsave("im_resized.jpg", im_resized)
```



# Image operations - Show

```
import matplotlib.pyplot as plt  
from skimage.io import imshow  
plt.imshow(im)  
imshow(im)
```

# Image operations - Filter

```
from skimage.filters import sobel, gaussian, median  
im_sobel = sobel(im)  
im_gauss = gaussian(im, sigma=3)  
im_median = median(im)
```

# Image operations - Exercise

- Download an image from internet or use an existing image
- Upload it to Google Drive
- Mount your Google Drive and authorize
- List the contents of your Google Drive to figure out the path to that image
- Read that image and show it
- Resize it to double its original size
- Apply Sobel filtering to the resized image
- Show the resized and filtered images
- Save the resized and filtered images to Google Drive
- Download the image and open it locally

**Thanks !**