Open in Colab

T81-558: Applications of Deep Neural Networks

Module 1: Python Preliminaries

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 1 Material

- **Part 1.1: Course Overview** [Video] [Notebook]
- Part 1.2: Introduction to Python [Video] [Notebook]
- Part 1.3: Python Lists, Dictionaries, Sets and JSON [Video] [Notebook]
- Part 1.4: File Handling [Video] [Notebook]
- Part 1.5: Functions, Lambdas, and Map/Reduce [Video] [Notebook]

Watch one (or more) of these depending on how you want to setup your Python TensorFlow environment:

- How to Submit a Module Assignment locally
- How to Use Google CoLab and Submit Assignment
- Installing TensorFlow, Keras, and Python in Windows CPU or GPU
- Installing TensorFlow, Keras, and Python for an Intel Mac
- Installing TensorFlow, Keras, and Python for an M1 Mac

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:
```
try:
    from google.colab import drive
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

```
Note: not using Google CoLab
```

Part 1.1: Overview

Deep learning is a group of exciting new technologies for neural networks. [Cite:lecun2015deep] By using a combination of advanced training techniques neural network architectural components, it is now possible to train neural networks of much greater complexity. This book introduces the reader to deep neural networks, regularization units (ReLU), convolution neural networks, and recurrent neural networks. High-performance computing (HPC) aspects demonstrate how deep learning can be leveraged both on graphical processing units (GPUs), as well as grids. Deep learning allows a model to learn hierarchies of information in a way that is similar to the function of the human brain. The focus is primarily upon the application of deep learning, with some introduction to the mathematical foundations of deep learning. Readers will make use of the Python programming language to architect a deep learning model for several real-world data sets and interpret the results of these networks. [Cite:goodfellow2016deep]

Origins of Deep Learning

Neural networks are one of the earliest examples of a machine learning model. Neural networks were initially introduced in the 1940s and have risen and fallen several times in popularity. The current generation of deep learning began in 2006 with an improved training algorithm by Geoffrey Hinton. [Cite:hinton2006fast] This technique finally allowed neural networks with many layers (deep neural networks) to be efficiently trained. Four researchers have contributed significantly to the development of neural networks. They have consistently pushed neural network research, both through the ups and downs. These four luminaries are shown in Figure 1.LUM.

Figure 1.LUM: Neural Network Luminaries

The current luminaries of artificial neural network (ANN) research and ultimately deep learning, in order as appearing in the figure:

- Yann LeCun, Facebook and New York University - Optical character recognition and computer vision using convolutional neural networks (CNN). The founding father of convolutional nets.
- Geoffrey Hinton, Google and University of Toronto. Extensive work on neural networks. Creator of deep learning and early adapter/creator of backpropagation for neural networks.
- Yoshua Bengio, University of Montreal and Botler AI. Extensive research into deep learning, neural networks, and machine learning.
- Andrew Ng, Badiu and Stanford University. Extensive research into deep learning, neural networks, and application to robotics.

Geoffrey Hinton, Yann LeCun, and Yoshua Bengio won the Turing Award for their contributions to deep learning.

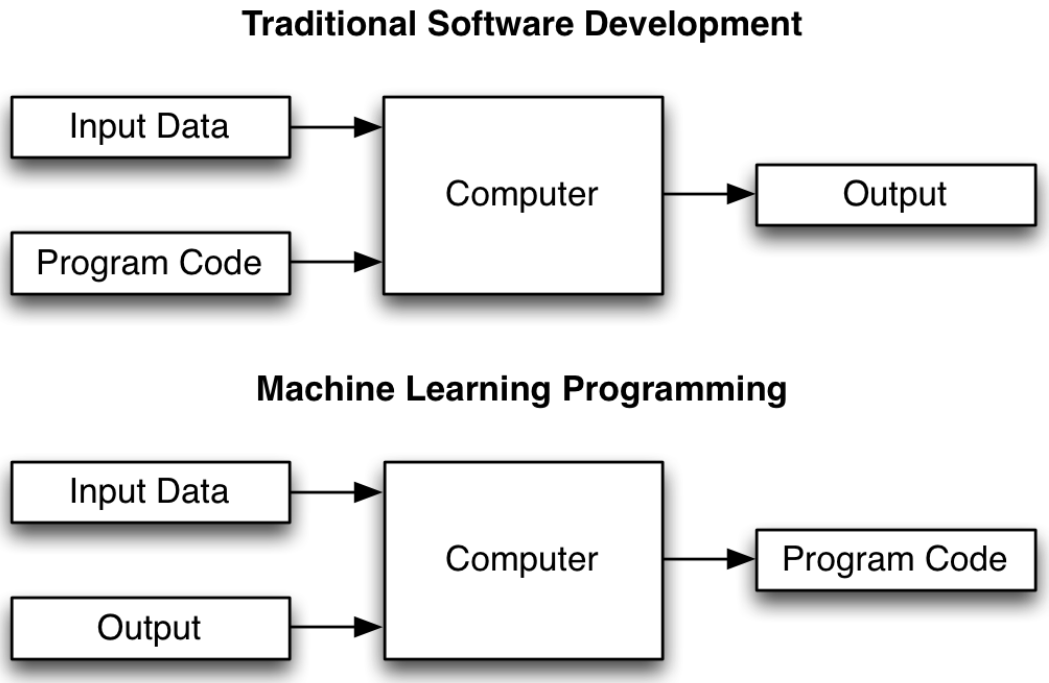
What is Deep Learning

The focus of this book is deep learning, which is a prevalent type of machine learning that builds upon the original neural networks popularized in the 1980s. There is very little difference between how a deep neural network is calculated compared with the first neural network. We've always been able to create and calculate deep neural networks. A deep neural network is nothing more than a neural network with many layers. While we've always been able to

create/calculate deep neural networks, we've lacked an effective means of training them. Deep learning provides an efficient means to train deep neural networks.

If deep learning is a type of machine learning, this begs the question, "What is machine learning?" Figure 1.ML-DEV illustrates how machine learning differs from traditional software development.

Figure 1.ML-DEV: ML vs Traditional Software Development



- **Traditional Software Development** - Programmers create programs that specify how to transform input into the desired output.
- **Machine Learning** - Programmers create models that can learn to produce the desired output for given input. This learning fills the traditional role of the computer program.

Researchers have applied machine learning to many different areas. This class explores three specific domains for the application of deep neural networks, as illustrated in Figure 1.ML-DOM.

Figure 1.ML-DOM: Application of Machine Learning

- **Computer Vision** - The use of machine learning to detect patterns in visual data. For example, is an image a picture of a cat or a dog.
