Python for Machine Learning

by A4Ayub Data Science Labs (http://www.a4ayub.me (http://www.a4ayub.me))

Introduction

You will learn how to write and run python code and at the end of the session you will learn about:

1. Data Types and Variables
2. Strings and String Operations
3. Print Statements
4. List, Dictionaries and Tuples
5. Numpy and Arrays
6. Functions and Modules
7. If Else Try Except
8. Loops
9. Plotting with Matplotlib

Data Types and Variables

Variables

Variables in Python are defined with the assignment operator, the equals sign =. Variable names in Python must adhere to the following rules:

1. variable names must start with a letter
2. variable names can only contain letters, numbers and the underscore character _
3. variable names can not contain spaces or punctuation
4. variable names are not enclosed in quotes or brackets

Numeric Data Types

Integers

Integers are one of the Python data types. An integer is a whole number, negative, positive or zero. In Python, integer variables are defined by assigning a whole number to a variable. Python's type() function can be used to determine the data type of a variable.

In [1]:

```
a = 5
type(a)
```

Out[1]:

```
int
```

In [2]:

```
b = -2
type(b)
```

Out[2]:

```
int
```

In [3]:

```
z = 0
type(z)
```

Out[3]:

```
int
```

Floating Point Numbers

Floating point numbers or floats are another Python data type. Floats are decimals, positive, negative and zero. Floats can also be represented by numbers in scientific notation which contain exponents.

Both a lower case e or an upper case E can be used to define floats in scientific notation. In Python, a float can be defined using a decimal point . when a variable is assigned.

In [4]:

```
c = 6.2
type(c)
```

Out[4]:

```
float
```

In [5]:

```
d = -0.03
type(d)
```

Out[5]:

```
float
```

In [6]:

```
Na = 6.02e23
Na
```

Out[6]:

```
6.02e+23
```

In [7]:

```
type(Na)
```

Out[7]:

```
float
```

Complex Numbers

Another useful numeric data type for problem solvers is the complex number data type. A complex number is defined in Python using a real component + an imaginary component j. The letter j must be used to denote the imaginary component. Using the letter i to define a complex number returns an error in Python.

In [8]:

```
comp = 4 + 2j
type(comp)
```

Out[8]:

```
complex
```

Boolean Data Types

The boolean data type is either True or False. In Python, boolean variables are defined by the True and False keywords.

In [9]:

```
a = True
type(a)
```

Out[9]:

```
bool
```

In [10]:

```
b = False
type(b)
```

Out[10]:

```
bool
```

Integers and floating point numbers can be converted to the boolean data type using Python's bool() function. An int, float or complex number set to zero returns False. An integer, float or complex number set to any other number, positive or negative, returns True.

In [11]:

```
zero_int = 0
bool(zero_int)
```

Out[11]:

```
False
```

In [12]:

```
pos_int = 1
bool(pos_int)
```

Out[12]:

```
True
```

In [13]:

```
neg_flt = -5.1
bool(neg_flt)
```

Out[13]:

```
True
```

Boolean arithmetic is the arithmetic of true and false logic. A boolean or logical value can either be True or False. Boolean values can be manipulated and combined with boolean operators. Boolean operators in Python include and, or, and not.

The common boolean operators in Python are below:

1. or
2. and
3. not
4. == (equivalent)
5. != (not equivalent)

In the code section below, two variables are assigned the boolean values True and False. Then these boolean values are combined and manipulated with boolean operators.

In [14]:

```
A = True
B = False
```

In [15]:

```
A or B
```

Out[15]:

```
True
```

In [16]:

```
A and B
```

Out[16]:

```
False
```

In [17]:

```
not A
```

Out[17]:

```
False
```

In [18]:

```
not B
```

Out[18]:

```
True
```

In [19]:

```
A == B
```

Out[19]:

```
False
```

In [20]:

```
A != B
```

Out[20]:

```
True
```

Boolean operators such as and, or, and not can be combined with parenthesis to make compound boolean expressions.

