Day 1: Python Basics

Objectives:

(1) Download and set up the following: Python 3.6, Pycharm IDE, installing python libraries with pip (pip3) in the terminal, setting up the project folder for this class.
(2) Understand Python basics: variable types, loops (including for and while), functions, proper use of return statements

Lesson Plan:

Module 1: Setting up Python

Setting up the Project Folder:
On their desktops, have students make the following folders on their desktop and organize them recursively like so:

```
ml-quick-start
    Day-1
    Day-2
    Day-3
```

Downloading Python 3.6:
Start by showing students where they can download python from the python official website (https://www.python.org/downloads/). Explain that this comes with one IDE as well—IDLE—which we won't be using, but is in some ways that are different from PyCharm. It's worth exploring the two different IDEs on their own as well.

Downloading & Setting up PyCharm Community Edition:
Next, we need to show students where they can download PyCharm from their website (https://www.jetbrains.com/pycharm/download/#section=mac).

Have students open the ml-quick-start folder once PyCharm starts up. Remember: do not make a new project, simply open the folder you already made. THIS will be your project folder.

From here we need to set up the python environment and compiler in PyCharm. It is possible to set up a virtual environment in this program, but for the sake of simplicity (and for the sake of having global usage of the libraries we'll be using in Python), we're going to use the global python environment you downloaded in step 1. Do the following:

(1) In the main menu up top, select PyCharm and then select "preferences"
(2) Click on "Project: ml-quick-start"
(3) Click on "Project Interpreter"
(4) In the drop down menu, select Python 3.6, or if it's not there, click browse.

(5) IFF (“browse”==True), go to your computer’s “/Library/” folder in the browser (this may take some guidance).
(6) IFF (“browse”==True), in “/Library/”, go to “Frameworks/Python.Frameworks/version/3.6”

Wham. Your project is all set now in PyCharm. Now we start coding.

15-minute break

Module 2: Basic Python

Variables & Print:
Have students start a new file in their project in the folder Day-1. Have them title it “basics.py”

Python has a few different native variable types. They are as follows: (1) int() (whole integers, like 1, and 456), float() (decimals, like .0004, and 500.232567), and string() (think: ‘Hello world!’). Now, we can assign any value we want to a variable to be called later. So we can write in our new .py program something like this:

```
a=1
b=2.444
c=’hello world!’
```

now, we can refer to the values for these items by typing the variable’s name. So if we type

```
c
```

and run our script, it should spit out

```
“hello world!”
```

Oh, the places you’ll go. We can do a few things with these as well. Say we want to add 3 to a. There are a few ways we can do this.

```
a+3 #prints out 4, but does not save the variable.
a=a+3 #saves the new value as the variable. This way is the most logical for beginners.
a+=3 #+= means that the original value for a is called, and then 3 is added. -= does the same, but subtracts.
```

We can also do this for floats:
```
b+=3.2
```

And strings:

c+='@#$!'

But what we can't do is add variables of different types to one another.

a+b ===== ERROR

We CAN convert one variable to another type though, and then do some math.
b+=float(a)

By calling the variable in float( ) we convert it to a float. But this can't work between strings like "hello" and numeric values . . . though stand-alone numbers in string format can be converted. Go figure.

Here are some other math functions you should know:

* = multiplication
/ = division
% = the remainder given after division.

Got that? Well now we need to talk about lists and dictionaries. Type in your screen the following

listy=[1,2,3,4]
dic={"a": 1, "b":2, "c":1}
tup=("a", 2)

a list is just that—a list of things, be it ints, floats, strings, or a combination there-in. We can call an item in a list according to it's position. Type in

listy[0] #returns 1

Wait a sec, isn't "1" in position 1? Not so. EVERYthing in Python starts at index 0. Remember that. What's great about lists is we can change the items in them in a few different ways. Type in the following:

listy.append('dufus!') # returns listy + 'dufus'
listy.remove(2) # returns listy without 3
listy[0]='hello' # returns listy after having changed 1 to 'hello'
listy.insert(2, 3) # returns listy after having inserted 3 after 2

We can call items in a tuple the same way, but we can't change anything in a tuple unlike in a list. Tuples are "immutable" for this reason.

tup[0] # returns 'a'

Dictionaries are divided into .keys and .values . . . the key is the first thing before the colon in each instance, and the value is whatever comes after it. Each key must be unique, but the value can be the same for multiple entries. This is really important later when we build a deep learning model. We can add items to a dictionary in a couple of ways, using either

```
dic.add({'c': 4})
```
or
```
dic + {'c': 4}.
```

Allow time for questions and discussion.

15-minute break

Module 3: Loops & Functions

Loops:
You need to know about the following important ways of recursively running a task. So here is the 'for' loop . . .

```
for k in range(5):
  print('Hello!')
```

Thus for every number in a range of 5, the python program will spit out "hello!"

"while" is a little trickier, and requires some Boolean logic. Check this out:

```
while x < 4:
  print('hello!')
  x+=1
```

Cool right? Some more Boolean commands you might want to explore are

<	greater than
<=	greater than or equal to
>=	less than or equal to
==	equals
!=	is not equal to

Functions:
So we don't want to write the same thing a thousand times, so we can create a function that we can call whenever we like. We do this like so:

```
def func():
  while x < 5:
    print('Hello!')
# outputs hello 4 times.
```

If we call this function multiple times, then . . . func() \n func() . . . we get eight 'hello!'s

Those parentheses are also important. We can add a variable there to be used in the function.

```
def func(thing):
  while x < 5:
    print(thing)
```

Remember though, you have to call the variable in the function, not whatever you're inputing. If you do

```
def func(thing):
  while x<4:
    print('Hello!')
```

that means it will output "hello" twice, and not 1 if that was your input.

Now what if we don't care to print it, but just want it to do some operation like multiplication? Now, my friend, we get to return . . .

```
def func(inty):
  b=3+inty
  return  b
```

don't want it to spit out the value but want to save the output? Assign this function to a variable.

```
yes=func(4)
```

Conclusion & Individual questions period.