- **Tabular Data** - Several named input values allow the neural network to predict another named value that becomes the output. For example, we are using four measurements of iris flowers to predict the species. This type of data is often called tabular data.
- **Natural Language Processing (NLP)** - Deep learning transformers have revolutionized NLP, allowing text sequences to generate more text, images, or classifications.
- **Reinforcement Learning** - Reinforcement learning trains a neural network to choose ongoing actions so that the algorithm rewards the neural network for optimally completing a task.
- **Time Series** - The use of machine learning to detect patterns in time. Typical time series applications are financial applications, speech recognition, and even natural language processing (NLP).
- **Generative Models** - Neural networks can learn to produce new original synthetic data from input. We will examine StyleGAN, which learns to create new images similar to those it saw during training.

Regression, Classification and Beyond

Machine learning research looks at problems in broad terms of supervised and unsupervised learning. Supervised learning occurs when you know the correct outcome for each item in the training set. On the other hand, unsupervised learning utilizes training sets where no correct outcome is known. Deep learning supports both supervised and unsupervised learning; however, it also adds reinforcement and adversarial learning. Reinforcement learning teaches the neural network to carry out actions based on an environment. Adversarial learning pits two neural networks against each other to learn when the data provides no correct outcomes. Researchers continue to add new deep learning training techniques.

Machine learning practitioners usually divide supervised learning into classification and regression. Classification networks might accept financial data and classify the investment risk as risk or safe. Similarly, a regression neural network outputs a number and might take the same data and return a risk score. Additionally, neural networks can output multiple regression and classification scores simultaneously.

One of the most powerful aspects of neural networks is that the input and output of a neural network can be of many different types, such as:

- An image
- A series of numbers that could represent text, audio, or another time series
- A regression number
- A classification class

Why Deep Learning?

For tabular data, neural networks often do not perform significantly better that different than other models, such as:

- Support Vector Machines
- Random Forests
- Gradient Boosted Machines

Like these other models, neural networks can perform both **classification** and **regression**. When applied to relatively low-dimensional tabular data tasks, deep neural networks do not necessarily add significant accuracy over other model types. However, most state-of-the-art solutions depend on deep neural networks for images, video, text, and audio data.

Python for Deep Learning

We will utilize the Python 3.x programming language for this book. Python has some of the widest support for deep learning as a programming language. The two most popular frameworks for deep learning in Python are:

- TensorFlow/Keras (Google)
- PyTorch (Facebook)

This book focuses primarily upon Keras, with some applications in PyTorch. For many tasks, we will utilize Keras directly. We will utilize third-party libraries for higher-level tasks, such as reinforcement learning, generative adversarial neural networks, and others. These third-party libraries may internally make use of

either PyTorch or Keras. I chose these libraries based on popularity and application, not whether they used PyTorch or Keras.

To successfully use this book, you must be able to compile and execute Python code that makes use of TensorFlow for deep learning. There are two options for you to accomplish this:

- Install Python, TensorFlow and some IDE (Jupyter, TensorFlow, and others).
- Use Google CoLab in the cloud, with free GPU access.

If you look at this notebook on Github, near the top of the document, there are links to videos that describe how to use Google CoLab. There are also videos explaining how to install Python on your local computer. The following sections take you through the process of installing Python on your local computer. This process is essentially the same on Windows, Linux, or Mac. For specific OS instructions, refer to one of the tutorial YouTube videos earlier in this document.

To install Python on your computer, complete the following instructions:

- Installing Python and TensorFlow - Windows/Linux
- Installing Python and TensorFlow - Mac Intel
- Installing Python and TensorFlow - Mac M1

Check your Python Installation

Once you've installed Python, you can utilize the following code to check your Python and library versions. If you have a GPU, you can also check to see that Keras recognize it.

In [3]:
```
# What version of Python do you have?
import sys

import tensorflow.keras
import pandas as pd
import sklearn as sk
import tensorflow as tf

check_gpu = len(tf.config.list_physical_devices('GPU'))>0

print(f"Tensor Flow Version: {tf.__version__}")
print(f"Keras Version: {tensorflow.keras.__version__}")
print()
print(f"Python {sys.version}")
print(f"Pandas {pd.__version__}")
print(f"Scikit-Learn {sk.__version__}")
print("GPU is", "available" if check_gpu \
      else "NOT AVAILABLE")
```