In [21]:

```
C = False
```

In [22]:

```
A or (C and B)
```

Out[22]:

```
True
```

In [23]:

```
(A and B) or C
```

Out[23]:

```
False
```

Strings

Strings are sequences of letters, numbers, punctuation, and spaces. Strings are defined at the Python REPL by enclosing letters, numbers, punctuation, and spaces in single quotes ' ' or double quotes " ".

Another built-in Python data type is strings. Strings are sequences of letters, numbers, symbols, and spaces. In Python, strings can be almost any length and can contain spaces. Strings are assigned in Python using single quotation marks ' ' or double quotation marks " ".

Python strings can contain blank spaces. A blank space is a valid character in Python string.

In [24]:

```
string = 'z'
```

In [25]:

```
type(string)
```

Out[25]:

```
str
```

Numbers and decimals can be defined as strings too. If a decimal number is defined using quotes ' ', the number is saved as a string rather than as a float. Integers defined using quotes become strings as well.

In [26]:

```
num = '5.2'
```

In [27]:

```
type(num)
```

Out[27]:

```
str
```

Strings can be converted to boolean values (converted to True or False). The empty string "" returns as False. All other strings convert to True.

In [28]:

```
name = "Gabby"
```

In [29]:

```
bool(name)
```

Out[29]:

```
True
```

In [30]:

```
empty = ""
```

In [31]:

```
bool(empty)
```

Out[31]:

```
False
```

Note that a string which contains just one space (" ") is not empty. It contains the space character. Therefore a string made up of just one space converts to True.

In [32]:

```
space = " "
```

In [33]:

```
bool(space)
```

Out[33]:

```
True
```

String indexing is the process of pulling out specific characters from a string in a particular order. In Python, strings are indexed using square brackets [ ]. An important point to remember:

Python counting starts at 0 and ends at n-1.

In [34]:

```
# Consider the word Solution
word = 'Solution'
word[0]
```

Out[34]:

```
'S'
```

In [35]:

```
word[1]
```

Out[35]:

```
'o'
```

In [36]:

```
word[7]
```

Out[36]:

```
'n'
```

Placing a negative number inside of the square brackets pulls a character out of a string starting from the end of the string.

In [37]:

```
word[-1]
```

Out[37]:

```
'n'
```

In [38]:

```
word[-2]
```

Out[38]:

```
'o'
```

String slicing is an operation to pull out a sequence of characters from a string. In Python, a colon on the inside of the square brackets between two numbers in a slicing operation indicates through. If the index [0:3] is called, the characters at positions 0 through 3 are returned.

In [39]:

```
word[0:3]
```

Out[39]:

```
'Sol'
```

A colon by itself on the inside of square brackets indicates all.

In [40]:

```
word[:]
```

Out[40]:

```
'Solution'
```

When three numbers are separated by two colons inside of square brackets, the numbers represent start : stop : step. Remember that Python counting starts at 0 and ends at n-1.

In [41]:

```
word[0:7:2]  #start:stop:step
```

Out[41]:

```
'Slto'
```

When two colons are used inside of square brackets, and less than three numbers are specified, the missing numbers are set to their "defaults". The default start is 0, the default stop is n-1, and the default step is 1.

The two code lines below produce the same output since 0 is the default start and 7 (n-1) is the default stop. Both lines of code use a step of 2.

In [42]:

```
word[0:7:2]
```

Out[42]:

```
'Slto'
```

In [43]:

```
word[::2]
```

Out[43]:

```
'Slto'
```

The characters that make up a string can be reversed by using the default start and stop values and specifying a step of -1.

In [44]:

```
word[::-1]
```

Out[44]:

```
'noituloS'
```

Strings can be concatenated or combined using the + operator.

In [48]:

```
another_word = "another solution"
third_word = "3rd solution!"
all_words = word+" "+another_word+" "+third_word
all_words
```

Out[48]:

```
'Solution another solution 3rd solution!'
```

Strings can be compared using the comparison operator; the double equals sign ==. Note the comparison operator (double equals ==) is not the same as the assignment operator, a single equals sign =.

In [49]:

```
name1 = 'Gabby'
name2 = 'Gabby'
name1 == name2
```

Out[49]:

```
True
```

In [50]:

```
name1 = 'Gabby'
name2 = 'Maelle'
name1 == name2
```

Out[50]:

```
False
```

Capital letters and lower case letters are different characters in Python. A string with the same letters, but different capitalization are not equivalent.

