COMP2211 Exploring Artificial Intelligence

Advanced Python for Artificial Intelligence

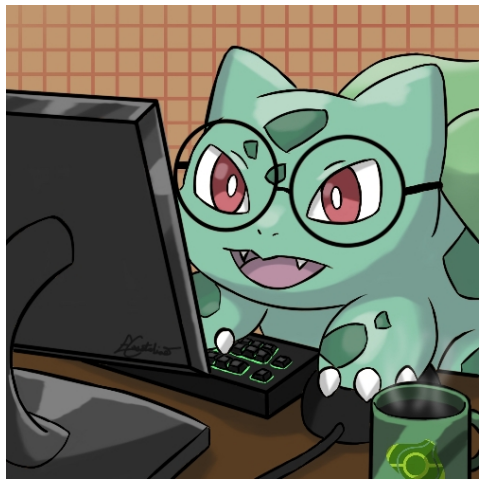
Huiru Xiao

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology

Why More Python?

You have taken COMP 1021/COMP 1029P. So you can program in Python, right?

- Think about this: You have been learned Python (mostly Turtle) in COMP1021, but can you write AI programs?
- You have learned the basics of Python in COMP1021/COMP1029P with a brief introduction to Python classes, and you can write small Python programs.
- In this lecture, we will give you a crash course on more advanced Python programming elements and other essentials for writing AI programs.



More Advanced Python Programming Elements

Python Versions

- As of January 1, 2020, Python has officially dropped support for Python 2.
- For this course, all code will use Python 3.11.13.
- The latest version is Python 3.15.4 (released on 11 June 2025).
- You can check your Python version at the command line after activating your environment by running

```
import sys
print(sys.version)
```

Data Types and type Function

- Like most languages, Python has a number of basic types including integers, floats, booleans, strings, and containers including lists, dictionaries, sets, tuples.
- type() function returns class type of the argument(object) passed as parameter.
- type() function is mostly used for debugging purposes.

Syntax

```
type(<object>)
```

```
x = 3
print(type(x))     # Print "<class 'int'>"
y = 3.14
print(type(y))     # Print "<class 'float'>"
z = True
print(type(z))     # Print "<class 'bool'>"
str = "COMP2211"
print(type(str))   # Print "<class 'str'>"
l = [1, 2, 3]
print(type(l))     # Print "<class 'list'>"
d = { 'cat': 'cute', 'dog' : 'furry' }
print(type(d))     # Print "<class 'dict'>"
s = { 1, 2, 3 }
print(type(s))     # Print "<class 'set'>"
t = ( 1, 2, 3 )
print(type(t))     # Print "<class 'tuple'>"
```

List Comprehensions

- List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Syntax

```
<new-list-name> = [ <expression> for <element> in <list-name>
                    if <condition> ]
```

- List comprehensions start and end with opening and closing square brackets.
- Parameters:
  - expression: Operation we perform on each value inside the original list
  - element: A temporary variable we use to store for each item in the original list
  - list-name: The name of the list that we want to go through
  - condition: If condition evaluates to True, add the processed element to the new list
  - new-list-name: New values created which are saved

Note: There can be an optional if statement and additional for clause.

List Comprehensions

- 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]"
```

Dictionary Comprehensions

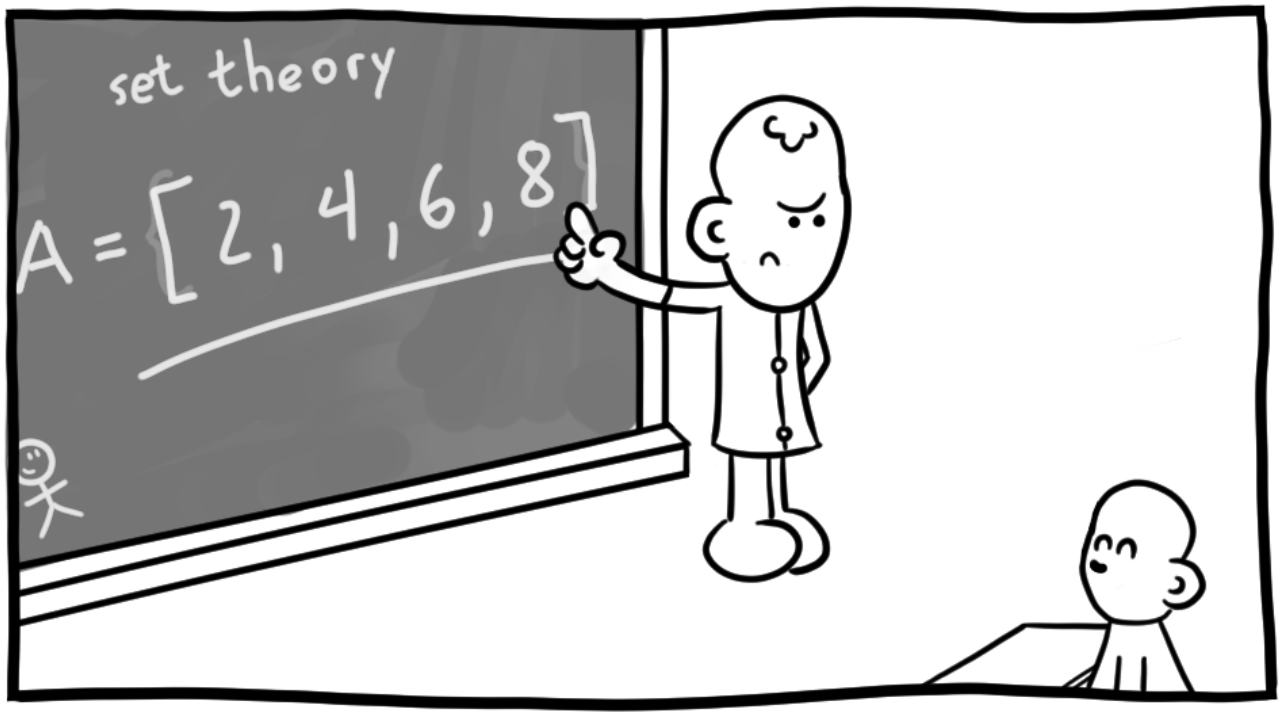
- These are similar to list comprehensions, but allow you to easily construct dictionaries. For example:

```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square)  # Print "{0: 0, 2: 4, 4: 16}"
```

Set Comprehensions

- Like lists and dictionaries, we can easily construct sets using set comprehensions:

```
from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums)  # Print "{0, 1, 2, 3, 4, 5}"
```



Difference between Lists and Tuples

- A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot.
- Here is a trivial example:

```
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tupl
t = (5, 6)          # Create a tuple
print(type(t))      # Print "<class 'tuple'>"
print(d[t])         # Print "5"
print(d[(1, 2)])    # Print "1"
```

More Details About Tuples

https://docs.python.org/3.10/tutorial/datastructures.html#tuples-and-sequences

Parallel Iteration

- The zip() method takes iterable or containers and returns a single iterator object, having mapped values from all the containers.
- If the passed iterable or containers have different lengths, the one with the least items decides the length of the new iterator.

Syntax

```
zip(<iterator1>, <iterator2>, <iterator3>...)
```

- Parameters:
  - iterator1, iterator2, iterator3, $\cdots$: Iterator objects that will be joined together

Parallel Iteration

```
fruits = ['apple', 'peach', 'banana', 'guava', 'papaya']
colors = ['red', 'pink', 'yellow', 'green', 'orange']

for name, color in zip(fruits, colors):
    print(name, 'is', color)

# It prints
# apple is red
# peach is pink
# banana is yellow
# guava is green
# papaya is orange
```

Functions

- If you do not know how many arguments that will be passed into your function, add an asterisk (i.e., *) before the parameter name in the function definition. This way the function will receive a tuple of arguments, and can access the items accordingly:

```
def print_months(*months):
  print("The last month is " + months[-1])

# Print "The last month is December"
print_months("January", "Febuary", "March", "April", "May", "June",
             "July", "August", "September", "October",
             "November", "December")
```

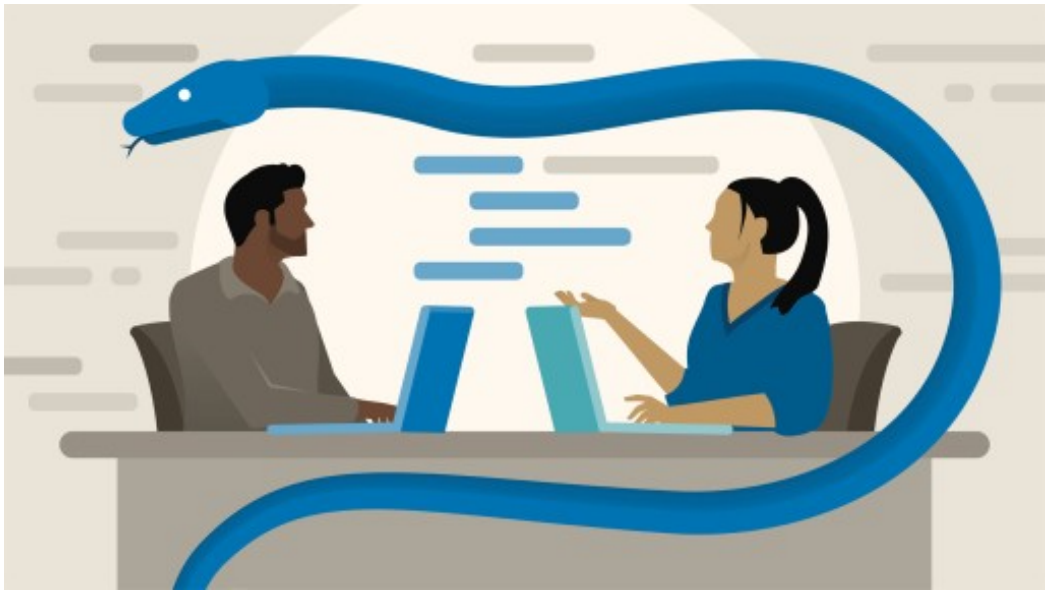
Functions

- If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition. This way the function will receive a dictionary of arguments, and can access the items accordingly:

```
def print_month(**month):
  print("The month is " + month["name"] + ", it is the " +
        month["order"] + "th month, it has " + month["days"] +
        " days.")