```
Tensor Flow Version: 2.1.0
Keras Version: 2.2.4-tf

Python 3.7.16 (default, Jan 17 2023, 22:20:44)
[GCC 11.2.0]
Pandas 1.3.5
Scikit-Learn 1.0.2
GPU is NOT AVAILABLE
```

Module 1 Assignment

You can find the first assignment here: assignment 1

Open in Colab

T81-558: Applications of Deep Neural Networks

Module 1: Python Preliminaries

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 1 Material

- Part 1.1: Course Overview [Video] [Notebook]
- **Part 1.2: Introduction to Python** [Video] [Notebook]
- Part 1.3: Python Lists, Dictionaries, Sets and JSON [Video] [Notebook]
- Part 1.4: File Handling [Video] [Notebook]
- Part 1.5: Functions, Lambdas, and Map/Reduce [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:
```
try:
    from google.colab import drive
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

```
Note: not using Google CoLab
```

Part 1.2: Introduction to Python

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help

programmers write clear, logical code for small and large-scale projects. Python has become a common language for machine learning research and is the primary language for TensorFlow.

Python 3.0, released in 2008, was a significant revision of the language that is not entirely backward-compatible, and much Python 2 code does not run unmodified on Python 3. This course makes use of Python 3. Furthermore, TensorFlow is not compatible with versions of Python earlier than 3. A non-profit organization, the Python Software Foundation (PSF), manages and directs resources for Python development. On January 1, 2020, the PSF discontinued the Python 2 language and no longer provides security patches and other improvements. Python interpreters are available for many operating systems.

The first two modules of this course provide an introduction to some aspects of the Python programming language. However, entire books focus on Python. Two modules will not cover every detail of this language. The reader is encouraged to consult additional sources on the Python language.

Like most tutorials, we will begin by printing Hello World.

In [2]:
```
print("Hello World")
```

```
Hello World
```

The above code passes a constant string, containing the text "hello world" to a function that is named print.

You can also leave comments in your code to explain what you are doing. Comments can begin anywhere in a line.

In [3]:
```
# Single line comment (this has no effect on your program)
print("Hello World") # Say hello
```

```
Hello World
```

Strings are very versatile and allow your program to process textual information. Constant string, enclosed in quotes, define literal string values inside your program. Sometimes you may wish to define a larger amount of literal text inside of your program. This text might consist of multiple lines. The triple quote allows for multiple lines of text.

In [4]:
```
print("""Print
Multiple
Lines
""")
```

```
Print
Multiple
Lines
```

Like many languages Python uses single (') and double (") quotes interchangeably to denote literal string constants. The general convention is that double quotes should enclose actual text, such as words or sentences. Single quotes should enclose symbolic text, such as error codes. An example of an error code might be 'HTTP404'.

However, there is no difference between single and double quotes in Python, and you may use whichever you like. The following code makes use of a single quote.

In [5]:
```
print('Hello World')
```

```
Hello World
```

In addition to strings, Python allows numbers as literal constants in programs. Python includes support for floating-point, integer, complex, and other types of numbers. This course will not make use of complex numbers. Unlike strings, quotes do not enclose numbers.

The presence of a decimal point differentiates floating-point and integer numbers. For example, the value 42 is an integer. Similarly, 42.5 is a floating-point number. If you wish to have a floating-point number, without a fraction part, you should specify a zero fraction. The value 42.0 is a floating-point number, although it has no fractional part. As an example, the following code prints two numbers.

In [6]:
```
print(42)
print(42.5)
```

```
42
42.5
```

So far, we have only seen how to define literal numeric and string values. These literal values are constant and do not change as your program runs. Variables allow your program to hold values that can change as the program runs. Variables have names that allow you to reference their values. The following code assigns an integer value to a variable named "a" and a string value to a variable named "b."

In [7]:
```
a = 10
b = "ten"
print(a)
print(b)
```

```
10
ten
```

The key feature of variables is that they can change. The following code demonstrates how to change the values held by variables.

In [8]:
```
a = 10
print(a)
a = a + 1
print(a)
```

```
10
11
```

You can mix strings and variables for printing. This technique is called a formatted or interpolated string. The variables must be inside of the curly braces. In Python, this type of string is generally called an f-string. The f-string is denoted by placing an "f" just in front of the opening single or double quote that begins the string. The following code demonstrates the use of an f-string to mix several variables with a literal string.

In [4]:
```
a = 9
print(f'The value of a is {a}')
```

```
The value of a is 9
```

You can also use f-strings with math (called an expression). Curly braces can enclose any valid Python expression for printing. The following code demonstrates the use of an expression inside of the curly braces of an f-string.

In [10]:
```
a = 10
print(f'The value of a plus 5 is {a+5}')
```

```
The value of a plus 5 is 15
```

Python has many ways to print numbers; these are all correct. However, for this course, we will use f-strings. The following code demonstrates some of the varied methods of printing numbers in Python.

In [5]:
```
a = 5

print(f'a is {a}') # Preferred method for this course.
print('a is {}'.format(a))
print('a is ' + str(a))
print('a is %d' % (a))
```

```
a is 5
a is 5
a is 5
a is 5
```

You can use if-statements to perform logic. Notice the indents? These if-statements are how Python defines blocks of code to execute together. A block usually begins after a colon and includes any lines at the same level of indent. Unlike many other programming languages, Python uses whitespace to define

blocks of code. The fact that whitespace is significant to the meaning of program code is a frequent source of annoyance for new programmers of Python. Tabs and spaces are both used to define the scope in a Python program. Mixing both spaces and tabs in the same program is not recommended.

In [12]:
```
a = 5
if a>5:
    print('The variable a is greater than 5.')
else:
    print('The variable a is not greater than 5')
```

```
The variable a is not greater than 5
```

The following if-statement has multiple levels. It can be easy to indent these levels improperly, so be careful. This code contains a nested if-statement under the first "a==5" if-statement. Only if a is equal to 5 will the nested "b==6" if-statement be executed. Also, note that the "elif" command means "else if."

In [13]:
```
a = 5
b = 6

if a==5:
    print('The variable a is 5')
    if b==6:
        print('The variable b is also 6')
elif a==6:
    print('The variable a is 6')
```

```
The variable a is 5
The variable b is also 6
```

It is also important to note that the double equal ("==") operator is used to test the equality of two expressions. The single equal ("=") operator is only used to assign values to variables in Python. The greater than (">"), less than ("<"), greater than or equal (">="), less than or equal ("<=") all perform as would generally be accepted. Testing for inequality is performed with the not equal ("!=") operator.

It is common in programming languages to loop over a range of numbers. Python accomplishes this through the use of the **range** operation. Here you can see a **for** loop and a **range** operation that causes the program to loop between 1 and 3.

In [14]:
```
for x in range(1, 3):  # If you ever see xrange, you are in Python 2
    print(x)
# If you ever see print x (no parenthesis), you are in Python 2
```

```
1
2
```

This code illustrates some incompatibilities between Python 2 and Python 3. Before Python 3, it was acceptable to leave the parentheses off of a *print* function call. This method of invoking the *print* command is no longer allowed in Python 3. Similarly, it used to be a performance improvement to use the *xrange* command in place of range command at times. Python 3 incorporated all of the functionality of the *xrange* Python 2 command into the normal *range* command. As a result, the programmer should not use the *xrange* command in Python 3. If you see either of these constructs used in example code, then you are looking at an older Python 2 era example.

The *range* command is used in conjunction with loops to pass over a specific range of numbers. Cases, where you must loop over specific number ranges, are somewhat uncommon. Generally, programmers use loops on collections of items, rather than hard-coding numeric values into your code. Collections, as well as the operations that loops can perform on them, is covered later in this module.

The following is a further example of a looped printing of strings and numbers.

In [15]:
```
acc = 0
for x in range(1, 3):
    acc += x
    print(f"Adding {x}, sum so far is {acc}")

print(f"Final sum: {acc}")
```

```
Adding 1, sum so far is 1
Adding 2, sum so far is 3
Final sum: 3
```

Open in Colab

T81-558: Applications of Deep Neural Networks

Module 1: Python Preliminaries

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 1 Material

- Part 1.1: Course Overview [Video] [Notebook]
- Part 1.2: Introduction to Python [Video] [Notebook]
- **Part 1.3: Python Lists, Dictionaries, Sets and JSON** [Video] [Notebook]
- Part 1.4: File Handling [Video] [Notebook]
- Part 1.5: Functions, Lambdas, and Map/Reduce [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:
```
try:
    from google.colab import drive
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")

except:
    print("Note: not using Google CoLab")
    COLAB = False
```

```
Note: not using Google CoLab
```

Part 1.3: Python Lists, Dictionaries, Sets, and JSON

Like most modern programming languages, Python includes Lists, Sets, Dictionaries, and other data structures as built-in types. The syntax appearance

of both of these is similar to JSON. Python and JSON compatibility is discussed later in this module. This course will focus primarily on Lists, Sets, and Dictionaries. It is essential to understand the differences between these three fundamental collection types.

- **Dictionary** - A dictionary is a mutable unordered collection that Python indexes with name and value pairs.
- **List** - A list is a mutable ordered collection that allows duplicate elements.
- **Set** - A set is a mutable unordered collection with no duplicate elements.
- **Tuple** - A tuple is an immutable ordered collection that allows duplicate elements.

Most Python collections are mutable, meaning the program can add and remove elements after definition. An immutable collection cannot add or remove items after definition. It is also essential to understand that an ordered collection means that items maintain their order as the program adds them to a collection. This order might not be any specific ordering, such as alphabetic or numeric.

Lists and tuples are very similar in Python and are often confused. The significant difference is that a list is mutable, but a tuple isn't. So, we include a list when we want to contain similar items and a tuple when we know what information goes into it ahead of time.

Many programming languages contain a data collection called an array. The array type is noticeably absent in Python. Generally, the programmer will use a list in place of an array in Python. Arrays in most programming languages were fixed-length, requiring the program to know the maximum number of elements needed ahead of time. This restriction leads to the infamous array-overrun bugs and security issues. The Python list is much more flexible in that the program can dynamically change the size of a list.

The next sections will look at each collection type in more detail.

Lists and Tuples

For a Python program, lists and tuples are very similar. Both lists and tuples hold an ordered collection of items. It is possible to get by as a programmer using only lists and ignoring tuples.

The primary difference that you will see syntactically is that a list is enclosed by square braces [], and a tuple is enclosed by parenthesis (). The following code defines both list and tuple.

In [1]:
```
l = ['a', 'b', 'c', 'd']
t = ('a', 'b', 'c', 'd')
```

```
print(l)
print(t)
```

```
['a', 'b', 'c', 'd']
('a', 'b', 'c', 'd')
```

The primary difference you will see programmatically is that a list is mutable, which means the program can change it. A tuple is immutable, which means the program cannot change it. The following code demonstrates that the program can change a list. This code also illustrates that Python indexes lists starting at element 0. Accessing element one modifies the second element in the collection. One advantage of tuples over lists is that tuples are generally slightly faster to iterate over than lists.

In [2]:
```
l[1] = 'chaged'
#t[1] = 'changed' # This would result in an error

print(l)
```

```
['a', 'chaged', 'c', 'd']
```

Like many languages, Python has a for-each statement. This statement allows you to loop over every element in a collection, such as a list or a tuple.

In [4]:
```
# Iterate over a collection.
for s in l:
    print(s)
```

```
a
changed
c
d
```

The **enumerate** function is useful for enumerating over a collection and having access to the index of the element that we are currently on.

In [5]:
```
# Iterate over a collection, and know where your index.  (Python is zero-bas
for i,l in enumerate(l):
    print(f"{i}:{l}")
```

```
0:a
1:changed
2:c
3:d
```

A **list** can have multiple objects added, such as strings. Duplicate values are allowed. **Tuples** do not allow the program to add additional objects after definition.

In [6]:
```
# Manually add items, lists allow duplicates
c = []
c.append('a')
c.append('b')
```

```
c.append('c')
c.append('c')
print(c)
```

```
['a', 'b', 'c', 'c']
```

Ordered collections, such as lists and tuples, allow you to access an element by its index number, as done in the following code. Unordered collections, such as dictionaries and sets, do not allow the program to access them in this way.

In [7]:
```
print(c[1])
```

```
b
```

A **list** can have multiple objects added, such as strings. Duplicate values are allowed. Tuples do not allow the program to add additional objects after definition. The programmer must specify an index for the insert function, an index. These operations are not allowed for tuples because they would result in a change.

In [8]:
```
# Insert
c = ['a', 'b', 'c']
c.insert(0, 'a0')
print(c)
# Remove
c.remove('b')
print(c)
# Remove at index
del c[0]
print(c)
```

```
['a0', 'a', 'b', 'c']
['a0', 'a', 'c']
['a', 'c']
```

Sets

A Python **set** holds an unordered collection of objects, but sets do *not* allow duplicates. If a program adds a duplicate item to a set, only one copy of each item remains in the collection. Adding a duplicate item to a set does not result in an error. Any of the following techniques will define a set.

In [9]:
```
s = set()
s = { 'a', 'b', 'c'}
s = set(['a', 'b', 'c'])
print(s)
```

```
{'c', 'a', 'b'}
```

A **list** is always enclosed in square braces [], a **tuple** in parenthesis (), and similarly a **set** is enclosed in curly braces. Programs can add items to a **set** as they run. Programs can dynamically add items to a **set** with the **add** function. It

is important to note that the **append** function adds items to lists, whereas the **add** function adds items to a **set**.

In [5]:
```
# Manually add items, sets do not allow duplicates
# Sets add, lists append.  I find this annoying.
c = set()
c.add('a')
c.add('b')
c.add('c')
c.add('c')
c.add('gc')
print(c)
```

```
{'a', 'b', 'gc', 'c'}
```

Maps/Dictionaries/Hash Tables

Many programming languages include the concept of a map, dictionary, or hash table. These are all very related concepts. Python provides a dictionary that is essentially a collection of name-value pairs. Programs define dictionaries using curly braces, as seen here.

In [8]:
```
d = {'name': "Jeff", 'address':"123 Main"}
print(d)
print(d['name'])

if 'name' in d:
    print("Name is defined")
    print(d['name'])
    print(d['address'])

if 'age' in d:
    print("age defined")
else:
    print("age undefined")
```

```
{'name': 'Jeff', 'address': '123 Main'}
Jeff
Name is defined
Jeff
123 Main
age undefined
```

Be careful that you do not attempt to access an undefined key, as this will result in an error. You can check to see if a key is defined, as demonstrated above. You can also access the dictionary and provide a default value, as the following code demonstrates.

In [12]:
```
d.get('unknown_key', 'default')
```

Out[12]:
```
'default'
```

You can also access the individual keys and values of a dictionary.

In [13]:
```
d = {'name': "Jeff", 'address':"123 Main"}
# All of the keys
print(f"Key: {d.keys()}")

# All of the values
print(f"Values: {d.values()}")
```

```
Key: dict_keys(['name', 'address'])
Values: dict_values(['Jeff', '123 Main'])
```

Dictionaries and lists can be combined. This syntax is closely related to JSON. Dictionaries and lists together are a good way to build very complex data structures. While Python allows quotes (") and apostrophe (') for strings, JSON only allows double-quotes ("). We will cover JSON in much greater detail later in this module.

The following code shows a hybrid usage of dictionaries and lists.

In [14]:
```
# Python list & map structures
customers = [
    {"name": "Jeff & Tracy Heaton", "pets": ["Wynton", "Cricket",
        "Hickory"]},
    {"name": "John Smith", "pets": ["rover"]},
    {"name": "Jane Doe"}
]

print(customers)

for customer in customers:
    print(f"{customer['name']}:{customer.get('pets', 'no pets')}")
```

```
[{'name': 'Jeff & Tracy Heaton', 'pets': ['Wynton', 'Cricket', 'Hickory']},
{'name': 'John Smith', 'pets': ['rover']}, {'name': 'Jane Doe'}]
Jeff & Tracy Heaton:['Wynton', 'Cricket', 'Hickory']
John Smith:['rover']
Jane Doe:no pets
```

The variable **customers** is a list that holds three dictionaries that represent customers. You can think of these dictionaries as records in a table. The fields in these individual records are the keys of the dictionary. Here the keys **name** and **pets** are fields. However, the field **pets** holds a list of pet names. There is no limit to how deep you might choose to nest lists and maps. It is also possible to nest a map inside of a map or a list inside of another list.

More Advanced Lists

Several advanced features are available for lists that this section introduces. One such function is **zip**. Two lists can be combined into a single list by the **zip**

command. The following code demonstrates the **zip** command.

In [9]:
```
a = [1,2,3,4,5]
b = [5,4,3,2,1]

print(zip(a,b))
```

```
<zip object at 0x7fc5d8419e60>
```

To see the results of the **zip** function, we convert the returned zip object into a list. As you can see, the **zip** function returns a list of tuples. Each tuple represents a pair of items that the function zipped together. The order in the two lists was maintained.

In [12]:
```
a = [1,2,3,4,5,9]
b = [5,4,3,2,1,3]

print(list(zip(a,b)))
```

```
[(1, 5), (2, 4), (3, 3), (4, 2), (5, 1), (9, 3)]
```

The usual method for using the **zip** command is inside of a for-loop. The following code shows how a for-loop can assign a variable to each collection that the program is iterating.

In [17]:
```
a = [1,2,3,4,5]
b = [5,4,3,2,1]

for x,y in zip(a,b):
    print(f'{x} - {y}')
```

```
1 - 5
2 - 4
3 - 3
4 - 2
5 - 1
```

Usually, both collections will be of the same length when passed to the **zip** command. It is not an error to have collections of different lengths. As the following code illustrates, the **zip** command will only process elements up to the length of the smaller collection.

In [18]:
```
a = [1,2,3,4,5]
b = [5,4,3]

print(list(zip(a,b)))
```

```
[(1, 5), (2, 4), (3, 3)]
```

Sometimes you may wish to know the current numeric index when a for-loop is iterating through an ordered collection. Use the **enumerate** command to track the index location for a collection element. Because the **enumerate** command

deals with numeric indexes of the collection, the zip command will assign arbitrary indexes to elements from unordered collections.

Consider how you might construct a Python program to change every element greater than 5 to the value of 5. The following program performs this transformation. The enumerate command allows the loop to know which element index it is currently on, thus allowing the program to be able to change the value of the current element of the collection.

In [19]:
```
a = [2, 10, 3, 11, 10, 3, 2, 1]
for i, x in enumerate(a):
    if x>5:
        a[i] = 5
print(a)
```

```
[2, 5, 3, 5, 5, 3, 2, 1]
```

The comprehension command can dynamically build up a list. The comprehension below counts from 0 to 9 and adds each value (multiplied by 10) to a list.

In [20]:
```
lst = [x*10 for x in range(10)]
print(lst)
```

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

A dictionary can also be a comprehension. The general format for this is:

```
dict_variable = {key:value for (key,value) in
dictonary.items()}
```

A common use for this is to build up an index to symbolic column names.

In [21]:
```
text = ['col-zero','col-one', 'col-two', 'col-three']
lookup = {key:value for (value,key) in enumerate(text)}
print(lookup)
```

```
{'col-zero': 0, 'col-one': 1, 'col-two': 2, 'col-three': 3}
```

This can be used to easily find the index of a column by name.

In [22]:
```
print(f'The index of "col-two" is {lookup["col-two"]}')
```

```
The index of "col-two" is 2
```

An Introduction to JSON

Data stored in a CSV file must be flat; it must fit into rows and columns. Most people refer to this type of data as structured or tabular. This data is tabular because the number of columns is the same for every row. Individual rows may

be missing a value for a column; however, these rows still have the same columns.

This data is convenient for machine learning because most models, such as neural networks, also expect incoming data to be of fixed dimensions. Real-world information is not always so tabular. Consider if the rows represent customers. These people might have multiple phone numbers and addresses. How would you describe such data using a fixed number of columns? It would be useful to have a list of these courses in each row that can be variable length for each row or student.

JavaScript Object Notation (JSON) is a standard file format that stores data in a hierarchical format similar to eXtensible Markup Language (XML). JSON is nothing more than a hierarchy of lists and dictionaries. Programmers refer to this sort of data as semi-structured data or hierarchical data. The following is a sample JSON file.

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

The above file may look somewhat like Python code. You can see curly braces that define dictionaries and square brackets that define lists. JSON does require

there to be a single root element. A list or dictionary can fulfill this role. JSON requires double-quotes to enclose strings and names. Single quotes are not allowed in JSON.

JSON files are always legal JavaScript syntax. JSON is also generally valid as Python code, as demonstrated by the following Python program.

In [23]:
```
jsonHardCoded = {
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": True,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": None
}
```

Generally, it is better to read JSON from files, strings, or the Internet than hard coding, as demonstrated here. However, for internal data structures, sometimes such hard-coding can be useful.

Python contains support for JSON. When a Python program loads a JSON the root list or dictionary is returned, as demonstrated by the following code.

In [24]:
```
import json

json_string = '{"first":"Jeff","last":"Heaton"}'
obj = json.loads(json_string)
print(f"First name: {obj['first']}")
print(f"Last name: {obj['last']}")
```

```
First name: Jeff
Last name: Heaton
```

Python programs can also load JSON from a file or URL.

In [25]:
```
import requests

r = requests.get("https://raw.githubusercontent.com/jeffheaton/"
                 +"t81_558_deep_learning/master/person.json")
print(r.json())
```

```
{'firstName': 'John', 'lastName': 'Smith', 'isAlive': True, 'age': 27, 'addr
ess': {'streetAddress': '21 2nd Street', 'city': 'New York', 'state': 'NY',
'postalCode': '10021-3100'}, 'phoneNumbers': [{'type': 'home', 'number': '21
2 555-1234'}, {'type': 'office', 'number': '646 555-4567'}, {'type': 'mobil
e', 'number': '123 456-7890'}], 'children': [], 'spouse': None}
```

Python programs can easily generate JSON strings from Python objects of dictionaries and lists.

In [26]:
```
python_obj = {"first":"Jeff","last":"Heaton"}
print(json.dumps(python_obj))
```

```
{"first": "Jeff", "last": "Heaton"}
```

A data scientist will generally encounter JSON when they access web services to get their data. A data scientist might use the techniques presented in this section to convert the semi-structured JSON data into tabular data for the program to use with a model such as a neural network.

In [ ]:
```

```

Open in Colab

T81-558: Applications of Deep Neural Networks

Module 1: Python Preliminaries

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 1 Material

- Part 1.1: Course Overview [Video] [Notebook]
- Part 1.2: Introduction to Python [Video] [Notebook]
- Part 1.3: Python Lists, Dictionaries, Sets and JSON [Video] [Notebook]
- **Part 1.4: File Handling** [Video] [Notebook]
- Part 1.5: Functions, Lambdas, and Map/Reduce [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow. Running the following code will map your GDrive to `/content/drive`.

In [ ]:
```
try:
    from google.colab import drive
    drive.mount('/content/drive', force_remount=True)
    COLAB = True
    print("Note: using Google CoLab")
    %tensorflow_version 2.x
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

Part 1.4: File Handling

Files often contain the data that you use to train your AI programs. Once trained, your models may use real-time data to form predictions. These predictions might

be made on files too. Regardless of predicting or training, file processing is a vital skill for the AI practitioner.

There are many different types of files that you must process as an AI practitioner. Some of these file types are listed here:

- **CSV files** (generally have the .csv extension) hold tabular data that resembles spreadsheet data.
- **Image files** (generally with the .png or .jpg extension) hold images for computer vision.
- **Text files** (often have the .txt extension) hold unstructured text and are essential for natural language processing.
- **JSON** (often have the .json extension) contain semi-structured textual data in a human-readable text-based format.
- **H5** (can have a wide array of extensions) contain semi-structured textual data in a human-readable text-based format. Keras and TensorFlow store neural networks as H5 files.
- **Audio Files** (often have an extension such as .au or .wav) contain recorded sound.

Data can come from a variety of sources. In this class, we obtain data from three primary locations:

- **Your Hard Drive** - This type of data is stored locally, and Python accesses it from a path that looks something like: **c:\data\myfile.csv or /Users/jheaton/data/myfile.csv**.
- **The Internet** - This type of data resides in the cloud, and Python accesses it from a URL that looks something like:

https://data.heatonresearch.com/data/t81-558/iris.csv.

- **Google Drive (cloud)** - If your code in Google CoLab, you use GoogleDrive to save and load some data files. CoLab mounts your GoogleDrive into a path similar to the following: **/content/drive/My Drive/myfile.csv**.

Read a CSV File

Python programs can read CSV files with Pandas. We will see more about Pandas in the next section, but for now, its general format is:

In [1]:
```
import pandas as pd

df = pd.read_csv("https://data.heatonresearch.com/data/t81-558/iris.csv")
```

The above command loads Fisher's Iris data set from the Internet. It might take a few seconds to load, so it is good to keep the loading code in a separate Jupyter notebook cell so that you do not have to reload it as you test your program. You can load Internet data, local hard drive, and Google Drive data this way.

Now that the data is loaded, you can display the first five rows with this command.

In [2]:
```
display(df[0:5])
```

	sepal_l	sepal_w	petal_l	petal_w	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Read (stream) a Large CSV File

Pandas will read the entire CSV file into memory. Usually, this is fine. However, at times you may wish to "stream" a huge file. Streaming allows you to process this file one record at a time. Because the program does not load all of the data into memory, you can handle huge files. The following code loads the Iris dataset and calculates averages, one row at a time. This technique would work for large files.

In [3]:
```
import csv
import urllib.request
import codecs
import numpy as np

url = "https://data.heatonresearch.com/data/t81-558/iris.csv"
urlstream = urllib.request.urlopen(url)
csvfile = csv.reader(codecs.iterdecode(urlstream, 'utf-8'))
next(csvfile) # Skip header row
sum = np.zeros(4)
count = 0

for line in csvfile:
    # Convert each row to Numpy array
    line2 = np.array(line)[0:4].astype(float)

    # If the line is of the right length (skip empty lines), then add
    if len(line2) == 4:
        sum += line2
        count += 1
```

```
# Calculate the average, and print the average of the 4 iris
# measurements (features)
print(sum/count)
```

```
[5.84333333 3.05733333 3.758      1.19933333]
```

Read a Text File

The following code reads the Sonnet 18 by William Shakespeare as a text file. This code streams the document and reads it line-by-line. This code could handle a huge file.

In [4]:
```
import urllib.request
import codecs

url = "https://data.heatonresearch.com/data/t81-558/datasets/sonnet_18.txt"
with urllib.request.urlopen(url) as urlstream:
    for line in codecs.iterdecode(urlstream, 'utf-8'):
        print(line.rstrip())
```

```
Sonnet 18 original text
William Shakespeare

Shall I compare thee to a summer's day?
Thou art more lovely and more temperate:
Rough winds do shake the darling buds of May,
And summer's lease hath all too short a date:
Sometime too hot the eye of heaven shines,
And often is his gold complexion dimm'd;
And every fair from fair sometime declines,
By chance or nature's changing course untrimm'd;
But thy eternal summer shall not fade
Nor lose possession of that fair thou owest;
Nor shall Death brag thou wander'st in his shade,
When in eternal lines to time thou growest:
So long as men can breathe or eyes can see,
So long lives this and this gives life to thee.
```

Read an Image

Computer vision is one of the areas that neural networks outshine other models. To support computer vision, the Python programmer needs to understand how to process images. For this course, we will use the Python PIL package for image processing. The following code demonstrates how to load an image from a URL and display it.

In [5]:
```
%matplotlib inline
from PIL import Image
import requests
from io import BytesIO
```

```
url = "https://data.heatonresearch.com/images/jupyter/brookings.jpeg"

response = requests.get(url)
img = Image.open(BytesIO(response.content))

img
```

Out[5]:

In [ ]:

Open in Colab

T81-558: Applications of Deep Neural Networks

Module 1: Python Preliminaries

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 1 Material

- Part 1.1: Course Overview [Video] [Notebook]
- Part 1.2: Introduction to Python [Video] [Notebook]
- Part 1.3: Python Lists, Dictionaries, Sets and JSON [Video] [Notebook]
- Part 1.4: File Handling [Video] [Notebook]
- **Part 1.5: Functions, Lambdas, and Map/Reduce** [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:
```
try:
    from google.colab import drive
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

Part 1.5: Functions, Lambdas, and Map/Reduce

Functions, **lambdas**, and **map/reduce** can allow you to process your data in advanced ways. We will introduce these techniques here and expand on them in the next module, which will discuss Pandas.

Function parameters can be named or unnamed in Python. Default values can also be used. Consider the following function.

In [2]:
```
def say_hello(speaker, person_to_greet, greeting = "Hello"):
    print(f'{greeting} {person_to_greet}, this is {speaker}.')

say_hello('Jeff', "John")
say_hello('Jeff', "John", "Goodbye")
say_hello(speaker='Jeff', person_to_greet="John", greeting = "Goodbye")
```

```
Hello John, this is Jeff.
Goodbye John, this is Jeff.
Goodbye John, this is Jeff.
```

A function is a way to capture code that is commonly executed. Consider the following function that can be used to trim white space from a string capitalize the first letter.

In [3]:
```
def process_string(str):
    t = str.strip()
    return t[0].upper()+t[1:]
```

This function can now be called quite easily.

In [4]:
```
str = process_string("  hello  ")
print(f'"{str}"')
```

```
"Hello"
```

Python's **map** is a very useful function that is provided in many different programming languages. The **map** function takes a **list** and applies a function to each member of the **list** and returns a second **list** that is the same size as the first.

In [5]:
```
l = ['   apple  ', 'pear ', 'orange', 'pine apple  ']
list(map(process_string, l))
```

Out[5]:
```
['Apple', 'Pear', 'Orange', 'Pine apple']
```

Map

The **map** function is very similar to the Python **comprehension** that we previously explored. The following **comprehension** accomplishes the same task as the previous call to **map**.

In [6]:
```
l = ['   apple  ', 'pear ', 'orange', 'pine apple  ']
l2 = [process_string(x) for x in l]
print(l2)
```

```
['Apple', 'Pear', 'Orange', 'Pine apple']
```

The choice of using a **map** function or **comprehension** is up to the programmer. I tend to prefer **map** since it is so common in other programming languages.

Filter

While a **map function** always creates a new **list** of the same size as the original, the **filter** function creates a potentially smaller **list**.

In [7]:
```
def greater_than_five(x):
    return x>5

l = [ 1, 10, 20, 3, -2, 0]
l2 = list(filter(greater_than_five, l))
print(l2)
```

```
[10, 20]
```

Lambda

It might seem somewhat tedious to have to create an entire function just to check to see if a value is greater than 5. A **lambda** saves you this effort. A lambda is essentially an unnamed function.

In [8]:
```
l = [ 1, 10, 20, 3, -2, 0]
l2 = list(filter(lambda x: x>5, l))
print(l2)
```

```
[10, 20]
```

Reduce

Finally, we will make use of **reduce**. Like **filter** and **map** the **reduce** function also works on a **list**. However, the result of the **reduce** is a single value. Consider if you wanted to sum the **values** of a **list**. The sum is implemented by a **lambda**.

In [9]:
```
from functools import reduce

l = [ 1, 10, 20, 3, -2, 0]
result = reduce(lambda x,y: x+y,l)
print(result)
```

```
32
```