In [51]:

```
name1 = 'Maelle'
name2 = 'maelle'
name1 == name2
```

Out[51]:

```
False
```

Print Statements

One built-in function in Python is print(). The value or expression inside of the parenthesis of a print() function "prints" out to the REPL when the print() function is called.

In [52]:

```
name = "Gabby"
print("Your name is: {} ".format(name))
```

```
Your name is: Gabby
```

Expressions passed to the print() function are evaluated before they are printed out. For instance, the sum of two numbers can be shown with the print() function.

In [53]:

```
print(1+2)
```

```
3
```

If you want to see the text 1+2, you need to define "1+2" as a string and print out the string "1+2" instead.

In [54]:

```
print("1+2")
```

```
1+2
```

Strings can be concatenated (combined) inside of a print() statement.

In [56]:

```
name = "Gabby"
print('Your name is: ' + name)
```

```
Your name is: Gabby
```

The print() function also prints out individual expressions one after another with a space in between when the expressions are placed inside the print() function and separated by a comma.

In [57]:

```
print("Name:","Gabby","Age", 2+7)
```

```
Name: Gabby Age 9
```

Lists, Dictionaries and Tuples

Lists

A list is a data structure in Python that can contain multiple elements of any of the other data type. A list is defined with square brackets [ ] and commas , between elements.

In [58]:

```
lst = [ 1, 2, 3 ]
type(lst)
```

Out[58]:

```
list
```

In [59]:

```
lst = [ 1, 5.3, '3rd_Element']
type(lst)
```

Out[59]:

```
list
```

Individual elements of a list can be accessed or indexed using bracket [ ] notation. Note that Python lists start with the index zero, not the index 1. For example:

In [60]:

```
lst = ['statics', 'strengths', 'dynamics']
lst[0]
```

Out[60]:

```
'statics'
```

In [61]:

```
lst[1]
```

Out[61]:

```
'strengths'
```

In [62]:

```
lst[2]
```

Out[62]:

```
'dynamics'
```

Slicing Lists

In [63]:

```
lst = [2, 4, 6]
lst[:]
```

Out[63]:

```
[2, 4, 6]
```

Negative numbers can be used as indexes to call the last number of elements in the list

In [65]:

```
lst = [2, 4, 6]
lst[-1]
```

Out[65]:

```
6
```

The colon operator can also be used to denote all up to and thru end.

In [66]:

```
lst = [2, 4, 6]
lst[:2]          # all up to 2
```

Out[66]:

```
[2, 4]
```

In [67]:

```
lst = [2, 4, 6]
lst[2:]         # 2 thru end
```

Out[67]:

```
[6]
```

The colon operator can also be used to denote start : end + 1. Note that indexing here in not inclusive. lst[1:3] returns the 2nd element, and 3rd element but not the fourth even though 3 is used in the index.

Remember! Python indexing is not inclusive. The last element called in an index will not be returned.

Dictionaries

Dictionaries are made up of key: value pairs. In Python, lists and tuples are organized and accessed based on position. Dictionaries in Python are organized and accessed using keys and values. The location of a pair of keys and values stored in a Python dictionary is irrelevant.

Dictionaries are defined in Python with curly braces { }. Commas separate the key-value pairs that make up the dictionary. Each key-value pair is related by a colon :.

Let's store the ages of two people in a dictionary. The two people are Gabby and Maelle. Gabby is 8 and Maelle is 5. Note the name Gabby is a string and the age 8 is an integer.

In [73]:

```
age_dict = {"Gabby": 8 , "Maelle": 5}
type(age_dict)
```

Out[73]:

```
dict
```

The values stored in a dictionary are called and assigned using the following syntax: **dict_name[key] = value**

In [74]:

```
age_dict = {"Gabby": 8 , "Maelle": 5}
age_dict["Gabby"]
```

Out[74]:

```
8
```

We can add a new person to our age_dict with the following command:

In [75]:

```
age_dict = {"Gabby": 8 , "Maelle": 5}
age_dict["Peter"]= 40
age_dict
```

Out[75]:

```
{'Gabby': 8, 'Maelle': 5, 'Peter': 40}
```

Dictionaries can be converted to lists by calling the .items(), .keys(), and .values() methods.

