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
```