# Print "The month is September, it is the 9th month, it has 30 days."
print_month(name = "September", order = "9", days = "30")
```

Object-Oriented Programming

- Object-Oriented Programming (OOP) is a way of structuring a program by bundling related properties (data) and behaviors (functionality) into individual objects.
- The fundamental concepts of OOP can be introduced via the use of class and objects.



Classes

- A class is a user-defined data type from which objects are created. It is created by keyword class.
- Class provides a means of bundling data (i.e., instance variables) and functionality (i.e., methods) together.
- Instance variables:
  - They are the variables that belong to an object.
  - They are always public by default and can be accessed using the dot (.) operator.
- Methods:
  - They have an extra first parameter, self, in the method definition.
  - We do not give a value for this parameter when we call the method, Python provides it.
- The instance variables and methods are accessed by using the object.
- `__init__` method is similar to constructors in Java/C++. Constructors are used to initializing the instance variables of objects.

Classes

- Define a class with instance variables and methods (and a constructor).

Syntax

```
class <class-name>:
  # __init__ is a constructor
  def __init__(self, <arguments0>):
    self.<instance-variable1> = <value1>
    self.<instance-variable2> = <value2>
    ...
  def <method1-name>(self, <arguments1>):
    <statement1>
    <statement2>
    ...
  def <method2-name>(self, <arguments2>):
    <statement1>
    <statement2>
    ...
```

- Parameters:
  - class-name: The name of the class.
  - arguments0, arguments1, arguments2, ...: Method parameters (i.e., variables)
  - instance-variable1, instance-variable2, ...: Instance variables
  - value1, value2, ...: Value assigned to the instance variables
  - statement1, statement2, ...: Python statements

Classes

- Create an object, call a method, and modify an instance variable.

Syntax

```
<object-name> = <class-name>(<arguments>)          # Create an object
<object-name>.<method-name>(<arguments>)           # Call a method
<object-name>.<instance-variable-name> = <value>   # Modify an instance
                                                   # variable
```

- Parameters:
  - object-name: The name of an object
  - class-name: The name of a class
  - arguments: The values that we pass to the constructor/method
  - method-name: The name of the method
  - instance-variable-name: The name of an instance variable
  - value: The value assigned to the instance variable

Example

```
import math                                 # Import math library

class Circle:                               # Define a new type Circle
  def __init__(self, radius):               # Define a constructor with parameter,
                                            # radius
    self.radius = radius                    # Define an instance variable radius
                                            # and assign it with parameter radius
  def area(self):                           # Define the method: area
    return math.pi * self.radius**2         # Compute the area of circle

  def circumference(self):                  # Define the method: circumference
    return 2 * math.pi * self.radius # Compute the circumference of circle

circle_object = Circle(10) # Define an object of Circle named circle_object
circle_object.radius = 100 # Modify circle_object's radius to 100
print("Area:", circle_object.area())                    # Call area()
print("Circumference:", circle_object.circumference()) # Call circumference
```

```
Area: 31415.926535897932
Circumference: 628.3185307179587
```

Classes: Public, Private and Protected Members

- As mentioned, all members in a Python class are public by default, i.e., any member can be accessed from outside the class. To restrict the access to the members, we can make them protected/private.
- Protected members of a class are accessible from within the class and also available to its sub-classes. By convention, Python makes an/a instance variable/method protected by adding a prefix _ (single underscore) to it.
- Private members of a class are accessible from with the class only. By convention, Python makes an instance variable/method private by adding a prefix __ (double underscore) to it.

Note

"By convention" means the responsible programmer would refrain from accessing and modifying instance variables prefixed with _ or __ from outside its class. But if they do so, still works. :(

Example: Access Private Instance Variables Outside Class (Error!!!)

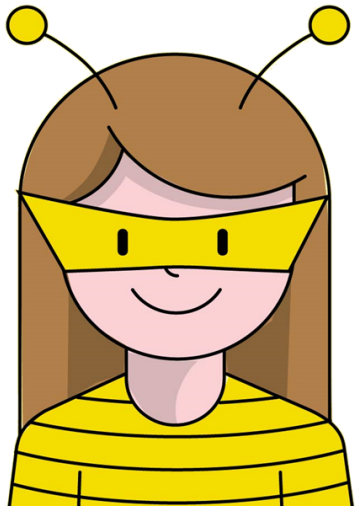
```
class Pokemon:
  def __init__(self, name='Pikachu',
               Ndex='0025',
               types='Electric'):
    # private instance variables
    self.__name = name
    self.__Ndex = Ndex
    self.__types = types
  def print(self):
    print('--- Print Pokemon ---')
    print('Name: ' + self.__name)
    print('Ndex: ' + self.__Ndex)
    print('Type: ' + self.__types)
```

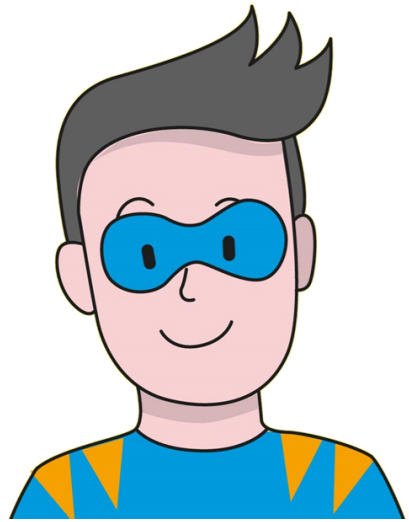
```
vulpix = Pokemon('Vulpix', '0037',
                 'Fire')

# Error
print('Name: ' + vulpix.__name)
print('Ndex: ' + vulpix.__Ndex)
print('Type: ' + vulpix.__types)

# Okay, but do not work
vulpix.__name = 'AlolanVulpix'
vulpix.__types = 'Ice'

vulpix.print()   # Okay
```

How to retrieve and modify instance variable values? We need to define accessors and mutators!

Accessors and Mutators

- Accessors and mutator methods are used to access the protected/private instance variables which cannot be accessed from outside the class.
- Accessor methods (or getters) are used to retrieve the values of instance variables of an object. When the accessor method is called, it returns the private instance variable value of the object.
- Mutator methods (or setters) are used to modify the values of instance variables of an object. When the mutator method is called, it modifies the private instance variable value of the object.





Example: Add Accessors and Mutators

```
class Pokemon:
  def __init__(self, name='Pikachu',
               Ndex='0025',
               types='Electric'):
    self.__name = name
    self.__Ndex = Ndex
    self.__types = types

  def get_name(self): # Accessor
    return self.__name

  def get_Ndex(self): # Accessor
    return self.__Ndex

  def get_types(self): # Accessor
    return self.__types

  def set_name(self, name): # Mutator
    self.__name = name

  def set_Ndex(self, Ndex): # Mutator
    self.__Ndex = Ndex

  def set_types(self, types): # Mutat
    self.__types = types

  def print(self):
    print('--- Print Pokemon ---')
    print('Name: ' + self.__name)
    print('Ndex: ' + self.__Ndex)
    print('Type: ' + self.__types)

vulpix = Pokemon('Vulpix', '0037',
                 'Fire')
print('Name: ' + vulpix.get_name())
print('Ndex: ' + vulpix.get_Ndex())
print('Type: ' + vulpix.get_types())
vulpix.set_name('AlolanVulpix')
vulpix.set_types('Ice')
vulpix.print()
```

Class Variables

- A class variable is a variable that is defined within a class and outside of any method.
- It is a variable that is shared by all instances of the class, meaning that if the variable's value is changed, the change will be reflected in all instances of the class.
- Class variables are accessed using the class name, followed by the class variable name.

Example

```
class Pokemon:
  num_pokemon = 0 # Class variable
  def __init__(self, name='Pikachu',
               Ndex='0025',
               types='Electric'):
    self.__name = name
    self.__Ndex = Ndex
    self.__types = types
    Pokemon.num_pokemon += 1
  def print(self):
    print('--- Print Pokemon ---')
    print('Name: ' + self.__name)
    print('Ndex: ' + self.__Ndex)
    print('Type: ' + self.__types)

vulpix = Pokemon('Vulpix', '0037', 'Fire')
togepi = Pokemon('Togepi', '0175', 'Fairy')
vulpix.print()
togepi.print()
print("Total number of pokemons:", Pokemon.num_pokemon)
```

Output:

```
--- Print Pokemon ---
Name: Vulpix
Ndex: 0037
Type: Fire
--- Print Pokemon ---
Name: Togepi
Ndex: 0175
Type: Fairy
Total number of pokemons: 2
```

Class Variables vs. Instance Variables

	Class Variables	Instance Variables
Definition	Defined within the class but outside of any methods.	Defined within methods, typically the constructor.
Scope	Changes are made to the class variables affect all instances.	Changes made to the instance variables does not affect all instances.
Initialization	Can be initialized either inside the class definition or outside the class definition	Typically initialized in the constructor of the class.
Access	Accessed using the class name, followed by the variable name.	Accessed using the instance name, followed by the variable name.
Usage	Useful for storing data that is shared among all instances of a class, such as constants or default values.	Used to store data that is unique to each instance of a class, such as object properties.

Inheritance

- Inheritance is a mechanism that allows us to define a new class based on existing classes.
- All the attributes and methods of existing classes will be inherited by thew new class.
- The existing classes are called super/parent/base classes, and the new class is called sub/child/derived class.

Syntax

```
class <subclass-name>(<superclass-name>):
  # __init__ is a constructor
  def __init__(self, <arguments0>):
    super().__init__(<attributes>)
    self.<instance-variable1> = <value1>
    self.<instance-variable2> = <value2>
    ...
  def <method1-name>(self, <arguments1>):
    <statement1>
    <statement2>
    ...
```

- Parameters:
  - subclass-name: The name of the subclass.
  - superclass-name: The name of the superclass.
  - arguments0, arguments1, ...: Method parameters (i.e., variables)
  - instance-variable1, instance-variable2, ...: Instance variables
  - value1, value2, ...: Value assigned to the instance variables
  - statement1, statement2, ...: Python statements

Pokemon Class Again

```
class Pokemon:
  def __init__(self, name='Pikachu',
               Ndex='0025',
               types='Electric'):
    self.__name = name
    self.__Ndex = Ndex
    self.__types = types

  def get_name(self):
    return self.__name

  def get_Ndex(self):
    return self.__Ndex

  def get_types(self):
    return self.__types
```

```
def set_name(self, name):
  self.__name = name

def set_Ndex(self, Ndex):
  self.__Ndex = Ndex

def set_types(self, types):
  self.__types = types

def print(self):
  print('--- Print Pokemon ---')
  print('Name: ' + self.__name)
  print('Ndex: ' + self.__Ndex)
  print('Type: ' + self.__types)
```

EvolPokemon Class super().__init__(name=name, types=types)

```
# Define a new class EvolPokemon that inherits Pokemon class
class EvolPokemon(Pokemon):
  def __init__(self, name='Pikachu', Ndex='0025',
               types='Electric', evol='Raichu'): # Constructor
    super().__init__(name, Ndex, types) # Invoke superclass' init
    self.__evol = evol # Define an instance variable evol

  def get_evol(self): return self.__evol         # Accessor

  def set_evol(self, evol): self.__evol = evol   # Mutator

  def print(self):                               # Overwrite print method
    super().print()                              # Invoke superclass' print
    print('Evolve into: ' + self.__evol)         # Print evol

# Define an object of EvolPokemon named jigglypuff and print
jigglypuff = EvolPokemon('Jigglypuff', '0039', 'Normal', 'Wigglytuff')
jigglypuff.print()
```

```
--- Print Pokemon ---
Name: Jigglypuff
Ndex: 0039
Type: Normal
Evolve into: Wigglytuff
```

Importing Libraries and Data

Modules, Packages, Libraries

Modules

A modules in Python is a bunch of related code (functions, classes, variables) saved in a file with the extension .py.

Packages

A package is basically a directory of a collection of modules.

Libraries

A library is a collection of packages.

Imports

- The import keyword is the most common way to access external modules.
- Modules are files (.py) that contain functions, classes and variables that you can reference in your program without having to write them yourself.
- This is how you get access to machine learning libraries, like `numpy`, `keras`, `sklearn`, `tensorflow`, `pandas`, `pytorch`, `matplotlib`, etc.
- import is similar to the `#include <something>` in C++.
- However, you will usually only need to import a few functions from each of those modules.
- In that case, we can use the `from` keyword to import only the exact functions/classes/variables we are using in the code.

Syntax

```
from [module] import [function/class/variable]
from [package].[module] import [function/class/variable]
```

Imports

- For instance, if we only want to import the `sqrt` function from the `math` module, we just need to do:

```
from math import sqrt # Import only sqrt function from math module
x = 100
# Now, it's imported, we can directly use sqrt to call the math.sqrt()
print(sqrt(x)) # It prints 10
```

- There is also the "`from [module] import *`" syntax which imports all functions. So the following code works as well.

```
from math import * # Import ALL functions from math module
x = 100
print(sqrt(x)) # It print 10
```

Imports

- However, we should generally avoid importing all functions since we do not know if sqrt function is from the math library. If we use the import all syntax for multiple libraries, we may end up with clashes or ambiguity, which decreases the readability of our code and makes it harder to spot errors.
- One such example:

```
# Import ALL functions from math module, including math.sqrt()
from math import *
# Import ALL functions from cmath module, including cmath.sqrt()
from cmath import *

x = 100
# The following will use cmath.sqrt(), which is a different
# implementation to math.sqrt()
print(sqrt(x)) # It print 10
```

Imports

- One of the most frequently used machine learning libraries is `sklearn`.
- For instance, you may use the `K-Means` algorithm (we will cover this later in class) to separate your data into groups.
- Let's look at the scikit-learn documentation
  `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans`
  and try to understand it.

```
class sklearn.cluster.KMeans(n_clusters=8, *, init='k-means++',
                             n_init=10,max_iter=300, tol=0.0001,
                             verbose=0, random_state=None,
                             copy_x=True, algorithm='auto')
```

Each "." refers to accessing something inside a package.

Imports

- For instance, the following is a simplified layout of sklearn:

```
sklearn/
    __init__.py
    discriminant_analysis.py
    cluster/
        KMeans.py
        SpectralClustering.py
        affinity_propagation.py
        ...
    linear_model/
        bayes.py
        logistic.py
    neural_network/
        multilayer_perceptron.py
```

Imports

- `import sklearn.cluster` means we import all the modules inside the package `sklearn.cluster`, including but not limited to (`KMeans`, `SpectralClustering`, `affinity_propagation`).
- `import sklearn.cluster.KMeans` means we only import the module `KMeans` inside the package `sklearn.cluster`. **kmeans = sklearn.cluster.KMeans()**
- If we want to call K-Means on our data, we need to call `sklean.cluster.KMeans`. But that's quite a long function call, so we typically declare the everything but the name of the actual module in the import section, like so:

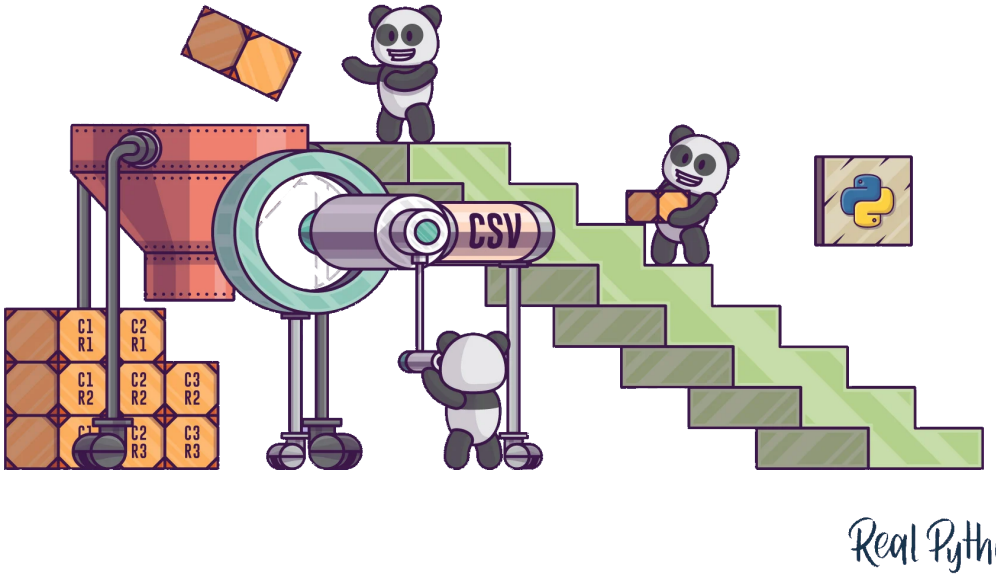
```
# Import KMeans module from package sklearn.cluster
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=2, random_state=600)
```

  Notice that we can just use the function name directly in the 2nd line as `KMeans`.
- If our function is particularly long and we do not want to type it out every time, we can shorten it by using the as keyword to create an alias (a "nickname") to use in place of the original (and long) function name.

```
from sklearn.cluster import KMeans as km
kmeans = km(n_clusters=2, random_state=600)
```

Importing and Exporting Data

- Your data can come in many forms, which are too exhaustive to list here.
- We will introduce how to deal with data in csv (comma-separated-values) form, since that is fairly common.
- In Python, most data manipulation is done with the pandas library.



Importing Data

- To import data from Google Drive, we need to "mount" the directory.
- It will ask you to visit a link to allow Colab access your drive.

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

- Copy the alphanumeric code and paste it into your notebook.
- Afterwards, Drive files will be mounted, which you can view in the sidebar to the left.
- To access file `training_data.csv` in the root "My Drive" folder, for instance:

```
import pandas as pd
df = pd.read_csv('/content/drive/My Drive/training_data.csv',
                 index_col=False)
```

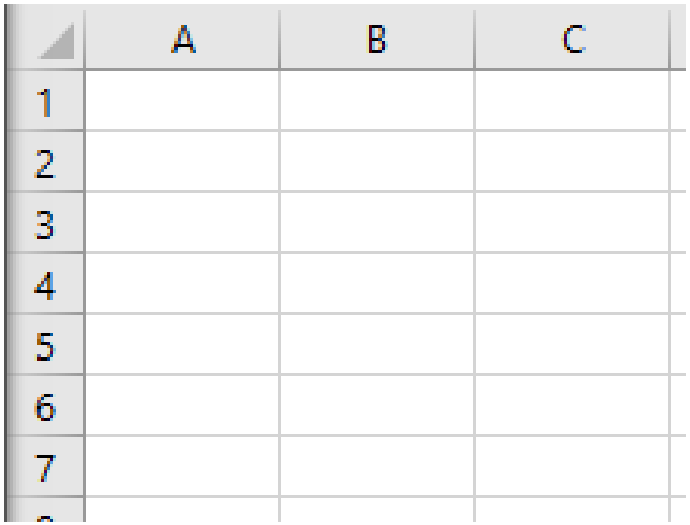
The following examples demonstrate what `index_col` means.

Importing Data - Example 1

```
from google.colab import drive
import pandas as pd
drive.mount('/content/drive')
df = pd.read_csv('/content/drive/My Drive/pokemons.csv', index_col=False)
print(df)
```

```
          Name  Ndex      Type
0      Pikachu    25  Electric
1       Vulpix    37      Fire
2       Togepi   175     Fairy
3  Jigglypuff    39    Normal
```

If we set `index_col` to False, the current columns (such as Name, Score, and Remark) will not be used as the column index. Instead, a new column will be added on the left with numbers starting from 0, 1, 2, and so on, similar to the left-most indices in Excel.



Importing Data - Example 2

```
from google.colab import drive
import pandas as pd
drive.mount('/content/drive')
df = pd.read_csv('/content/drive/My Drive/pokemons.csv', index_col=0)
print(df)
```

```
              Ndex      Type
Name
Pikachu         25  Electric
Vulpix          37      Fire
Togepi         175     Fairy
Jigglypuff      39    Normal
```

If `index_col = 0` (i.e., the column "Name") will be used as the column index.

Importing Data - Example 3

```
from google.colab import drive
import pandas as pd
drive.mount('/content/drive')
df = pd.read_csv('/content/drive/My Drive/pokemons.csv', index_col="Name")
print(df)
```

```
              Ndex      Type
Name
Pikachu         25  Electric
Vulpix          37      Fire
Togepi         175     Fairy
Jigglypuff      39    Normal
```

If `index_col = "Name"`, the column "Name" will be used as the column index.

Conclusion

We can assign False, integer (refer to which column), or string (column name) to index_col to indicate which column would be used as the column index.

Exporting Data

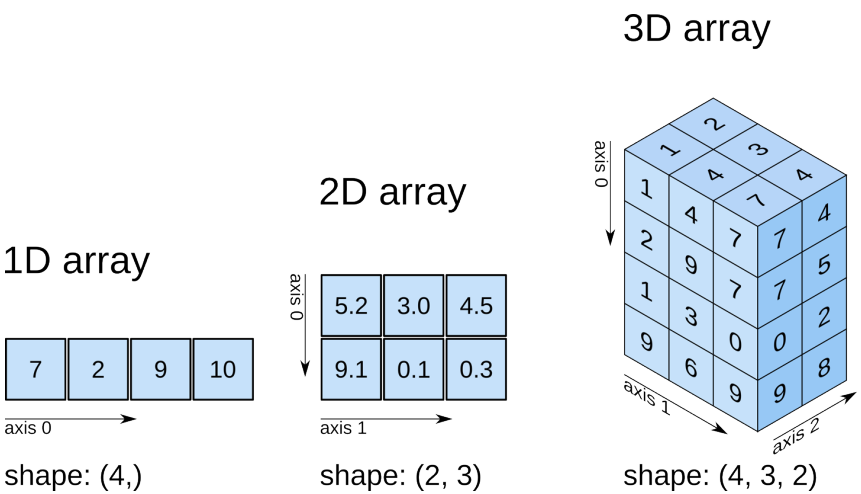
- To save a file in the root "My Drive" folder, we do

```
import pandas as pd
# Create a pandas.DataFrame object with two columns:
# 'CustomerId' and 'CanRepayLoan'
output = pd.DataFrame({'CustomerId': np.array([1, 2, 3]),
                       'CanRepayLoan': np.array([0, 0, 1])})
# Convert the pandas.DataFrame object into the csv "loans .csv" and
# Export it to the root folder of My Drive
output.to_csv('/content/drive/My Drive/loans.csv', index=False)
```

Numpy

NumPy

- NumPy is a short form for Numerical Python and is the core library for numeric and scientific computing in Python.
- It provides high-performance multidimensional array object, and tools for working with these arrays.



Arrays

- A NumPy array is a grid of values, all of the same type, and is indexed by a tuple of non-negative integers.
- 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 dimensions.
- We can initialize NumPy arrays from nested Python lists, and access elements using square brackets.

```
import numpy as np

a = np.array([1, 2, 3])                  # Create a rank 1 array
print(type(a))                           # Print "<class 'numpy.ndarray'>"
print(a.shape)                           # Print "(3,)"
print(a[0], a[1], a[2])                  # Print "1 2 3"
a[0] = 5                                 # Change an element of the array
print(a)                                 # Print "[5 2 3]"
b = np.array([[1,2,3],[4,5,6]])          # Create a rank 2 array
print(b.shape)                           # Print "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])         # Print "1 2 4"
```

Arrays

- NumPy also provides many functions to create arrays.

```
import numpy as np

a = np.zeros((2,2))           # Create an array of all zeros
print(a)                      # Print "[[ 0.  0.]
                              #         [ 0.  0.]]"
b = np.ones((1,2))            # Create an array of all ones
print(b)                      # Print "[[ 1.  1.]]"

c = np.full((2,2), 7)         # Create a constant array
print(c)                      # Print "[[ 7  7]
                              #         [ 7  7]]"
d = np.eye(2)                 # Create a 2x2 identity matrix
print(d)                      # Print "[[ 1.  0.]
                              #         [ 0.  1.]]"
e = np.random.random((2,2))   # Create an array filled with random values
print(e)                      # Might print "[[ 0.91940167  0.08143941]
                              #              [ 0.68744134  0.87236687]]"
```

Array Indexing

Expression	Description
a[i]	Select element at index i, where i is an integer (start counting from 0).
a[-i]	Select the i-th element from the end of the list, where n is an integer. The last element in the list is addressed as -1, the second to last element as -2, and so on.
a[i:j]	Select elements with indexing starting at i and ending at j-1.
a[:] or a[0:]	Select all elements in the given axis.
a[:i]	Select elements starting with index 0 and going up to index i - 1.
a[i:]	Select elements starting with index i and going up to the last element in the array.
a[i:j:n]	Select elements with index i through j (exclusive), with increment n.
a[::-1]	Select all the elements, in reverse order.

Array Indexing

- Similar to Python lists, NumPy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array.

```
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"
```

a[0, 1] = 88
print(b[0, 0]) # 88

Array Slicing and Striding

- The basic slice syntax is `i:j:k` where i is the starting index, j is the stopping index, and k is the step ($k \neq 0$).

```
import numpy as np

a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print(a[1: 7: 2]) # Prints [1 3 5]
```

- Negative i and j are interpreted as $n + i$ and $n + j$ where n is the number of elements in the corresponding dimension. Negative k makes stepping go towards smaller indices.

```
print(a[-2: 10])    # Prints [8 9]
print(a[-3: 3: -1]) # Prints [7 6 5 4]
```

- More challenging examples:

```
print(a[1: 100: 2])    # Prints [1 3 5 7 9]
print(a[4: 3: 2])      # Prints []
print(a[100: 200: 2])  # Prints []
print(a[: : -2])       # Prints [9 7 5 3 1]
print(a[0: -1: -2])    # Prints [9 7 5 3 1]
print(a[0: : -2])      # Prints [0]
```

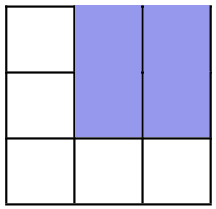
This is a mistake!!! It should print []

Array Indexing

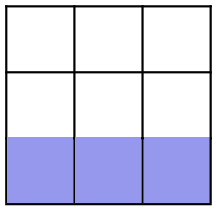
- You can also mix integer indexing with slice indexing.
- However, doing so will yield an array of lower rank than the original array.

```
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]])

# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of 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)"
```

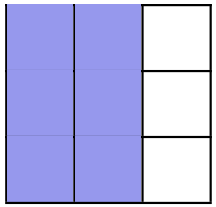
Arrays



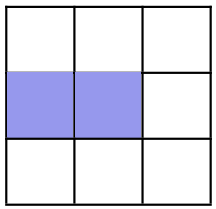
Expression	Shape
arr[:2, 1:]	(2, 2)



Expression	Shape
arr[2]	(3,)
arr[2, :]	(3,)
arr[2:, :]	(1, 3)



Expression	Shape
arr[:, :2]	(3, 2)



Expression	Shape
arr[1, :2]	(2,)
arr[1:2, :2]	(1, 2)

Integer Array Indexing

- Integer array indexing allows you to construct arbitrary arrays using the data from another array.

```
import numpy as np

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

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

Integer Array Indexing

- One useful trick with integer indexing is selecting or mutating one element from each row of a matrix.

```
import numpy as np

# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

print(a)  # prints [[ 1  2  3]
          #         [ 4  5  6]
          #         [ 7  8  9]
          #         [10 11 12]]"

# Create an array of indices
b = np.array([0, 2, 0, 1])
# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"
# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10

print(a)  # prints [[11  2  3]
          #         [ 4  5 16]
          #         [17  8  9]
          #         [10 21 12]])"
```

Boolean Array Indexing

- Boolean array indexing lets you pick out arbitrary elements of an array.
- Frequently this type of indexing is used to select the elements of an array that satisfy some condition.

```
import numpy as np

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

bool_idx = (a > 2)   # Find the elements of a that are bigger than 2; this returns a numpy
                     # array of Booleans of the same shape as a, where each slot of bool_idx
                     # whether that element of a is > 2.

print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array consisting of the elements of a
# corresponding to the True values of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

Data Types

- Every NumPy array is a grid of elements of the same type.
- NumPy provides a large set of numeric data types that you can use to construct arrays.
- NumPy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype.

```
import numpy as np

x = np.array([1, 2])   # Let numpy choose the datatype
print(x.dtype)         # Prints "int64"

x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

- Note the difference between Python data type and Numpy data type!

```
x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
print(type(x))                         # Prints "<class 'numpy.ndarray'>"
```

Data Types

Category	Data Type	Description
Numeric	`np.int8`	8-bit signed integer
	`np.int16`	16-bit signed integer
	`np.int32`	32-bit signed integer
	`np.int64`	64-bit signed integer
Unsigned Integer	`np.uint8`	8-bit unsigned integer
	`np.uint16`	16-bit unsigned integer
	`np.uint32`	32-bit unsigned integer
	`np.uint64`	64-bit unsigned integer
Floating Point	`np.float16`	16-bit floating point
	`np.float32`	32-bit floating point
	`np.float64`	64-bit floating point
	`np.float128`	128-bit floating point
Complex	`np.complex64`	64-bit complex number
	`np.complex128`	128-bit complex number
	`np.complex256`	256-bit complex number
Boolean	`np.bool_`	Boolean type (True/False)
String	`np.str_`	Variable-length string
	`np.bytes_`	Byte string
Object	`np.object_`	General-purpose type for arbitrary objects
Datetime	`np.datetime64`	Represents dates and times
	`np.timedelta64`	Represents time differences
Void	`np.void`	Represents flexible type

Array Math

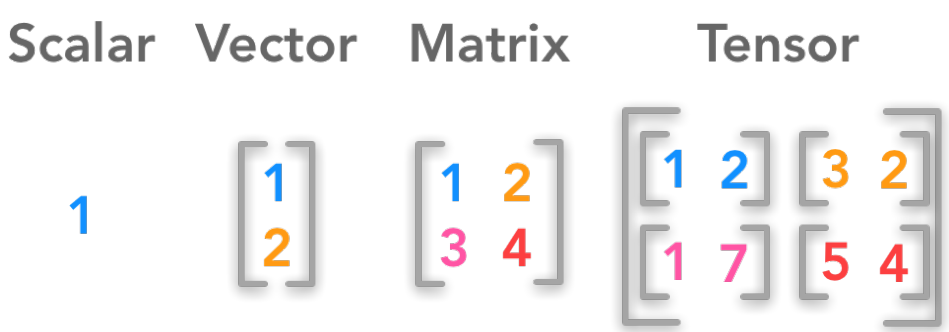
- Basic mathematical functions operate element-wise on arrays, and are available both as operator overloads and as functions in the NumPy module.

```python
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))
```

```python
# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

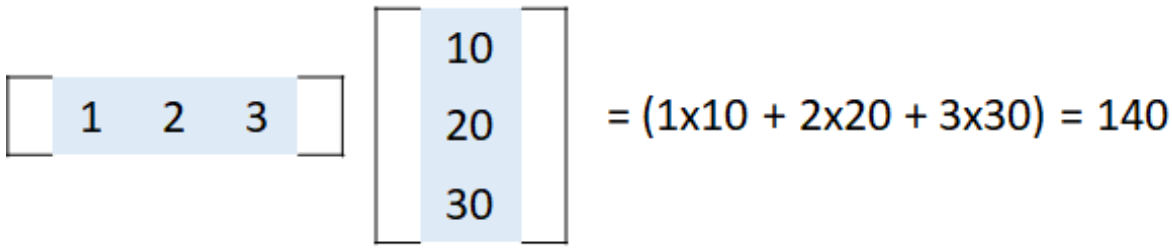
Vector, Matrix and Tensor

- Vectors are 1-dimensional arrays of numbers.
- Matrices are 2-dimensional arrays of numbers.
- Tensors are multi-dimensional arrays of numbers.
- They are useful in expressing numerical information, and are extremely useful in expressing different operations.
- So, they are important in statistics, machine learning and computer science.



Dot Product

- Dot product is the sum of products of values in two same-sized vectors (arrays).
- The output of the dot product is a scalar.
- Let's see how to perform dot product.



Matrix Multiplications

- Matrix multiplication is one of the very important operation for artificial intelligence.
- It is a matrix version of the dot product with two matrices.
- The output of the matrix multiplication is a matrix whose elements are the dot products of pairs of vectors in each matrix.
- Let's see how to perform matrix multiplication.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}_{2 \times 3} \times \begin{bmatrix} 10 & 11 \\ 20 & 21 \\ 30 & 31 \end{bmatrix}_{3 \times 2}$$

$$= \begin{bmatrix} 1X10 + 2X20 + 3X30 & 1X11 + 2X21 + 3X31 \\ 4X10 + 5X20 + 6X30 & 4X11 + 5X21 + 6X31 \end{bmatrix}$$

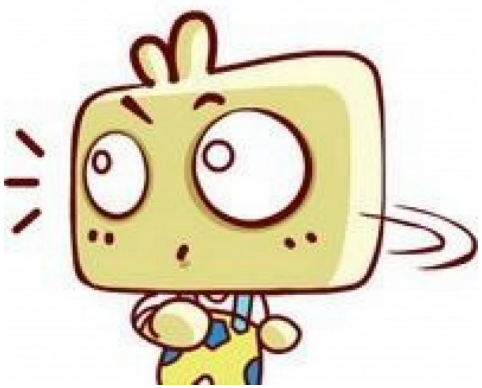
$$= \begin{bmatrix} 10 + 40 + 90 & 11 + 42 + 93 \\ 40 + 100 + 180 & 44 + 105 + 186 \end{bmatrix} = \begin{bmatrix} 140 & 146 \\ 320 & 335 \end{bmatrix}$$

Array Math

- Dot product can be performed using dot function/method.
- Matrix multiplication can be performed in one of the following ways in Python.
  - dot function/method
  - matmul method
  - @ operator for Python 3.5 or above (equivalent to matmul method)

Difference between dot and matmul on matrix multiplication

When we deal with multi-dimensional arrays (N-dimensional arrays with N>2), the result produced by the two is slightly different.
Recommendation: Only use matmul or @ operator to perform matrix multiplications.



Array Math

- dot is available both as a function in the NumPy module and as an instance method of array objects.

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])
v = np.array([9,10])
w = np.array([11, 12])

# Dot product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

Array Math

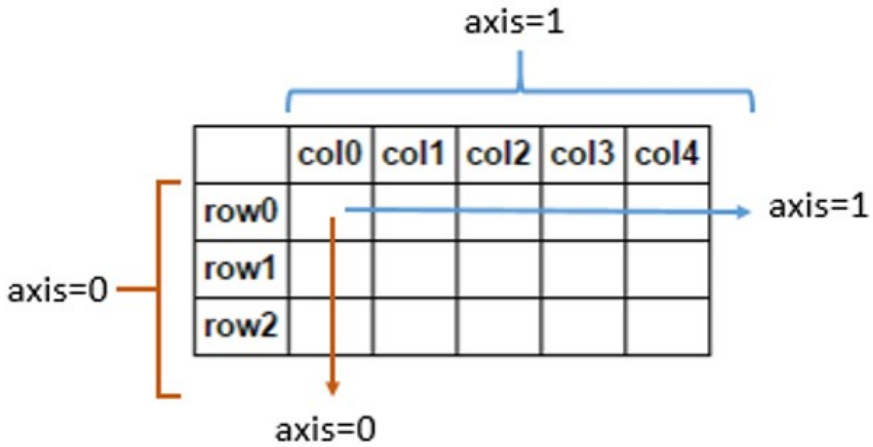
- matmul is only available as a function in the NumPy module.

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])
v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(np.matmul(v, w))
print(v@w)

# Matrix / vector product; both produce the rank 1 array [29 67]
print(np.matmul(x, v))
print(x@v)

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(np.matmul(x, y))
print(x@y)
```

NumPy Functions

- 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]"
```



NumPy Functions

- Other NumPy funtions follow the similar rules to operate on axes.

```
import numpy as np

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

print(np.mean(x))           # Compute mean of all elements; prints "2.5"
print(np.mean(x, axis=0)) # Compute mean of each column; prints "[2 3]"
print(np.mean(x, axis=1)) # Compute mean of each row; prints "[1.5 2.5]"

print(np.max(x))            # Compute max of all elements; prints "4"
print(np.max(x, axis=0))  # Compute max of each column; prints "[3 4]"
print(np.max(x, axis=1))  # Compute max of each row; prints "[2 4]"

print(np.min(x))            # Compute min of all elements; prints "1"
print(np.min(x, axis=0))  # Compute min of each column; prints "[1 2]"
print(np.min(x, axis=1))  # Compute min of each row; prints "[1 3]"
```

Transpose

- Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate in arrays.
- The simplest example of this type of operation is transposing a matrix (or two-dimensional array), to transpose a matrix, simply use the T attribute of an array object, or transpose function, or transpose method of array.
- T attribute of an array object or NumPy's transpose() function is used to reverse the dimensions of the given array, i.e. it changes the row elements to column elements and column to row elements.

```
import numpy as np

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

Transpose

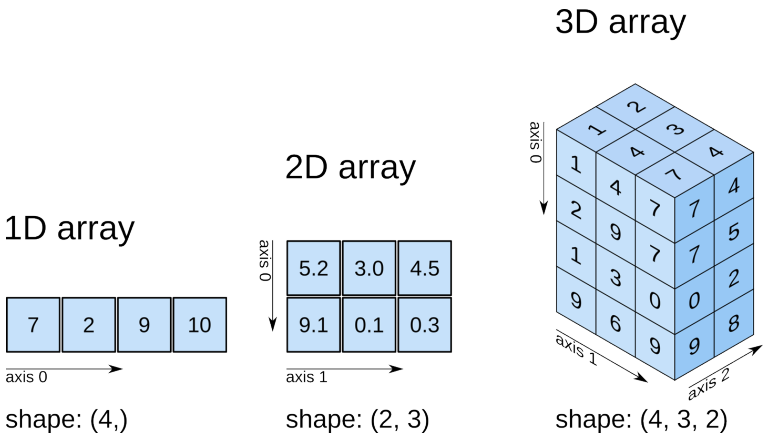
Can we perform transpose on a multi-dimensional array? Yes

- The transpose function comes with axes parameter which, according to the values specified to the axes parameter, permutes the array.

Syntax

`np.transpose(<arr>,<axis>)`

- Parameters:
  - arr: the array you want to transpose
  - axis: By default, the value is None. When None or no value is passed it will reverse the dimensions of array arr. If specified, it must be the tuple or list, which contains the permutation of [0,1,.., N-1] where N is the number of axes of arr.



Transpose

```python
import numpy as np
# Create a 3D array
# (2 layers, 3 rows, and 4 columns)
# with numbers 0 to 23.
arr = np.array([[[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8,  9, 10, 11]],
                [[12, 13, 14, 15],
                 [16, 17, 18, 19],
                 [20, 21, 22, 23]]])

# transpose can re-order the array axes.
# For example, if we want to
# take the current last axis (i.e. 2),
# make it the axis 0 (put 2 at the front),
# take the current first axis (i.e. 0),
# make it the axis 1 (put 0 in the middle),
# take the current second axis (i.e. 1),
# and make it the axis 2 (put 1 at the end).
# We pass transpose [2, 0, 1].
arr_T = np.transpose(arr, [2,0,1])
print(arr_T)
# Same output if we put the line below
print(arr.transpose([2, 0, 1]))

# Print "[[[ 0  4  8]
#           [12 16 20]
#
#          [[ 1  5  9]
#           [13 17 21]]
#
#          [[ 2  6 10]
#           [14 18 22]]
#
#          [[ 3  7 11]
#           [15 19 23]]]"

# Element-wise mapping:
# arr[i, j, k] = arr_T[k, i, j]
```

Transpose

- Note that taking the transpose of a rank 1 array (i.e. 1D array) does nothing.

Note

```
import numpy as np

v = np.array([1,2,3])
print(v)                # Print "[1 2 3]"
print(v.T)              # Print "[1 2 3]"
print(np.transpose(v)) # Print "[1 2 3]"
```

Reshaping

- Reshaping means changing the shape of an array.
- Recall, the shape of an array is the number of elements in each dimension.
- By reshaping, we can add or remove dimensions or change number of elements in each dimension.

Syntax

```
numpy.reshape(<arr>, <newshape>)
```

- Parameters:
  - arr: Array to be reshaped.
  - newshape: int or tuple of ints
    The new shape should be compatible with the original shape. If an integer, then the result will be a 1D array of that length. One shape dimension can be -1. In this case, the value is inferred from the length of the array and the remaining dimensions.
    Specifically, if arr has the shape $(a_0, a_1, \ldots, a_{n-1})$, and newshape $= (b_0, b_1, \ldots, b_{m-1})$, then $a_0 \cdot a_1 \cdot \cdots \cdot a_{n-1} = b_0 \cdot b_1 \cdot \cdots \cdot b_{m-1}$.

Reshaping

```python
import numpy as np

arr1 = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
arr2 = arr1.reshape(4, 3)    # Convert arr1 to a 2D array with 4 rows, 3 columns
# arr2 = arr1.reshape(4,-1) # This line has the same effect as arr1.reshape(4,3)
print(arr2) # Print "[[ 1  2  3]
            #         [ 4  5  6]
            #         [ 7  8  9]
            #         [10 11 12]]"
print("Shape of arr2:", arr2.shape) # Print "Shape of arr2: (4,3)"
```

Question

How to create the following array in one line of code?

```python
arr = np.array([[[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8,  9, 10, 11]],
                [[12, 13, 14, 15],
                 [16, 17, 18, 19],
                 [20, 21, 22, 23]]])
# Answer: arr = np.arange(24).reshape(2,3,4)
```

Increasing the Dimensions of an Array

- `np.newaxis` and `np.expand\_dims()` allow us to increase the dimensions of an array by adding new axes.
- Note: `numpy.newaxis` is an alias for `None`.

Syntax

`numpy.newaxis`

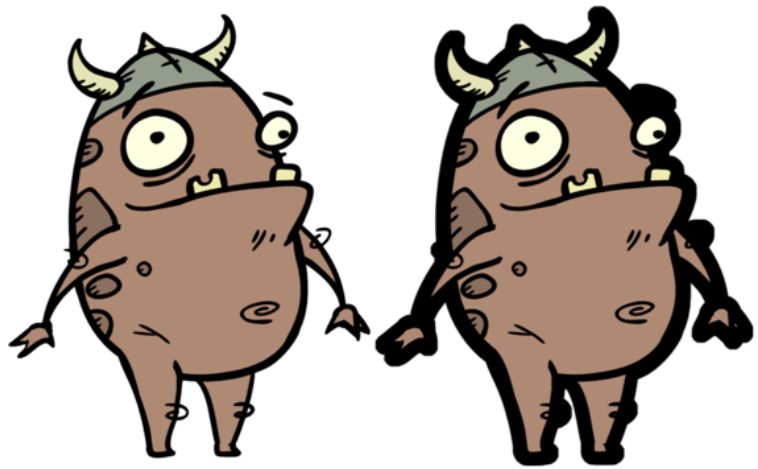
- Usage:
  - Placing `numpy.newaxis` inside [] adds a new dimension of size 1 at that position.

`numpy.expand_dims(arr, axis)`

- Parameters:
  - arr: Input array.
  - axis: int or tuple of ints, which refers to the position in the expanded axes where the new axis (or axes) is placed.

Increasing the Dimensions of an Array using np.newaxis

```
import numpy as np

arr1 = np.array([1, 2, 3, 4, 5])
print(arr1)  # It prints [1 2 3 4 5]

# Convert 1D array to a column matrix
arr2 = arr1[np.newaxis]
print(arr2)  # It prints [[1 2 3 4 5]]

# Convert 1D array to a column matrix
arr3 = arr1[None]
print(arr3)  # It prints [[1 2 3 4 5]]

arr4 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr4)  # It prints [[1 2 3]
             #             [4 5 6]]

arr5 = arr4[np.newaxis]
print(arr5)  # It prints [[[1 2 3]
             #              [4 5 6]]]
```



Increasing the Dimensions of an Array using np.newaxis

```
import numpy as np

arr1 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr1)  # It prints [[1 2 3]
             #            [4 5 6]]

# np.newaxis means adding a new dimension of size 1, i.e., 1 column
arr2 = arr1[:, :, np.newaxis]  # Equivalent to arr2 = arr1[..., np.newaxis]

print(arr2)  # It prints [[[1]
             #             [2]
             #             [3]]
             #            [[4]
             #             [5]
             #             [6]]]

print(arr2.shape) # It prints (2, 3, 1), which means 2 layers, 3 rows, 1 column
```

Increasing the Dimensions of an Array using np.newaxis

```
import numpy as np

arr1 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr1)  # It prints [[1 2 3]
             #           [4 5 6]]

# np.newaxis means adding a new dimension of size 1, i.e., 1 row
# highest dimension, i.e. column
arr2 = arr1[:, np.newaxis, :]

print(arr2)  # It prints [[[1 2 3]]
             #           [[4 5 6]]]

print(arr2.shape) # It prints (2, 1, 3), which means 2 layers, 1 row, 3 columns
```

Increasing the Dimensions of an Array using np.expand_dims

```python
import numpy as np

arr1 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr1)  # It prints [[1 2 3]
             #            [4 5 6]]

# 2 means adding a new dimension of size 1, after the 2nd dimension
arr2 = np.expand_dims(arr1, 2)

print(arr2)  # It prints [[[1]
             #             [2]
             #             [3]]
             #            [[4]
             #             [5]
             #             [6]]]

print(arr2.shape) # It prints (2, 3, 1), which means 2 layers, 3 rows, 1 column
```

Increasing the Dimensions of an Array using np.expand_dims

```python
import numpy as np

arr1 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr1)  # It prints [[1 2 3]
             #            [4 5 6]]

# 1 means adding a new dimension of size 1, after the 1st dimension
arr2 = np.expand_dims(arr1, 1)

print(arr2)  # It prints [[[1 2 3]]
             #            [[4 5 6]]]

print(arr2.shape) # It prints (2, 1, 3), which means 2 layers, 1 row, 3 columns
```

View and Copy

- In Python, the terms view and copy refer to how data is accessed and modified.
- View
  - A view is a reference to the original data.
  - Changes made through a view affect the original data.

```
import numpy as np    # Filename: view_example.py
arr = np.array([1, 2, 3])
view_arr = arr[1:3]   # Creates a view
view_arr[0] = 10      # Modifies arr
print(arr)            # Print [ 1 10 3]
```

- Copy
  - A copy creates a new object with the same data.
  - Modifications to a copy do not affect the original object.
  - Useful for preserving the original data while making changes.

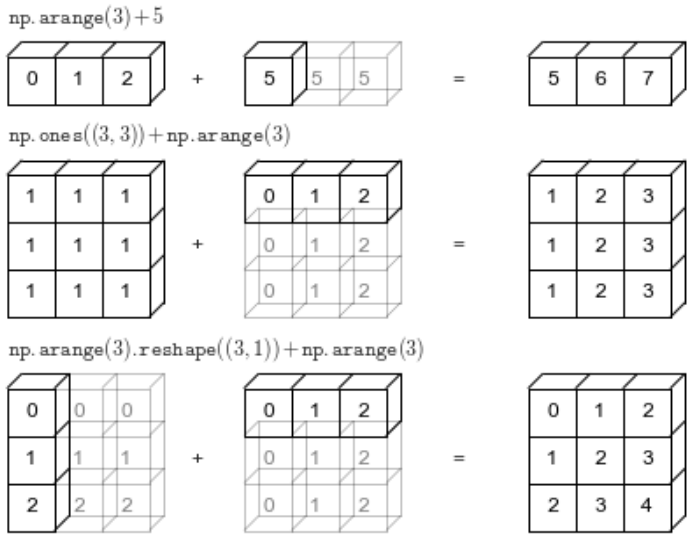
```
import numpy as np  # Filename: copy_example.py
import copy
original = np.array([1, 2, 3])
copy_list = copy.copy(original)  # Creates a shallow copy
copy_list[0] = 10                # Does not affect original
print(original)                  # Print [1 2 3]
```

NumPy Operations: View vs. Copy

- Operations that produce a view:
  - `arr[1:3]` - Slicing an array.
  - `arr.reshape(new_shape)` - Reshaping an array without changing data.
  - `arr.transpose()` - Transposing an array.
  - `arr.flatten()` - Returns a flattened view of the array.
  - `arr[:, :]` - Subsetting the array.
  - `arr.view()` - Creates a view of the array.
  - `arr[np.newaxis]` - Adds a new axis, creating a view.
  - `np.expand_dims(arr, axis)` - Expands the shape of the array, creating a view.
- Operations that produce a copy:
  - `np.copy(arr)` - Creates a deep copy of the array.
  - `arr.copy()` - Method to create a copy of the array.
  - `arr.astype(new_dtype)` - Converts to a new data type, creating a copy.
  - `np.array(arr)` - Creating a new array from an existing one.
  - `arr.tolist()` - Converts the array to a list, creating a copy.
  - `arr[np.arange(n)]` - Indexing with an array produces a copy.

Broadcasting

- Broadcasting is a powerful mechanism that allows NumPy to work with arrays of different shapes when performing arithmetic operations.
- Frequently, we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.



Broadcasting solves the problem of arithmetic between arrays of differing shapes by replicating the smaller array along the last mismatched dimension.

Broadcasting - Motivation

```python
import numpy as np

# We will add the vector v to each row of the matrix x, storing the result in y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

print(y)  # It prints "[[2 2 4]
          #             [5 5 7]
          #             [8 8 10]
          #             [11 11 13]]"
```

The above works, however when the matrix x is very large, computing an explicit loop could be slow.

Broadcasting - Motivation

- Note that adding the vector v to each row of the matrix x is equivalent to forming a matrix vv by stacking multiple copies of v vertically, then performing element-wise summation of x and vv. We could implement this as follows:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]]"
y = x + vv                # Add x and vv elementwise

print(y)                  # Prints "[[2 2 4]
                          #          [5 5 7]
                          #          [8 8 10]
                          #          [11 11 13]]"
```

Broadcasting

- NumPy broadcasting allows us to perform this computation without actually creating multiple copies of v.
- Consider the following version, using broadcasting:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)   # Prints "[[ 2  2  4]
           #         [ 5  5  7]
           #         [ 8  8 10]
           #         [11 11 13]]"
```

The line `y = x + v` works even though x has shape (4, 3) and v has shape (3,) due to broadcasting; this line works as if v actually had shape (4, 3), where each row was a copy of v, and the sum was performed element-wise.

Broadcasting Rules

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 element-wise 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.

Understanding Broadcasting Rules

- Given two arrays `A = np.array([1, 2, 3])`, and `B = np.array([2])`. Can we perform `A * B`?
  1. Do they have the same rank? Yes, rank of A is 1, rank of B is 1.
  2. Are they compatible in all dimensions? Yes. Array B has size 1 in that dimension.
  3. So, they are compatible.

```
import numpy as np

A = np.array([1, 2, 3])
print(A.ndim)      # Print 1
print(A.shape)     # Print (3,)
B = np.array([2])
print(B.ndim)      # Print 1
print(B.shape)     # Print (1,)
print(A * B)       # Print "[2, 4, 6]"
# B behaves as if B=[2, 2, 2]
```

Array	Shape		
A	(	3,	)
B	(	1,	)
		B has size 1	
A * B	(	3,	)

Understanding Broadcasting Rules

- Given two arrays `A = np.array([1, 2, 3])`, and `B = np.array([[4, 4, 4], [3, 3, 3]])`. Can we perform `A * B`?
  1. Do they have the same rank? No, they do not have the same rank, rank of A is 1, rank of B is 2. But we can prepend the shape of A (the lower rank array) with 1s until they have the same length.
  2. Are they compatible in all dimensions? Yes. Array A has size 1 in that dimension.
  3. So, they are compatible.

```
import numpy as np

A = np.array([1, 2, 3])
print(A.ndim)  # Print 1
print(A.shape) # Print (3,)
B = np.array([[4, 4, 4],
              [3, 3, 3]])
print(B.ndim)  # Print 2
print(B.shape) # Print (2,3)
print(A*B)     # Print [[4 8 12]
               #        [3 6 9]]
```

```
# A behaves as if A = [[1, 2, 3],
#                      [1, 2, 3]]
```

Array	Shape			
A	(	1 (Prepended),	3	)
B	(	2,	3	)
		A has size 1	Same size	
A * B	(	2,	3	)

Understanding Broadcasting Rules

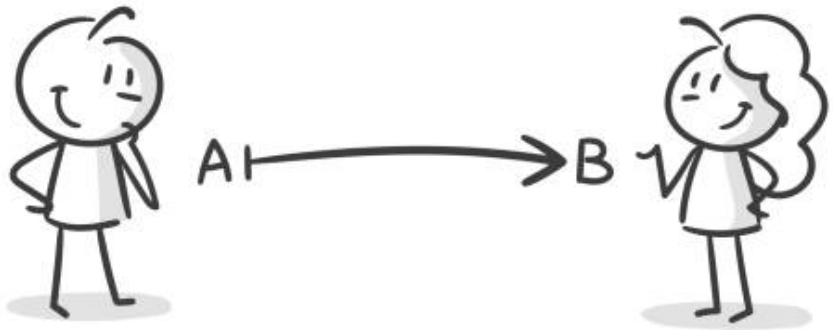
For each of the following pairs, state whether they are compatible. If they are compatible, what is the size of the resulting array after performing A*B? How about the following?

1. Pair 1
   - A: Shape - 5×4
   - B: Shape - 1
2. Pair 2
   - A: Shape - 5×4
   - B: Shape - 4
3. Pair 3
   - A: Shape - $15 \times 3 \times 5$
   - B: Shape - $15 \times 1 \times 5$
4. Pair 4
   - A: Shape - $15 \times 3 \times 5$
   - B: Shape - 3×5
5. Pair 5
   - A: Shape - $15 \times 3 \times 5$
   - B: Shape - 3×1
6. Pair 6
   - A: Shape - $16 \times 6 \times 7$
   - B: Shape - 16×6

1. Yes. Result - 5×4
2. Yes. Result - 5×4
3. Yes. Result - $15 \times 3 \times 5$
4. Yes. Result - $15 \times 3 \times 5$
5. Yes. Result - $15 \times 3 \times 5$
6. No.

Broadcasting in Practice

- Broadcasting operations are useful in simplifying the calculations.
- We will examine the following examples to illustrate their usefulness.
  1. Centering an array (Grade book example)
  2. Pairwise Distances (Euclidean distance between each pair of rows in two arrays)



Example 1: Centering an Array

- Suppose we have a grade book for 4 students, each of whom have taken 3 exams. Naturally, we store these scores in a 4×3 array.

```
import numpy as np
scores = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
```

- Now, we compute the mean of each exam using `mean` function across the first dimension.

```
score_mean = scores.mean(0)  # 0 here means the first dimension
print(score_mean)            # Prints array([5.5 6.5 7.5])
```

- Now, we can center the `scores` array by subtracting the mean (this is a broadcasting operation):

```
scores_centered = scores - score_mean
print(scores_centered)    # Prints array([[-4.5 -4.5 -4.5]
                          #               [-1.5 -1.5 -1.5]
                          #               [ 1.5  1.5  1.5]
                          #               [ 4.5  4.5  4.5]])
```

- To double-check what we have done is correct, we check the centered array and see whether it has zero mean.

```
print(scores_centered.mean(0))    # Prints array([0., 0., 0.])
```

Example 2: Pairwise Distances

- Suppose we have two, 2D arrays, x and y.
- x has a shape of (M, D) and y has a shape of (N, D), i.e., they have different number of rows, but same number of columns.
- We want to compute the Euclidean distance between each pair of rows between these two arrays.
- That is, if a given row of x is represented by D numbers $(x_0, x_1, \ldots, x_{D-1})$, and similarly, a row of y is represented by D numbers $(y_0, y_1, \ldots, y_{D-1})$, and we want to compute the Euclidean distance between the two rows:

$$\sqrt{(x_0 - y_0)^2 + (x_1 - y_1)^2 + \ldots + (x_{D-1} - y_{D-1})^2}$$

Example 2: Pairwise Distances

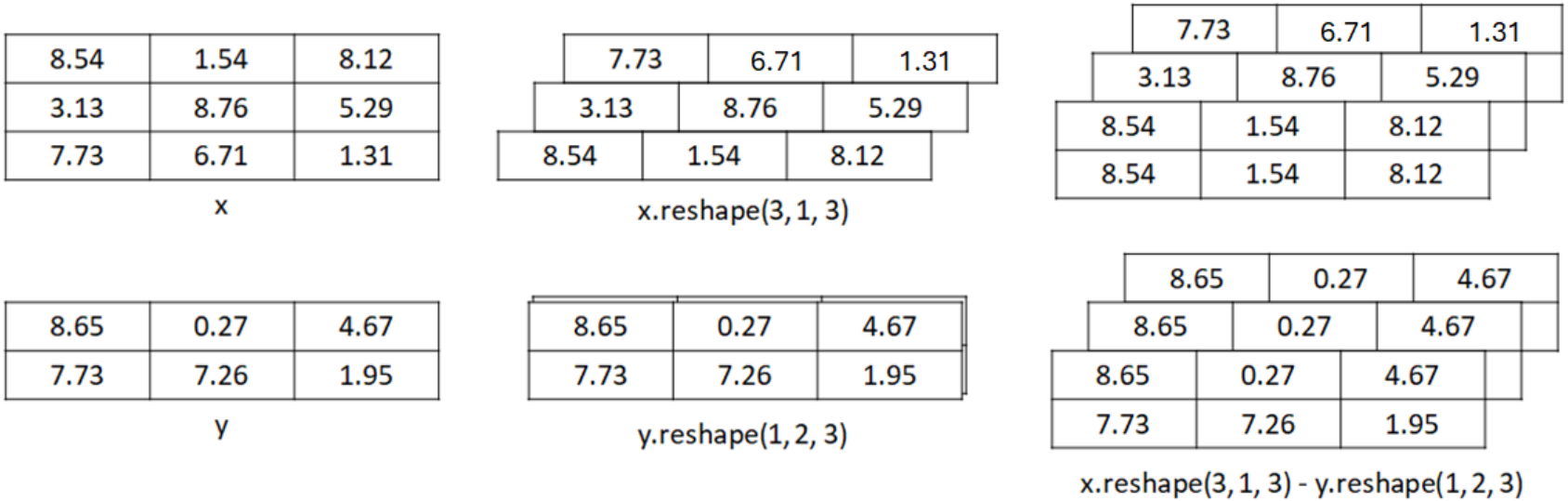
```
# a shape-(3, 3) array
x = np.array([[ 8.54,  1.54,  8.12],
              [ 3.13,  8.76,  5.29],
              [ 7.73,  6.71,  1.31]])

# a shape-(2, 3) array
y = np.array([[ 8.65,  0.27,  4.67],
              [ 7.73,  7.26,  1.95]])
```

Questions

1. How many distance values do we need to compute?
   Answer: 6 distances, one for each pair of rows from x and y.
2. If x has a shape of (M, 3) and y has a shape of (N, 3), how many distance values do we need to compute?
   Answer: M $\times$ N
3. We store the distance values in an array `Output`. What is each element in `Output`?
   Answer: `Output[i, j]` represents the distance between `x[i]` and `y[j]`.
4. Can `x` and `y` be broadcasting?
   Answer: No!

Example 2: Pairwise Distances

```
reshaped_x = x.reshape(3, 1, 3)
reshaped_y = y.reshape(1, 2, 3)
diffs = reshaped_x - reshaped_y
print("Shape of diffs", diffs.shape)    # Prints (3, 2, 3)
```



In the array `diffs`:

- Layer 0: `[[x[0] - y[0]], [x[0] - y[1]]]`
- Layer 1: `[[x[1] - y[0]], [x[1] - y[1]]]`
- Layer 2: `[[x[2] - y[0]], [x[2] - y[1]]]`

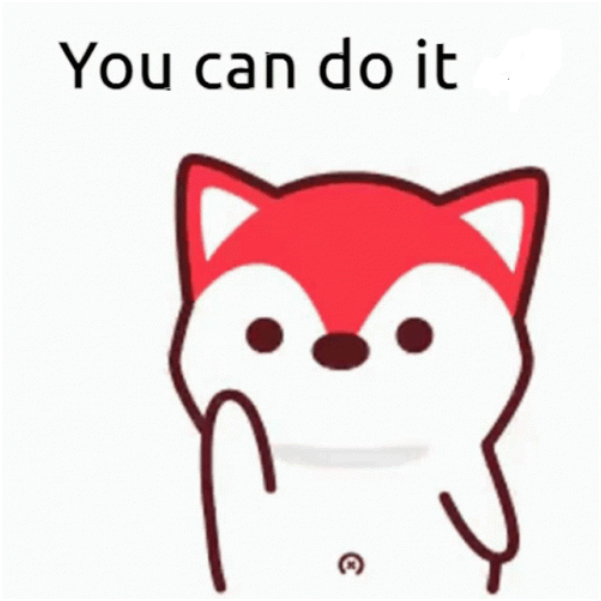
It is important to see, via broadcasting, that `diffs[i,j]` stores `x[i] - y[j]`!

Example 2: Pairwise Distances

```
print(diffs)  # It prints [[[-0.11  1.27  3.45]
              #             [ 0.81 -5.72  6.17]]
              #            [[-5.52  8.49  0.62]
              #             [-4.6   1.5   3.34]]
              #            [[-0.92  6.44 -3.36]
              #             [ 0.   -0.55 -0.64]]]

dists = np.sqrt(np.sum(diffs**2, axis=2))  # axis=2 means the column axis

print(dists)  # It prints [[ 3.67797499  8.45241977]
              #           [10.14568381  5.87925165]
              #           [ 7.32185769  0.84386018]]

print(dists.shape)  # It prints (3, 2), which means 3 rows, 2 columns
```

Question on Pairwise Distances

- Do you know how to write a program that computes the pairwise distances of two arrays, without using the reshape function but instead using np.newaxis?
- Similarly, do you know how to write a program that computes the pairwise distances of two arrays, without using the reshape function but instead using np.expand_dims?

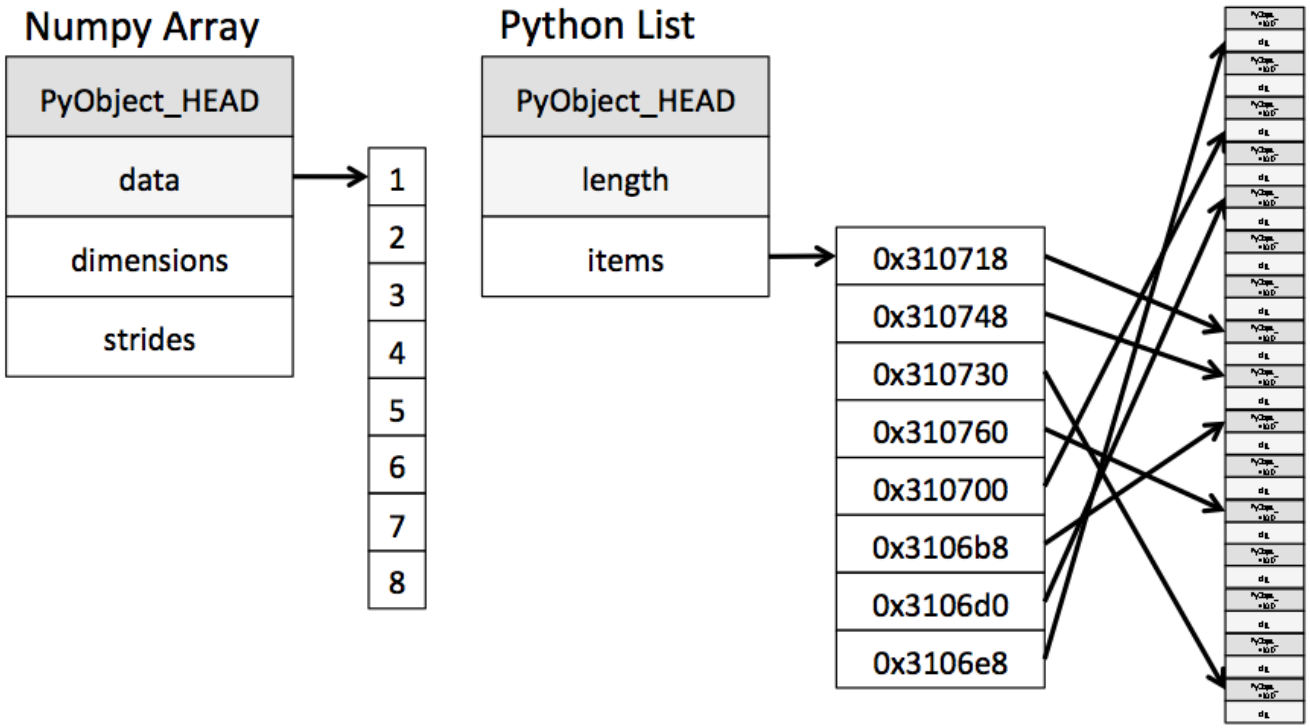


Why NumPy Arrays are Better?

- NumPy arrays consume less memory than Python List
- NumPy arrays are fast as compared to Python List
- NumPy arrays are more convenient to use

NumPy Array vs Python List

- NumPy array contains a single pointer to one contiguous block of data.
- The Python list contains a pointer to a block of pointers, each of which points to a full Python object like the Python object.



Memory Consumption for NumPy Arrays and Lists

```
import numpy as np; import sys; import gc  # Import numpy package, system and gc module

def actualsize(input_object):
    memory_size = 0                    # memory_size: the actual memory size of "input_object"
                                       # Initialize it to 0
    ids = set()                        # ids: An empty set to store all the ids of objects
    objects = [input_object]           # objects: A list with "input_object"
    while objects:                     # While "objects" is non-empty
        new = []                       # new: An empty list to keep the items linked by "objects"
        for obj in objects:            # obj: Each object in "objects"
            if id(obj) not in ids: # If the id of "obj" is not in "ids"
                ids.add(id(obj))    # Add the id of the "obj" to "ids"
                memory_size += sys.getsizeof(obj) # Use getsizeof to determine the size of "obj"
                                       # and add it to "memory_size"
                new.append(obj)             # Add "obj" to "new"
        objects = gc.get_referents(*new)  # Update "objects" with the list of objects directly
                                          # referred to by *new
    return memory_size                 # Return "memory_size"

L = list(range(0, 1000))               # Define a Python list of 1000 elements
A = np.arange(1000)                    # Define a numpy array of 1000 elements
# Print size of the whole list
print("Size of the whole list in bytes:", actualsize(L))        # Print 37116
# Print size of the whole NumPy array
print("Size of the whole numpy array in bytes:", actualsize(A)) # Print 8104
```

Time Comparison Between NumPy Arrays and Python Lists

```
import numpy as np        # Import required packages
import time as t

size = 1000000            # Size of arrays and lists

list1 = range(size)       # Declare lists
list2 = range(size)
array1 = np.arange(size) # Declare arrays
array2 = np.arange(size)

# Capture time before the multiplication of Python lists
initialTime = t.time()
# Multiply elements of both the lists and stored in another list
resultList = [(a * b) for a, b in zip(list1, list2)]
# Calculate execution time, it prints "Time taken by Lists: 0.1302425861358642 6 s"
print("Time taken by Lists:", (t.time() - initialTime), "s")

# Capture time before the multiplication of NumPy arrays
initialTime = t.time()
# Multiply elements of both the NumPy arrays and stored in another NumPy array
resultArray = array1 * array2
# Calculate execution time, it prints "Time taken by NumPy Arrays: 0.006006956100463867 s"
print("Time taken by NumPy Arrays:", (t.time() - initialTime), "s")
```

Effect of Operations on NumPy Arrays and Python Lists

```
import numpy as np # Import NumPy package

ls =[1, 2, 3]       # Declare a list
arr = np.array(ls) # Convert the list into a NumPy array

try:
    ls = ls + 4     # Add 4 to each element of list
except(TypeError):
    print("Lists don't support list + int")

# Now on array
try:
    arr = arr + 4  # Add 4 to each element of NumPy array
    print("Modified NumPy array: ",arr)  # Print the NumPy array
except(TypeError):
    print("NumPy arrays don't support list + int")
```

Output:

```
Lists don't support list + int
Modified NumPy array: [5 6 7]
```

Practice Problems (Print the result)

1. How to create array A of size 15, with all zeros?
2. How to find memory size of array A?
3. How to create array B with values ranging from 20 to 60?
4. How to create array C of reversed array of B?
5. How to create 4×4 array D with values from 0 to 15 (from top to bottom, left to right)?
6. How to find the dimensions of array E $[[3, 4, 5], [6, 7, 8]]$?
7. How to find indices for non-zero elements from array F $[0, 3, 0, 0, 4, 0]$?
8. How to create 3×3×3 array G with random values?
9. How to find maximum values in array H $[1, 13, 0, 56, 71, 22]$?
10. How to find minimum values in array H?
11. How to find mean values of array H?
12. How to find standard deviation of array H?
13. How to find median in array H?
14. How to transpose array D?

Practice Problems (Print the results)

15. How to append array [4, 5, 6] to array I [1, 2, 3]?
16. How to member-wise add, subtract, multiply and divide two arrays J [1, 2, 3] and K [4, 5, 6]?
17. How to find the total sum of elements of array I?
18. How to find natural log of array I?
19. How to use build an array L with [8, 8, 8, 8, 8] using full/repeat function?
20. How to sort array M [2, 5, 7, 3, 6]?
21. How to find the indices of the maximum values in array M?
22. How to find the indices of the minimum values in array M?
23. How to find the indices of elements in array M that will be sorted?
24. How to find the inverse of array N = [[6, 1, 1], [4, -2, 5], [2, 8, 7]] in NumPy?
25. How to find absolute value of array N?
26. How to extract the third column (from all rows) of the array O [[11, 22, 33], [44, 55, 66], [77, 88, 99]]?
27. How to extract the sub-array consisting of the odd rows and even columns of P [[3, 6, 9, 12], [15, 18, 21, 24], [27, 30, 33, 36], [39, 42, 45, 48], [51, 54, 57, 60]]?

Practice Problems (Answers)

1. `A = np.zeros(15); print(A)`
2. `print(A.size * A.itemsize)`
3. `B = np.arange(20, 61); print(B)`
4. `C = B[::-1]; print(C)`
5. `D = np.arange(16).reshape(4, 4); print(D)`
6. `E = np.array([[3, 4, 5], [6, 7, 8]]); print(E.shape)`
7. `F = np.array([0, 3, 0, 0, 4, 0]); print(F.nonzero())`
8. `G = np.random.random((3, 3, 3)); print(G)`
9. `H = np.array([1, 13, 0, 56, 71, 22]); print(H.max())`
10. `print(H.min())`
11. `print(H.mean())`
12. `print(H.std())`
13. `print(np.median(H))`
14. `print(np.transpose(D))`
15. `I = np.array([1, 2, 3]); I = np.append(I, [4, 5, 6]); print(I)`

Practice Problems (Answers)

16.
```
J = np.array([1, 2, 3]); K = np.array([4, 5, 6]);
print(J + K); print(J - K); print(J * K); print(J / K)
```
17.
```
print(np.sum(I))
```
18.
```
print(np.log(I))
```
19.
```
L = no.full(5, 8); print(L)
# L = np.repeat(8, 5); print(L)
```
20.
```
M = np.array([2, 5, 7, 3, 6]); print(np.sort(M))
```
21.
```
print(M.argmax())
```
22.
```
print(M.argmin())
```
23.
```
print(M.argsort())
```
24.
```
N = np.array([[6, 1, 1], [4, -2, 5], [2, 8, 7]]); print(np.linalg.inv(N))
```
25.
```
print(np.abs(N))
```
26.
```
O = np.array([[11, 22, 33], [44, 55, 66], [77, 88, 99]])
print(O[:,2])
```
27.
```
P = np.array([[3, 6, 9, 12], [15, 18, 21, 24], [27, 30, 33, 36],
              [39,  42, 45, 48], [51, 54, 57, 60]])
print(P[::2, 1::2])
```

That's all!

Any question?

Welcome Back!

Acknowledgments

- The lecture notes are developed based on Dr. Desmond Tsoi's lecture slides.