In [76]:

```
age_dict = {"Gabby": 8 , "Maelle": 5}
whole_list = list(age_dict.items())
whole_list
```

Out[76]:

```
[('Gabby', 8), ('Maelle', 5)]
```

In [77]:

```
name_list = list(age_dict.keys())
name_list
```

Out[77]:

```
['Gabby', 'Maelle']
```

In [79]:

```
age_list = list(age_dict.values())
age_list
```

Out[79]:

```
[8, 5]
```

Items can be removed from dictionaries by calling the .pop() method. The dictionary key (and that key's associated value) supplied to the .pop() method is removed from the dictionary.

In [81]:

```
age_dict = {"Gabby": 8 , "Maelle": 5}
age_dict.pop("Gabby")
age_dict
```

Out[81]:

```
{'Maelle': 5}
```

Tuples

Tuples are immutable lists. Elements of a list can be modified, but elements in a tuple can only be accessed, not modified. The name tuple does not mean that only two values can be stored in this data structure.

Tuples are defined in Python by enclosing elements in parenthesis ( ) and separating elements with commas. The command below creates a tuple containing the numbers 3, 4, and 5.

In [68]:

```
t_var = (3,4,5)
t_var
```

Out[68]:

```
(3, 4, 5)
```

Note how the elements of a list can be modified:

In [69]:

```
l_var = [3,4,5]  # a list
l_var[0]= 8
l_var
```

Out[69]:

```
[8, 4, 5]
```

The elements of a tuple can not be modified. If you try to assign a new value to one of the elements in a tuple, an error is returned.

In [70]:

```
t_var = (3,4,5)  # a tuple
t_var[0]= 8
t_var
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-70-9a804ce58fc6> in <module>
      1 t_var = (3,4,5)  # a tuple
----> 2 t_var[0]= 8
      3 t_var

TypeError: 'tuple' object does not support item assignment
```

To create a tuple that just contains one numerical value, the number must be followed by a comma. Without a comma, the variable is defined as a number.

In [71]:

```
num = (5)
type(num)
```

Out[71]:

```
int
```

When a comma is included after the number, the variable is defined as a tuple.

In [72]:

```
t_var = (5,)
type(t_var)
```

Out[72]:

```
tuple
```

Numpy and Arrays

Numpy

NumPy is a Python package used for numerical computation. NumPy is one of the foundational packages for scientific computing with Python. NumPy's core data type is the array and NumPy functions operate on arrays.

In [84]:

```
# Verify NumPy installation
import numpy as np
np.__version__
```

Out[84]:

```
'1.16.4'
```

NumPy is used to construct homogeneous arrays and perform mathematical operations on arrays. A NumPy array is different from a Python list. The data types stored in a Python list can all be different.

In [85]:

```
python_list = [ 1, -0.038, 'gear', True]
```

The Python list above contains four different data types: 1 is an integer, -0.038 is a float, 'gear' is a string, and 'True' is a boolean.

The code below prints the data type of each value store in python_list.

In [86]:

```
for item in python_list:
    print(type(item))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

The values stored in a NumPy array must all share the same data type. Consider the NumPy array below:

In [87]:

```
np.array([1.0, 3.1, 5e-04, 0.007])
```

Out[87]:

```
array([1.0e+00, 3.1e+00, 5.0e-04, 7.0e-03])
```

All four values stored in the NumPy array above share the same data type: 1.0, 3.1, 5e-04, and 0.007 are all floats.

The code below prints the data type of each value stored in the NumPy array above.

In [88]:

```
import numpy as np

for value in np.array([1.0, 3.1, 5e-04, 0.007]):
    print(type(value))
```

```
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
```

In the next code section, all four items are converted to type '<U32', which is a string data type in NumPy (the U refers Unicode strings; all strings in Python are Unicode by default).

In [89]:

```
np.array([1, -0.038, 'gear', True])
```

Out[89]:

```
array(['1', '-0.038', 'gear', 'True'], dtype='<U32')
```

NumPy arrays can also be two-dimensional, three-dimensional, or up to n-dimensional. In practice, computer resources limit array size. Remember that regardless of size, all elements in a NumPy array must be the same type. NumPy arrays are useful because mathematical operations can be run on an entire array simultaneously. If numbers are stored in a regular Python list and the list is multiplied by a scalar, the list extends and repeats- instead of multiplying each number in the list by the scalar.

The code below demonstrates list repetition using the multiplication operator, *.

In [90]:

```
lst = [1, 2, 3, 4]
lst*2
```

Out[90]:

```
[1, 2, 3, 4, 1, 2, 3, 4]
```

To multiply each element in a Python list by the number 2, a loop can be used:

In [91]:

```
lst = [1, 2, 3, 4]
for i, item in enumerate(lst):
    lst[i] = lst[i]*2
lst
```

Out[91]:

```
[2, 4, 6, 8]
```

The method above is relatively cumbersome and is also quite computationally expensive. An operation that is computationally expensive is an operation that takes a lot of processing time or storage resources like RAM and CPU bandwidth.

Another way to complete the same operation in the loop above is to use a NumPy array.

An entire NumPy array can be multiplied by a scalar in one step. The scalar multiplication operation below produces an array with each element multiplied by the scalar 2.

In [92]:

```
nparray = np.array([1,2,3,4])
2*nparray
```

Out[92]:

```
array([2, 4, 6, 8])
```

If we have a very long list of numbers, we can compare the amount of time it takes each of the two computation methods above, a list with a loop compared to array multiplication to complete the same operation. This comparison highlights an advantage of arrays compared to lists- speed.

Jupyter notebooks have a nice built-in method to time how long a line of code takes to execute. In a Jupyter notebook, when a line starts with %timeit followed by code, the kernel runs the line of code multiple times and outputs an average of the time spent to execute the line of code.

We can use %timit to compare a mathematical operation on a Python list using a for loop to the same mathematical operation on a NumPy array.

In [93]:

```
lst = list(range(10000))
%timeit for i, item in enumerate(lst): lst[i] = lst[i]*2
```

```
3 ms ± 79.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [94]:

```
nparray = np.arange(0,10000,1)
%timeit 2*nparray
```

```
16.2 µs ± 1.92 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

With 10,000 integers, the Python list and for loop takes an average of single milliseconds, while the NumPy array completes the same operation in tens of microseconds. This is a speed increase of over 100x by using the NumPy array (1 millisecond = 1000 microseconds).

For larger lists of numbers, the speed increase using NumPy is considerable.

NumPy arrays are created with the np.array() function. The arguments provided to np.array() needs to be a list or iterable. An example is below. Note how the list [1,2,3] is passed into the function with square brackets at either end.

In [95]:

```
import numpy as np
np.array([1,2,3])
```

Out[95]:

```
array([1, 2, 3])
```

The data type can be passed into the np.array() function as a second optional keyword argument. Available data types include 'int64', 'float', 'complex' and '>U32' (a string data type).

In [96]:

```
import numpy as np
np.array([1,2,3], dtype='float')
```

Out[96]:

```
array([1., 2., 3.])
```

The data type used in a NumPy array can be determined using the .dtype attribute. For instance, an array of floats returns float64.

In [97]:

```
import numpy as np
my_array = np.array([1,2,3], dtype='float')
my_array.dtype
```

Out[97]:

```
dtype('float64')
```

There are multiple ways to create arrays of regularly spaced numbers with NumPy. The next section introduces five NumPy functions to create regular arrays.

1. NumPy's np.arange() function creates a NumPy array according the arguments start, stop,step.
2. NumPy's np.linspace() function creates a NumPy array according the arguments start, stop,number of elements.
3. NumPy's np.logspace() function creates a NumPy array according the arguments start, stop,number of elements, but unlike np.linspace(), np.logspace() produces a logarithmically spaced array.
4. NumPy's np.zeros() function creates a NumPy array containing all zeros of a specific size. np.zeros() is useful when the size of an array is known, but the values that will go into the array have not been created yet.
5. NumPy's np.ones() function creates a NumPy array containing all 1's of a specific size. Like np.zeros(), np.ones() is useful when the size of an array is known, but the values that will go into the array have not been created yet.

In [98]:

```
np.arange(0,10+2,2)
```

Out[98]:

```
array([ 0,  2,  4,  6,  8, 10])
```

In [99]:

```
np.linspace(0,2*np.pi,10)
```

Out[99]:

```
array([0.        , 0.6981317 , 1.3962634 , 2.0943951 , 2.7925268 ,
       3.4906585 , 4.1887902 , 4.88692191, 5.58505361, 6.28318531])
```

In [100]:

```
np.logspace(1, 2, 4)
```

Out[100]:

```
array([ 10.        ,  21.5443469 ,  46.41588834, 100.        ])
```

In [101]:

```
np.zeros((5,5))
```

Out[101]:

```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

In [102]:

```
np.ones((3,5))
```

Out[102]:

```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

Arrays of random integers can be created with NumPy's np.random.randint() function. The general syntax is: **np.random.randint(lower limit, upper limit, number of values)**

In [103]:

```
np.random.randint(0,10,5)
```

Out[103]:

```
array([4, 8, 8, 2, 2])
```

Array dimensions can be provided as the third argument to the np.random.randint() function. The code below creates a 5 × 5 array of random numbers between 1 and 10:

In [104]:

```
np.random.randint(0,10,[5,5])
```

Out[104]:

```
array([[1, 7, 9, 7, 2],
       [4, 3, 2, 0, 0],
       [2, 3, 1, 2, 0],
       [2, 3, 1, 1, 6],
       [2, 1, 9, 0, 6]])
```

Arrays of random floating point numbers can be created with NumPy's np.random.rand() function. The general syntax is: **np.random.rand(number of values)**

To create an array of 5 random floats between 0 and 1:

In [105]:

```
np.random.rand(5)
```

Out[105]:

```
array([0.22079379, 0.49713726, 0.73449423, 0.61021137, 0.11927434])
```

The upper and lower ranges of random floats can me modified with arithmetic. To expand the range of random floats to between 0 and 10, multiply the result by 10

In [106]:

```
np.random.rand(5)*10
```

Out[106]:

```
array([5.46161263, 7.90340885, 9.6932633 , 9.5485654 , 2.30422821])
```

To change the range to between 11 and 13, we multiply the range by 2 (range 0-2), then add 11 to the result.

In [107]:

```
np.random.rand(5)*2+11
```

Out[107]:

```
array([11.23583947, 11.62705605, 11.31013985, 11.80913957, 12.09658604])
```

To choose three numbers at random from a list of [1,5,9,11] use:

In [108]:

```
lst = [1,5,9,11]
np.random.choice(lst,3)
```

Out[108]:

```
array([9, 1, 9])
```

np.random.randn() returns an array of random numbers with a normal distribution, assuming a mean of 0 and variance of 1.

In [109]:

```
np.random.randn(10)
```

Out[109]:

```
array([ 0.18868706,  0.07805909,  0.02967353, -0.07259888, -0.92500883,
       -0.07280422,  0.7855801 ,  0.56112494,  0.44749731, -1.26536995])
```

To specify a mean mu and a standard deviation sigma, the function can be wrapped with:

In [110]:

```
mu = 70
sigma = 6.6

sigma * np.random.randn(10) + mu
```

Out[110]:

```
array([83.0804913 , 75.68371124, 73.92503757, 73.43640786, 64.13778796,
       77.04756358, 71.57029886, 73.69841323, 70.46194768, 59.88281495])
```

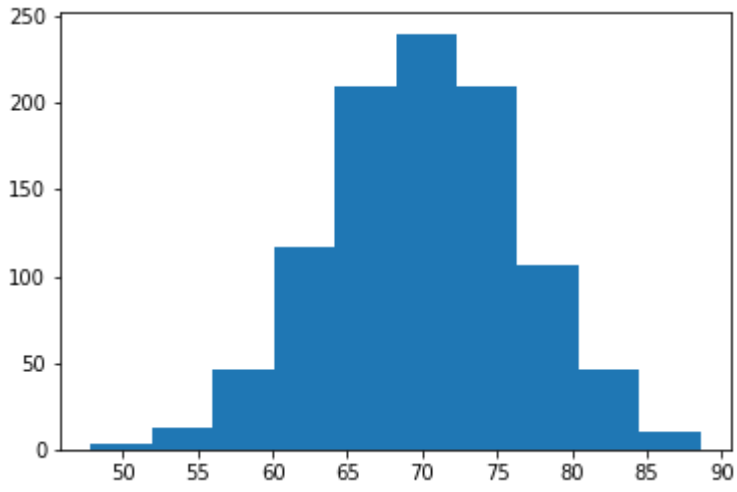
Matplotlib's plt.hist() function can be used to quickly plot a normal distribution created with NumPy's np.random.randn() function.

In [111]:

```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

mu = 70
sigma = 6.6


sample = sigma * np.random.randn(1000) + mu
plt.hist(sample)
plt.show()
```



NumPy's np.meshgrid() function takes in two positional arguments which are 1D NumPy arrays. The two input arrays do not have to contain the same number of elements. The outputs of the np.meshgrid() function are two 2D arrays.

One of the 2D arrays has the same values in each row; the other 2D array has the same values in each column.

In [113]:

```
x = np.arange(0,6)
y = np.arange(0,11,2)
X, Y = np.meshgrid(x,y)
print(X)
print(Y)
```

```
[[0 1 2 3 4 5]
 [0 1 2 3 4 5]
 [0 1 2 3 4 5]
 [0 1 2 3 4 5]
 [0 1 2 3 4 5]
 [0 1 2 3 4 5]]
[[ 0  0  0  0  0  0]
 [ 2  2  2  2  2  2]
 [ 4  4  4  4  4  4]
 [ 6  6  6  6  6  6]
 [ 8  8  8  8  8  8]
 [10 10 10 10 10 10]]
```

Note how the first array X has the same numbers in each row, and the second array Y has the same numbers in each column.

NumPy's np.mgrid[] function is similar to np.meshgrid(), but has a "MATLAB-like" syntax and behavior.

Use square brackets [ ] after the np.mgrid function name. Separate the two "lists" passed as input arguments with a comma and use the start:stop:step indexing method. The outputs of the np.mgrid[] function are two 2D arrays. The first 2D array has the same values in each row; the second 2D array has the same values in each column.

In [114]:

```
X, Y = np.mgrid[0:5,0:11:2]
print(X)
print(Y)
```

```
[[0 0 0 0 0 0]
 [1 1 1 1 1 1]
 [2 2 2 2 2 2]
 [3 3 3 3 3 3]
 [4 4 4 4 4 4]]
[[ 0  2  4  6  8 10]
 [ 0  2  4  6  8 10]
 [ 0  2  4  6  8 10]
 [ 0  2  4  6  8 10]
 [ 0  2  4  6  8 10]]
```

Elements in NumPy arrays can be accessed by indexing. Indexing is an operation that pulls out a select set of values from an array. The index of a value in an array is that value's location within the array. There is a difference between the value and where the value is stored in an array.

An array with 3 values is created in the code section below.

In [115]:

```
import numpy as np

a = np.array([2,4,6])
print(a)
```

```
[2 4 6]
```

The array above contains three values: 2, 4 and 6. Each of these values has a different index. Remember counting in Python starts at 0 and ends at n-1.

The value 2 has an index of 0. We could also say 2 is in location 0 of the array. The value 4 has an index of 1 and the value 6 has an index of 2. The table below shows the index (or location) of each value in the array.

In [116]:

```
value = a[2]
print(value)
```

```
6
```

Multi-dimensional arrays can be indexed as well. A simple 2-D array is defined by a list of lists.

In [117]:

```
import numpy as np

a = np.array([[2,3,4],[6,7,8]])
print(a)
```

```
[[2 3 4]
 [6 7 8]]
```

We can access the value 8 in the array above by calling the row and column index [1,2]. This corresponds to the 2nd row (remember row 0 is the first row) and the 3rd column (column 0 is the first column).

In [118]:

```
import numpy as np

a = np.array([[2,3,4],[6,7,8]])
print(a)
value = a[1,2]
print(value)
```

```
[[2 3 4]
 [6 7 8]]
8
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: