IntroductionToCourse

November 15, 2021

---

*You are currently looking at **version 1.1** of this notebook. To download notebooks and datafiles, as well as get help on Jupyter notebooks in the Coursera platform, visit the Jupyter Notebook FAQ course resource.*

---

1 The Python Programming Language: Functions

[1]:
```
x = 1
y = 2
x + y
```

[1]:
```
3
```

[2]:
```
y
```

[2]:
```
2
```

`add_numbers` is a function that takes two numbers and adds them together.

[3]:
```
def add_numbers(x, y):
    return x + y

add_numbers(1, 2)
```

[3]:
```
3
```

'add_numbers' updated to take an optional 3rd parameter. Using `print` allows printing of multiple expressions within a single cell.

[4]:
```
def add_numbers(x,y,z=None):
    if (z==None):
        return x+y
    else:
        return x+y+z

print(add_numbers(1, 2))
print(add_numbers(1, 2, 3))
```

```
3
6
```

add_numbers updated to take an optional flag parameter.

[5]:
```
def add_numbers(x, y, z=None, flag=False):
    if (flag):
        print('Flag is true!')
    if (z==None):
        return x + y
    else:
        return x + y + z

print(add_numbers(1, 2, flag=True))
```

```
Flag is true!
3
```

Assign function add_numbers to variable a.

[6]:
```
def add_numbers(x,y):
    return x+y

a = add_numbers
a(1,2)
```

[6]:
```
3
```

2 The Python Programming Language: Types and Sequences

Use type to return the object's type.

[7]:
```
type('This is a string')
```

[7]:
```
str
```

[8]:
```
type(None)
```

[8]:
```
NoneType
```

[9]:
```
type(1)
```

[9]:
```
int
```

[10]:
```
type(1.0)
```

[10]:
```
float
```

[11]:
```
type(add_numbers)
```

[11]:
```
function
```

Tuples are an immutable data structure (cannot be altered).

[12]:
```
x = (1, 'a', 2, 'b')
type(x)
```

[12]:
```
tuple
```

Lists are a mutable data structure.

[13]:
```
x = [1, 'a', 2, 'b']
type(x)
```

[13]:
```
list
```

Use append to append an object to a list.

[14]:
```
x.append(3.3)
print(x)
```

```
[1, 'a', 2, 'b', 3.3]
```

This is an example of how to loop through each item in the list.

[15]:
```
for item in x:
    print(item)
```

```
1
a
2
b
3.3
```

Or using the indexing operator:

[16]:
```
i=0
while( i != len(x) ):
    print(x[i])
    i = i + 1
```

```
1
a
2
b
3.3
```

Use + to concatenate lists.

[17]:
```
[1,2] + [3,4]
```

[17]:
```
[1, 2, 3, 4]
```

Use * to repeat lists.

[18]:
```
[1]*3
```

[18]:
```
[1, 1, 1]
```

Use the `in` operator to check if something is inside a list.

[19]:
```
1 in [1, 2, 3]
```

[19]:
```
True
```

Now let's look at strings. Use bracket notation to slice a string.

[20]:
```python
x = 'This is a string'
print(x[0]) #first character
print(x[0:1]) #first character, but we have explicitly set the end character
print(x[0:2]) #first two characters
```

```
T
T
Th
```

This will return the last element of the string.

[21]:
```python
x[-1]
```

[21]:
```
'g'
```

This will return the slice starting from the 4th element from the end and stopping before the 2nd element from the end.

[22]:
```python
x[-4:-2]
```

[22]:
```
'ri'
```

This is a slice from the beginning of the string and stopping before the 3rd element.

[23]:
```python
x[:3]
```

[23]:
```
'Thi'
```

And this is a slice starting from the 4th element of the string and going all the way to the end.

[24]:
```python
x[3:]
```

[24]:
```
's is a string'
```

[25]:
```python
firstname = 'Christopher'
lastname = 'Brooks'

print(firstname + ' ' + lastname)
print(firstname*3)
print('Chris' in firstname)
```

```
Christopher Brooks
ChristopherChristopherChristopher
True
```

split returns a list of all the words in a string, or a list split on a specific character.

[26]:
```python
firstname = 'Christopher Arthur Hansen Brooks'.split(' ')[0] # [0] selects the␣
 ↪first element of the list
lastname = 'Christopher Arthur Hansen Brooks'.split(' ')[-1] # [-1] selects the␣
 ↪last element of the list
print(firstname)
print(lastname)
```

```
Christopher
Brooks
```

Make sure you convert objects to strings before concatenating.

[27]:
```
'Chris' + 2
```

```
                                                                              ␣
↪---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call␣
↪last)

        <ipython-input-27-9d01956b24db> in <module>
    ----> 1 'Chris' + 2


        TypeError: can only concatenate str (not "int") to str
```

[ ]:
```
'Chris' + str(2)
```

Dictionaries associate keys with values.

[ ]:
```
x = {'Christopher Brooks': 'brooksch@umich.edu', 'Bill Gates': 'billg@microsoft.
↪com'}
x['Christopher Brooks'] # Retrieve a value by using the indexing operator
```

[ ]:
```
x['Kevyn Collins-Thompson'] = None
x['Kevyn Collins-Thompson']
```

Iterate over all of the keys:

[ ]:
```
for name in x:
    print(x[name])
```

Iterate over all of the values:

[ ]:
```
for email in x.values():
    print(email)
```

Iterate over all of the items in the list:

[ ]:
```
for name, email in x.items():
    print(name)
    print(email)
```

You can unpack a sequence into different variables:

[ ]:
```
x = ('Christopher', 'Brooks', 'brooksch@umich.edu')
fname, lname, email = x
```

[ ]:
```
fname
```

[ ]:
```
lname
```

Make sure the number of values you are unpacking matches the number of variables being assigned.

[ ]:
```
x = ('Christopher', 'Brooks', 'brooksch@umich.edu', 'Ann Arbor')
fname, lname, email = x
```

3 The Python Programming Language: More on Strings

[ ]:
```
print('Chris' + 2)
```

[ ]:
```
print('Chris' + str(2))
```

Python has a built in method for convenient string formatting.

[ ]:
```
sales_record = {
'price': 3.24,
'num_items': 4,
'person': 'Chris'}

sales_statement = '{} bought {} item(s) at a price of {} each for a total of {}'

print(sales_statement.format(sales_record['person'],
                             sales_record['num_items'],
                             sales_record['price'],
                             sales_record['num_items']*sales_record['price']))
```

4 Reading and Writing CSV files

Let's import our datafile mpg.csv, which contains fuel economy data for 234 cars.

- mpg : miles per gallon
- class : car classification
- cty : city mpg
- cyl : # of cylinders
- displ : engine displacement in liters
- drv : f = front-wheel drive, r = rear wheel drive, 4 = 4wd
- fl : fuel (e = ethanol E85, d = diesel, r = regular, p = premium, c = CNG)
- hwy : highway mpg
- manufacturer : automobile manufacturer
- model : model of car
- trans : type of transmission
- year : model year

[ ]:
```
import csv

%precision 2

with open('mpg.csv') as csvfile:
    mpg = list(csv.DictReader(csvfile))
```

```
mpg[:3] # The first three dictionaries in our list.
```

csv.Dictreader has read in each row of our csv file as a dictionary. len shows that our list is comprised of 234 dictionaries.

[ ]:
```
len(mpg)
```

keys gives us the column names of our csv.

[ ]:
```
mpg[0].keys()
```

This is how to find the average cty fuel economy across all cars. All values in the dictionaries are strings, so we need to convert to float.

[ ]:
```
sum(float(d['cty']) for d in mpg) / len(mpg)
```

Similarly this is how to find the average hwy fuel economy across all cars.

[ ]:
```
sum(float(d['hwy']) for d in mpg) / len(mpg)
```

Use set to return the unique values for the number of cylinders the cars in our dataset have.

[ ]:
```
cylinders = set(d['cyl'] for d in mpg)
cylinders
```

Here's a more complex example where we are grouping the cars by number of cylinder, and finding the average cty mpg for each group.

[ ]:
```
CtyMpgByCyl = []

for c in cylinders: # iterate over all the cylinder levels
    summpg = 0
    cyltypecount = 0
    for d in mpg: # iterate over all dictionaries
        if d['cyl'] == c: # if the cylinder level type matches,
            summpg += float(d['cty']) # add the cty mpg
            cyltypecount += 1 # increment the count
    CtyMpgByCyl.append((c, summpg / cyltypecount)) # append the tuple␣
 ↪('cylinder', 'avg mpg')

CtyMpgByCyl.sort(key=lambda x: x[0])
CtyMpgByCyl
```

Use set to return the unique values for the class types in our dataset.

[ ]:
```
vehicleclass = set(d['class'] for d in mpg) # what are the class types
vehicleclass
```

And here's an example of how to find the average hwy mpg for each class of vehicle in our dataset.

[ ]:
```
HwyMpgByClass = []

for t in vehicleclass: # iterate over all the vehicle classes
    summpg = 0
    vclasscount = 0
    for d in mpg: # iterate over all dictionaries
```

```
        if d['class'] == t: # if the cylinder amount type matches,
            summpg += float(d['hwy']) # add the hwy mpg
            vclasscount += 1 # increment the count
    HwyMpgByClass.append((t, summpg / vclasscount)) # append the tuple␣
 ↪('class', 'avg mpg')

HwyMpgByClass.sort(key=lambda x: x[1])
HwyMpgByClass
```

5 The Python Programming Language: Dates and Times

[ ]:
```
import datetime as dt
import time as tm
```

`time` returns the current time in seconds since the Epoch. (January 1st, 1970)

[ ]:
```
tm.time()
```

Convert the timestamp to datetime.

[ ]:
```
dtnow = dt.datetime.fromtimestamp(tm.time())
dtnow
```

Handy datetime attributes:

[ ]:
```
dtnow.year, dtnow.month, dtnow.day, dtnow.hour, dtnow.minute, dtnow.second #␣
 ↪get year, month, day, etc.from a datetime
```

`timedelta` is a duration expressing the difference between two dates.

[ ]:
```
delta = dt.timedelta(days = 100) # create a timedelta of 100 days
delta
```

`date.today` returns the current local date.

[ ]:
```
today = dt.date.today()
```

[ ]:
```
today - delta # the date 100 days ago
```

[ ]:
```
today > today-delta # compare dates
```

6 The Python Programming Language: Objects and map()

An example of a class in python:

[ ]:
```
class Person:
    department = 'School of Information' #a class variable

    def set_name(self, new_name): #a method
        self.name = new_name
    def set_location(self, new_location):
        self.location = new_location
```

[ ]:
```
person = Person()
person.set_name('Christopher Brooks')
person.set_location('Ann Arbor, MI, USA')
print('{} live in {} and works in the department {}'.format(person.name, person.
↪location, person.department))
```

Here's an example of mapping the `min` function between two lists.

[ ]:
```
store1 = [10.00, 11.00, 12.34, 2.34]
store2 = [9.00, 11.10, 12.34, 2.01]
cheapest = map(min, store1, store2)
cheapest
```

Now let's iterate through the map object to see the values.

[ ]:
```
for item in cheapest:
    print(item)
```

7 The Python Programming Language: Lambda and List Comprehensions

Here's an example of lambda that takes in three parameters and adds the first two.

[ ]:
```
my_function = lambda a, b, c : a + b
```

[ ]:
```
my_function(1, 2, 3)
```

Let's iterate from 0 to 999 and return the even numbers.

[ ]:
```
my_list = []
for number in range(0, 1000):
    if number % 2 == 0:
        my_list.append(number)
my_list
```

Now the same thing but with list comprehension.

[ ]:
```
my_list = [number for number in range(0,1000) if number % 2 == 0]
my_list
```

Numpy_ed

November 15, 2021

Numpy is the fundamental package for numeric computing with Python. It provides powerful ways to create, store, and/or manipulate data, which makes it able to seamlessly and speedily integrate with a wide variety of databases. This is also the foundation that Pandas is built on, which is a high-performance data-centric package that we will learn later in the course.

In this lecture, we will talk about creating array with certain data types, manipulating array, selecting elements from arrays, and loading dataset into array. Such functions are useful for manipulating data and understanding the functionalities of other common Python data packages.

[3]:
```
# You'll recall that we import a library using the `import` keyword as numpy's␣
↪common abbreviation is np
import numpy as np
import math
```

1 Array Creation

[4]:
```
# Arrays are displayed as a list or list of lists and can be created through␣
↪list as well. When creating an
# array, we pass in a list as an argument in numpy array
a = np.array([1, 2, 3])
print(a)
# We can print the number of dimensions of a list using the ndim attribute
print(a.ndim)
```

```
[1 2 3]
1
```

[5]:
```
# If we pass in a list of lists in numpy array, we create a multi-dimensional␣
↪array, for instance, a matrix
b = np.array([[1,2,3],[4,5,6]])
b
```

[5]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

[6]:
```
# We can print out the length of each dimension by calling the shape attribute,␣
↪which returns a tuple
b.shape
```

[6]:
```
(2, 3)
```

[7]:
```
# We can also check the type of items in the array
a.dtype
```

[7]:
```
dtype('int64')
```

[8]:
```
# Besides integers, floats are also accepted in numpy arrays
c = np.array([2.2, 5, 1.1])
c.dtype.name
```

[8]:
```
'float64'
```

[9]:
```
# Let's look at the data in our array
c
```

[9]:
```
array([2.2, 5. , 1.1])
```

[10]:
```
# Note that numpy automatically converts integers, like 5, up to floats, since␣
 ↪there is no loss of prescision.
# Numpy will try and give you the best data type format possible to keep your␣
 ↪data types homogeneous, which
# means all the same, in the array
```

[11]:
```
# Sometimes we know the shape of an array that we want to create, but not what␣
 ↪we want to be in it. numpy
# offers several functions to create arrays with initial placeholders, such as␣
 ↪zero's or one's.
# Lets create two arrays, both the same shape but with different filler values
d = np.zeros((2,3))
print(d)

e = np.ones((2,3))
print(e)
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
[[1. 1. 1.]
 [1. 1. 1.]]
```

[12]:
```
# We can also generate an array with random numbers
np.random.rand(2,3)
```

[12]:
```
array([[0.81967966, 0.24861747, 0.1324781 ],
       [0.10917913, 0.61025551, 0.87901984]])
```

[13]:
```
# You'll see zeros, ones, and rand used quite often to create example arrays,␣
 ↪especially in stack overflow
# posts and other forums.
```

[14]:
```
# We can also create a sequence of numbers in an array with the arrange()␣
 ↪function. The fist argument is the
```

```
# starting bound and the second argument is the ending bound, and the third␣
 ↪argument is the difference between
# each consecutive numbers

# Let's create an array of every even number from ten (inclusive) to fifty␣
 ↪(exclusive)
f = np.arange(10, 50, 2)
f
```

[14]:
```
array([10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42,
       44, 46, 48])
```

[15]:
```
# if we want to generate a sequence of floats, we can use the linspace()␣
 ↪function. In this function the third
# argument isn't the difference between two numbers, but the total number of␣
 ↪items you want to generate
np.linspace( 0, 2, 15 ) # 15 numbers from 0 (inclusive) to 2 (inclusive)
```

[15]:
```
array([0.        , 0.14285714, 0.28571429, 0.42857143, 0.57142857,
       0.71428571, 0.85714286, 1.        , 1.14285714, 1.28571429,
       1.42857143, 1.57142857, 1.71428571, 1.85714286, 2.        ])
```

2 Array Operations

[16]:
```
# We can do many things on arrays, such as mathematical manipulation (addition,␣
 ↪subtraction, square,
# exponents) as well as use boolean arrays, which are binary values. We can␣
 ↪also do matrix manipulation such
# as product, transpose, inverse, and so forth.
```

[17]:
```
# Arithmetic operators on array apply elementwise.

# Let's create a couple of arrays
a = np.array([10,20,30,40])
b = np.array([1, 2, 3,4])

# Now let's look at a minus b
c = a-b
print(c)

# And let's look at a times b
d = a*b
print(d)
```

```
[ 9 18 27 36]
[ 10  40  90 160]
```

[18]:
```
# With arithmetic manipulation, we can convert current data to the way we want␣
 ↪it to be. Here's a real-world
# problem I face - I moved down to the United States about 6 years ago from␣
 ↪Canada. In Canada we use celcius
# for temperatures, and my wife still hasn't converted to the US system which␣
 ↪uses farenheit. With numpy I
# could easily convert a number of farenheit values, say the weather forecase,␣
 ↪to ceclius

# Let's create an array of typical Ann Arbor winter farenheit values
farenheit = np.array([0,-10,-5,-15,0])

# And the formula for conversion is ((řF  32) Œ 5/9 = řC)
celcius = (farenheit - 31) * (5/9)
celcius
```

[18]:
```
array([-17.22222222, -22.77777778, -20.        , -25.55555556,
       -17.22222222])
```

[19]:
```
# Great, so now she knows it's a little chilly outside but not so bad.
```

[20]:
```
# Another useful and important manipulation is the boolean array. We can apply␣
 ↪an operator on an array, and a
# boolean array will be returned for any element in the original, with True␣
 ↪being emitted if it meets the condition and False oetherwise.
# For instance, if we want to get a boolean array to check celcius degrees that␣
 ↪are greater than -20 degrees
celcius > -20
```

[20]:
```
array([ True, False, False, False,  True])
```

[21]:
```
# Here's another example, we could use the modulus operator to check numbers in␣
 ↪an array to see if they are even. Recall that modulus does division but␣
 ↪throws away everything but the remainder (decimal) portion)
celcius%2 == 0
```

[21]:
```
array([False, False,  True, False, False])
```

[22]:
```
# Besides elementwise manipulation, it is important to know that numpy supports␣
 ↪matrix manipulation. Let's
# look at matrix product. if we want to do elementwise product, we use the "*"␣
 ↪sign
A = np.array([[1,1],[0,1]])
B = np.array([[2,0],[3,4]])
print(A*B)

# if we want to do matrix product, we use the "@" sign or use the dot function
print(A@B)
```

```
[[2 0]
```

```
 [0 4]]
[[5 4]
 [3 4]]
```

[23]:
```
# You don't have to worry about complex matrix operations for this course, but␣
↪it's important to know that
# numpy is the underpinning of scientific computing libraries in python, and␣
↪that it is capable of doing both
# element-wise operations (the asterix) as well as matrix-level operations (the␣
↪@ sign). There's more on this
# in a subsequent course.
```

[24]:
```
# A few more linear algebra concepts are worth layering in here. You might␣
↪recall that the product of two
# matrices is only plausible when the inner dimensions of the two matrices are␣
↪the same. The dimensions refer
# to the number of elements both horizontally and vertically in the rendered␣
↪matricies you've seen here. We
# can use numpy to quickly see the shape of a matrix:
A.shape
```

[24]:
```
(2, 2)
```

[25]:
```
# When manipulating arrays of different types, the type of the resulting array␣
↪will correspond to
# the more general of the two types. This is called upcasting.

# Let's create an array of integers
array1 = np.array([[1, 2, 3], [4, 5, 6]])
print(array1.dtype)

# Now let's create an array of floats
array2 = np.array([[7.1, 8.2, 9.1], [10.4, 11.2, 12.3]])
print(array2.dtype)
```

```
int64
float64
```

[26]:
```
# Integers (int) are whole numbers only, and Floating point numbers (float) can␣
↪have a whole number portion
# and a decimal portion. The 64 in this example refers to the number of bits␣
↪that the operating system is
# reserving to represent the number, which determines the size (or precision)␣
↪of the numbers that can be
# represented.
```

[27]:
```
# Let's do an addition for the two arrays
array3=array1+array2
```

```
print(array3)
print(array3.dtype)
```

```
[[ 8.1 10.2 12.1]
 [14.4 16.2 18.3]]
float64
```

[28]:
```
# Notice how the items in the resulting array have been upcast into floating␣
 ↪point numbers
```

[29]:
```
# Numpy arrays have many interesting aggregation functions on them, such as ␣
 ↪sum(), max(), min(), and mean()
print(array3.sum())
print(array3.max())
print(array3.min())
print(array3.mean())
```

```
79.3
18.3
8.1
13.216666666666667
```

[30]:
```
# For two dimensional arrays, we can do the same thing for each row or column
# let's create an array with 15 elements, ranging from 1 to 15,
# with a dimension of 3X5
b = np.arange(1,16,1).reshape(3,5)
print(b)
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]]
```

[31]:
```
# Now, we often think about two dimensional arrays being made up of rows and␣
 ↪columns, but you can also think
# of these arrays as just a giant ordered list of numbers, and the *shape* of␣
 ↪the array, the number of rows
# and columns, is just an abstraction that we have for a particular purpose.␣
 ↪Actually, this is exactly how
# basic images are stored in computer environments.

# Let's take a look at an example and see how numpy comes into play.
```

[32]:
```
# For this demonstration I'll use the python imaging library (PIL) and a␣
 ↪function to display images in the
# Jupyter notebook
from PIL import Image
from IPython.display import display
```

```
# And let's just look at the image I'm talking about
im = Image.open('chris.tiff')
display(im)
```

```
    ␣
↪---------------------------------------------------------------------------

       FileNotFoundError                         Traceback (most recent call␣
↪last)

       <ipython-input-32-12896d20ac91> in <module>
         5
         6 # And let's just look at the image I'm talking about
   ----> 7 im = Image.open('chris.tiff')
         8 display(im)


       /opt/conda/lib/python3.7/site-packages/PIL/Image.py in open(fp, mode)
      2768
      2769     if filename:
   -> 2770         fp = builtins.open(filename, "rb")
      2771         exclusive_fp = True
      2772


       FileNotFoundError: [Errno 2] No such file or directory: 'chris.tiff'
```

[ ]:
```
# Now, we can conver this PIL image to a numpy array
array=np.array(im)
print(array.shape)
array
```

[ ]:
```
# Here we see that we have a 200x200 array and that the values are all uint8.␣
 ↪The uint means that they are
# unsigned integers (so no negative numbers) and the 8 means 8 bits per byte.␣
 ↪This means that each value can
# be up to 2*2*2*2*2*2*2*2=256 in size (well, actually 255, because we start at␣
 ↪zero). For black and white
# images black is stored as 0 and white is stored as 255. So if we just wanted␣
 ↪to invert this image we could
# use the numpy array to do so

# Let's create an array the same shape
mask=np.full(array.shape,255)
```

```
mask
```

[ ]:
```
# Now let's subtract that from the modified array
modified_array=array-mask

# And lets convert all of the negative values to positive values
modified_array=modified_array*-1

# And as a last step, let's tell numpy to set the value of the datatype␣
 ↪correctly
modified_array=modified_array.astype(np.uint8)
modified_array
```

[ ]:
```
# And lastly, lets display this new array. We do this by using the fromarray()␣
 ↪function in the python
# imaging library to convert the numpy array into an object jupyter can render
display(Image.fromarray(modified_array))
```

[ ]:
```
# Cool. Ok, remember how I started this by talking about how we could just␣
 ↪think of this as a giant array
# of bytes, and that the shape was an abstraction? Well, we could just decide␣
 ↪to reshape the array and still
# try and render it. PIL is interpreting the individual rows as lines, so we␣
 ↪can change the number of lines
# and columns if we want to. What do you think that would look like?
reshaped=np.reshape(modified_array,(100,400))
print(reshaped.shape)
display(Image.fromarray(reshaped))
```

[ ]:
```
# Can't say I find that particularly flattering. By reshaping the array to be␣
 ↪only 100 rows high but 400
# columns we've essentially doubled the image by taking every other line and␣
 ↪stacking them out in width. This
# makes the image look more stretched out too.

# This isn't an image manipulation course, but the point was to show you that␣
 ↪these numpy arrays are really
# just abstractions on top of data, and that data has an underlying format (in␣
 ↪this case, uint8). But further,
# we can build abstractions on top of that, such as computer code which renders␣
 ↪a byte as either black or
# white, which has meaning to people. In some ways, this whole degree is about␣
 ↪data and the abstractions that
# we can build on top of that data, from individual byte representations␣
 ↪through to complex neural networks of
# functions or interactive visualizations. Your role as a data scientist is to␣
 ↪understand what the data means
```

```
# (it's context an collection), and transform it into a different␣
↪representation to be used for sensemaking.
```

[ ]:
```
# Ok, back to the mechanics of numpy.
```

3 Indexing, Slicing and Iterating

[ ]:
```
# Indexing, slicing and iterating are extremely important for data manipulation␣
↪and analysis because these
# techinques allow us to select data based on conditions, and copy or update␣
↪data.
```

3.1 Indexing

[ ]:
```
# First we are going to look at integer indexing. A one-dimensional array,␣
↪works in similar ways as a list -
# To get an element in a one-dimensional array, we simply use the offset index.
a = np.array([1,3,5,7])
a[2]
```

[33]:
```
# For multidimensional array, we need to use integer array indexing, let's␣
↪create a new multidimensional array
a = np.array([[1,2], [3, 4], [5, 6]])
a
```

[33]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

[34]:
```
# if we want to select one certain element, we can do so by entering the index,␣
↪which is comprised of two
# integers the first being the row, and the second the column
a[1,1] # remember in python we start at 0!
```

[34]:
```
4
```

[35]:
```
# if we want to get multiple elements
# for example, 1, 4, and 6 and put them into a one-dimensional array
# we can enter the indices directly into an array function
np.array([a[0, 0], a[1, 1], a[2, 1]])
```

[35]:
```
array([1, 4, 6])
```

[36]:
```
# we can also do that by using another form of array indexing, which essentiall␣
↪"zips" the first list and the
# second list up
print(a[[0, 1, 2], [0, 1, 1]])
```

```
[1 4 6]
```

3.2 Boolean Indexing

[37]:
```
# Boolean indexing allows us to select arbitrary elements based on conditions.␣
  ↪For example, in the matrix we
# just talked about we want to find elements that are greater than 5 so we set␣
  ↪up a conditon a >5
print(a >5)
# This returns a boolean array showing that if the value at the corresponding␣
  ↪index is greater than 5
```

```
[[False False]
 [False False]
 [False  True]]
```

[38]:
```
# We can then place this array of booleans like a mask over the original array␣
  ↪to return a one-dimensional
# array relating to the true values.
print(a[a>5])
```

```
[6]
```

[39]:
```
# As we will see, this functionality is essential in the pandas toolkit which␣
  ↪is the bulk of this course
```

3.3 Slicing

[40]:
```
# Slicing is a way to create a sub-array based on the original array. For␣
  ↪one-dimensional arrays, slicing
# works in similar ways to a list. To slice, we use the : sign. For instance,␣
  ↪if we put :3 in the indexing
# brackets, we get elements from index 0 to index 3 (excluding index 3)
a = np.array([0,1,2,3,4,5])
print(a[:3])
```

```
[0 1 2]
```

[41]:
```
# By putting 2:4 in the bracket, we get elements from index 2 to index 4␣
  ↪(excluding index 4)
print(a[2:4])
```

```
[2 3]
```

[42]:
```
# For multi-dimensional arrays, it works similarly, lets see an example
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
a
```

[42]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

[43]:
```
# First, if we put one argument in the array, for example a[:2] then we would␣
 ↪get all the elements from the
# first (0th) and second row (1th)
a[:2]
```

[43]:
```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

[44]:
```
# If we add another argument to the array, for example a[:2, 1:3], we get the␣
 ↪first two rows but then the
# second and third column values only
a[:2, 1:3]
```

[44]:
```
array([[2, 3],
       [6, 7]])
```

[45]:
```
# So, in multidimensional arrays, the first argument is for selecting rows, and␣
 ↪the second argument is for
# selecting columns
```

[46]:
```
# It is important to realize that a slice of an array is a view into the same␣
 ↪data. This is called passing by
# reference. So modifying the sub array will consequently modify the original␣
 ↪array

# Here I'll change the element at position [0, 0], which is 2, to 50, then we␣
 ↪can see that the value in the
# original array is changed to 50 as well

sub_array = a[:2, 1:3]
print("sub array index [0,0] value before change:", sub_array[0,0])
sub_array[0,0] = 50
print("sub array index [0,0] value after change:", sub_array[0,0])
print("original array index [0,1] value after change:", a[0,1])
```

```
sub array index [0,0] value before change: 2
sub array index [0,0] value after change: 50
original array index [0,1] value after change: 50
```

4 Trying Numpy with Datasets

[47]:
```
# Now that we have learned the essentials of Numpy let's use it on a couple of␣
 ↪datasets
```

[48]:
```
# Here we have a very popular dataset on wine quality, and we are going to only␣
 ↪look at red wines. The data
# fields include: fixed acidity, volatile aciditycitric acid, residual sugar,␣
 ↪chlorides, free sulfur dioxide,
# total sulfur dioxidedensity, pH, sulphates, alcohol, quality
```

[49]:
```
# To load a dataset in Numpy, we can use the genfromtxt() function. We can␣
 ↪specify data file name, delimiter
# (which is optional but often used), and number of rows to skip if we have a␣
 ↪header row, hence it is 1 here

# The genfromtxt() function has a parameter called dtype for specifying data␣
 ↪types of each column this
# parameter is optional. Without specifying the types, all types will be casted␣
 ↪the same to the more
# general/precise type

wines = np.genfromtxt("datasets/winequality-red.csv", delimiter=";",␣
 ↪skip_header=1)
wines
```

[49]:
```
array([[ 7.4  ,  0.7  ,  0.   , ...,  0.56 ,  9.4  ,  5.   ],
       [ 7.8  ,  0.88 ,  0.   , ...,  0.68 ,  9.8  ,  5.   ],
       [ 7.8  ,  0.76 ,  0.04 , ...,  0.65 ,  9.8  ,  5.   ],
       ...,
       [ 6.3  ,  0.51 ,  0.13 , ...,  0.75 , 11.   ,  6.   ],
       [ 5.9  ,  0.645,  0.12 , ...,  0.71 , 10.2  ,  5.   ],
       [ 6.   ,  0.31 ,  0.47 , ...,  0.66 , 11.   ,  6.   ]])
```

[50]:
```
# Recall that we can use integer indexing to get a certain column or a row. For␣
 ↪example, if we want to select
# the fixed acidity column, which is the first coluumn, we can do so by␣
 ↪entering the index into the array.
# Also remember that for multidimensional arrays, the first argument refers to␣
 ↪the row, and the second
# argument refers to the column, and if we just give one argument then we'll␣
 ↪get a single dimensional list
# back.

# So all rows combined but only the first column from them would be
print("one integer 0 for slicing: ", wines[:, 0])
# But if we wanted the same values but wanted to preserve that they sit in␣
 ↪their own rows we would write
print("0 to 1 for slicing: \n", wines[:, 0:1])
```

```
one integer 0 for slicing:  [7.4 7.8 7.8 ... 6.3 5.9 6. ]
0 to 1 for slicing:
 [[7.4]
 [7.8]
 [7.8]
 ...
 [6.3]
 [5.9]
 [6. ]]
```

[51]:
```
# This is another great example of how the shape of the data is an abstraction␣
↪which we can layer
# intentionally on top of the data we are working with.
```

[52]:
```
# If we want a range of columns in order, say columns 0 through 3 (recall, this␣
↪means first, second, and
# third, since we start at zero and don't include the training index value), we␣
↪can do that too
wines[:, 0:3]
```

[52]:
```
array([[7.4  , 0.7  , 0.   ],
       [7.8  , 0.88 , 0.   ],
       [7.8  , 0.76 , 0.04 ],
       ...,
       [6.3  , 0.51 , 0.13 ],
       [5.9  , 0.645, 0.12 ],
       [6.   , 0.31 , 0.47 ]])
```

[53]:
```
# What if we want several non-consecutive columns? We can place the indices of␣
↪the columns that we want into
# an array and pass the array as the second argument. Here's an example
wines[:, [0,2,4]]
```

[53]:
```
array([[7.4  , 0.   , 0.076],
       [7.8  , 0.   , 0.098],
       [7.8  , 0.04 , 0.092],
       ...,
       [6.3  , 0.13 , 0.076],
       [5.9  , 0.12 , 0.075],
       [6.   , 0.47 , 0.067]])
```

[54]:
```
# We can also do some basic summarization of this dataset. For example, if we␣
↪want to find out the average
# quality of red wine, we can select the quality column. We could do this in a␣
↪couple of ways, but the most
# appropriate is to use the -1 value for the index, as negative numbers mean␣
↪slicing from the back of the
# list. We can then call the aggregation functions on this data.
wines[:,-1].mean()
```

[54]:
```
5.6360225140712945
```

[55]:
```
# Let's take a look at another dataset, this time on graduate school admissions.
 ↪ It has fields such as GRE
# score, TOEFL score, university rating, GPA, having research experience or␣
 ↪not, and a chance of admission.
# With this dataset, we can do data manipulation and basic analysis to infer␣
 ↪what conditions are associated
# with higher chance of admission. Let's take a look.
```

[56]:
```
# We can specify data field names when using genfromtxt() to loads CSV data.␣
 ↪Also, we can have numpy try and
# infer the type of a column by setting the dtype parameter to None
graduate_admission = np.genfromtxt('datasets/Admission_Predict.csv',␣
 ↪dtype=None, delimiter=',', skip_header=1,
                                   names=('Serial No','GRE Score', 'TOEFL␣
 ↪Score', 'University Rating', 'SOP',
                                          'LOR','CGPA','Research', 'Chance of␣
 ↪Admit'))
graduate_admission
```

[56]:
```
array([(  1, 337, 118, 4, 4.5, 4.5, 9.65, 1, 0.92),
       (  2, 324, 107, 4, 4. , 4.5, 8.87, 1, 0.76),
       (  3, 316, 104, 3, 3. , 3.5, 8.  , 1, 0.72),
       (  4, 322, 110, 3, 3.5, 2.5, 8.67, 1, 0.8 ),
       (  5, 314, 103, 2, 2. , 3. , 8.21, 0, 0.65),
       (  6, 330, 115, 5, 4.5, 3. , 9.34, 1, 0.9 ),
       (  7, 321, 109, 3, 3. , 4. , 8.2 , 1, 0.75),
       (  8, 308, 101, 2, 3. , 4. , 7.9 , 0, 0.68),
       (  9, 302, 102, 1, 2. , 1.5, 8.  , 0, 0.5 ),
       ( 10, 323, 108, 3, 3.5, 3. , 8.6 , 0, 0.45),
       ( 11, 325, 106, 3, 3.5, 4. , 8.4 , 1, 0.52),
       ( 12, 327, 111, 4, 4. , 4.5, 9.  , 1, 0.84),
       ( 13, 328, 112, 4, 4. , 4.5, 9.1 , 1, 0.78),
       ( 14, 307, 109, 3, 4. , 3. , 8.  , 1, 0.62),
       ( 15, 311, 104, 3, 3.5, 2. , 8.2 , 1, 0.61),
       ( 16, 314, 105, 3, 3.5, 2.5, 8.3 , 0, 0.54),
       ( 17, 317, 107, 3, 4. , 3. , 8.7 , 0, 0.66),
       ( 18, 319, 106, 3, 4. , 3. , 8.  , 1, 0.65),
       ( 19, 318, 110, 3, 4. , 3. , 8.8 , 0, 0.63),
       ( 20, 303, 102, 3, 3.5, 3. , 8.5 , 0, 0.62),
       ( 21, 312, 107, 3, 3. , 2. , 7.9 , 1, 0.64),
       ( 22, 325, 114, 4, 3. , 2. , 8.4 , 0, 0.7 ),
       ( 23, 328, 116, 5, 5. , 5. , 9.5 , 1, 0.94),
       ( 24, 334, 119, 5, 5. , 4.5, 9.7 , 1, 0.95),
       ( 25, 336, 119, 5, 4. , 3.5, 9.8 , 1, 0.97),
       ( 26, 340, 120, 5, 4.5, 4.5, 9.6 , 1, 0.94),
       ( 27, 322, 109, 5, 4.5, 3.5, 8.8 , 0, 0.76),
       ( 28, 298,  98, 2, 1.5, 2.5, 7.5 , 1, 0.44),
       ( 29, 295,  93, 1, 2. , 2. , 7.2 , 0, 0.46),
       ( 30, 310,  99, 2, 1.5, 2. , 7.3 , 0, 0.54),
       ( 31, 300,  97, 2, 3. , 3. , 8.1 , 1, 0.65),
       ( 32, 327, 103, 3, 4. , 4. , 8.3 , 1, 0.74),
       ( 33, 338, 118, 4, 3. , 4.5, 9.4 , 1, 0.91),
       ( 34, 340, 114, 5, 4. , 4. , 9.6 , 1, 0.9 ),
       ( 35, 331, 112, 5, 4. , 5. , 9.8 , 1, 0.94),
       ( 36, 320, 110, 5, 5. , 5. , 9.2 , 1, 0.88),
       ( 37, 299, 106, 2, 4. , 4. , 8.4 , 0, 0.64),
       ( 38, 300, 105, 1, 1. , 2. , 7.8 , 0, 0.58),
       ( 39, 304, 105, 1, 3. , 1.5, 7.5 , 0, 0.52),
       ( 40, 307, 108, 2, 4. , 3.5, 7.7 , 0, 0.48),
       ( 41, 308, 110, 3, 3.5, 3. , 8.  , 1, 0.46),
       ( 42, 316, 105, 2, 2.5, 2.5, 8.2 , 1, 0.49),
       ( 43, 313, 107, 2, 2.5, 2. , 8.5 , 1, 0.53),
       ( 44, 332, 117, 4, 4.5, 4. , 9.1 , 0, 0.87),
       ( 45, 326, 113, 5, 4.5, 4. , 9.4 , 1, 0.91),
       ( 46, 322, 110, 5, 5. , 4. , 9.1 , 1, 0.88),
       ( 47, 329, 114, 5, 4. , 5. , 9.3 , 1, 0.86),
       ( 48, 339, 119, 5, 4.5, 4. , 9.7 , 0, 0.89),
       ( 49, 321, 110, 3, 3.5, 5. , 8.85, 1, 0.82),
       ( 50, 327, 111, 4, 3. , 4. , 8.4 , 1, 0.78),
       ( 51, 313,  98, 3, 2.5, 4.5, 8.3 , 1, 0.76),
       ( 52, 312, 100, 2, 1.5, 3.5, 7.9 , 1, 0.56),
       ( 53, 334, 116, 4, 4. , 3. , 8.  , 1, 0.78),
       ( 54, 324, 112, 4, 4. , 2.5, 8.1 , 1, 0.72),
       ( 55, 322, 110, 3, 3. , 3.5, 8.  , 0, 0.7 ),
       ( 56, 320, 103, 3, 3. , 3. , 7.7 , 0, 0.64),
       ( 57, 316, 102, 3, 2. , 3. , 7.4 , 0, 0.64),
       ( 58, 298,  99, 2, 4. , 2. , 7.6 , 0, 0.46),
       ( 59, 300,  99, 1, 3. , 2. , 6.8 , 1, 0.36),
       ( 60, 311, 104, 2, 2. , 2. , 8.3 , 0, 0.42),
       ( 61, 309, 100, 2, 3. , 3. , 8.1 , 0, 0.48),
       ( 62, 307, 101, 3, 4. , 3. , 8.2 , 0, 0.47),
       ( 63, 304, 105, 2, 3. , 3. , 8.2 , 1, 0.54),
       ( 64, 315, 107, 2, 4. , 3. , 8.5 , 1, 0.56),
       ( 65, 325, 111, 3, 3. , 3.5, 8.7 , 0, 0.52),
       ( 66, 325, 112, 4, 3.5, 3.5, 8.92, 0, 0.55),
       ( 67, 327, 114, 3, 3. , 3. , 9.02, 0, 0.61),
       ( 68, 316, 107, 2, 3.5, 3.5, 8.64, 1, 0.57),
       ( 69, 318, 109, 3, 3.5, 4. , 9.22, 1, 0.68),
       ( 70, 328, 115, 4, 4.5, 4. , 9.16, 1, 0.78),
       ( 71, 332, 118, 5, 5. , 5. , 9.64, 1, 0.94),
       ( 72, 336, 112, 5, 5. , 5. , 9.76, 1, 0.96),
       ( 73, 321, 111, 5, 5. , 5. , 9.45, 1, 0.93),
       ( 74, 314, 108, 4, 4.5, 4. , 9.04, 1, 0.84),
       ( 75, 314, 106, 3, 3. , 5. , 8.9 , 0, 0.74),
       ( 76, 329, 114, 2, 2. , 4. , 8.56, 1, 0.72),
       ( 77, 327, 112, 3, 3. , 3. , 8.72, 1, 0.74),
       ( 78, 301,  99, 2, 3. , 2. , 8.22, 0, 0.64),
       ( 79, 296,  95, 2, 3. , 2. , 7.54, 1, 0.44),
       ( 80, 294,  93, 1, 1.5, 2. , 7.36, 0, 0.46),
       ( 81, 312, 105, 3, 2. , 3. , 8.02, 1, 0.5 ),
       ( 82, 340, 120, 4, 5. , 5. , 9.5 , 1, 0.96),
       ( 83, 320, 110, 5, 5. , 4.5, 9.22, 1, 0.92),
       ( 84, 322, 115, 5, 4. , 4.5, 9.36, 1, 0.92),
       ( 85, 340, 115, 5, 4.5, 4.5, 9.45, 1, 0.94),
       ( 86, 319, 103, 4, 4.5, 3.5, 8.66, 0, 0.76),
       ( 87, 315, 106, 3, 4.5, 3.5, 8.42, 0, 0.72),
       ( 88, 317, 107, 2, 3.5, 3. , 8.28, 0, 0.66),
       ( 89, 314, 108, 3, 4.5, 3.5, 8.14, 0, 0.64),
       ( 90, 316, 109, 4, 4.5, 3.5, 8.76, 1, 0.74),
       ( 91, 318, 106, 2, 4. , 4. , 7.92, 1, 0.64),
       ( 92, 299,  97, 3, 5. , 3.5, 7.66, 0, 0.38),
       ( 93, 298,  98, 2, 4. , 3. , 8.03, 0, 0.34),
       ( 94, 301,  97, 2, 3. , 3. , 7.88, 1, 0.44),
       ( 95, 303,  99, 3, 2. , 2.5, 7.66, 0, 0.36),
       ( 96, 304, 100, 4, 1.5, 2.5, 7.84, 0, 0.42),
       ( 97, 306, 100, 2, 3. , 3. , 8.  , 0, 0.48),
       ( 98, 331, 120, 3, 4. , 4. , 8.96, 1, 0.86),
       ( 99, 332, 119, 4, 5. , 4.5, 9.24, 1, 0.9 ),
       (100, 323, 113, 3, 4. , 4. , 8.88, 1, 0.79),
       (101, 322, 107, 3, 3.5, 3.5, 8.46, 1, 0.71),
       (102, 312, 105, 2, 2.5, 3. , 8.12, 0, 0.64),
       (103, 314, 106, 2, 4. , 3.5, 8.25, 0, 0.62),
       (104, 317, 104, 2, 4.5, 4. , 8.47, 0, 0.57),
       (105, 326, 112, 3, 3.5, 3. , 9.05, 1, 0.74),
       (106, 316, 110, 3, 4. , 4.5, 8.78, 1, 0.69),
       (107, 329, 111, 4, 4.5, 4.5, 9.18, 1, 0.87),
       (108, 338, 117, 4, 3.5, 4.5, 9.46, 1, 0.91),
       (109, 331, 116, 5, 5. , 5. , 9.38, 1, 0.93),
       (110, 304, 103, 5, 5. , 4. , 8.64, 0, 0.68),
       (111, 305, 108, 5, 3. , 3. , 8.48, 0, 0.61),
       (112, 321, 109, 4, 4. , 4. , 8.68, 1, 0.69),
       (113, 301, 107, 3, 3.5, 3.5, 8.34, 1, 0.62),
       (114, 320, 110, 2, 4. , 3.5, 8.56, 0, 0.72),
       (115, 311, 105, 3, 3.5, 3. , 8.45, 1, 0.59),
       (116, 310, 106, 4, 4.5, 4.5, 9.04, 1, 0.66),
       (117, 299, 102, 3, 4. , 3.5, 8.62, 0, 0.56),
       (118, 290, 104, 4, 2. , 2.5, 7.46, 0, 0.45),
       (119, 296,  99, 2, 3. , 3.5, 7.28, 0, 0.47),
       (120, 327, 104, 5, 3. , 3.5, 8.84, 1, 0.71),
       (121, 335, 117, 5, 5. , 5. , 9.56, 1, 0.94),
       (122, 334, 119, 5, 4.5, 4.5, 9.48, 1, 0.94),
       (123, 310, 106, 4, 1.5, 2.5, 8.36, 0, 0.57),
       (124, 308, 108, 3, 3.5, 3.5, 8.22, 0, 0.61),
       (125, 301, 106, 4, 2.5, 3. , 8.47, 0, 0.57),
       (126, 300, 100, 3, 2. , 3. , 8.66, 1, 0.64),
       (127, 323, 113, 3, 4. , 3. , 9.32, 1, 0.85),
       (128, 319, 112, 3, 2.5, 2. , 8.71, 1, 0.78),
       (129, 326, 112, 3, 3.5, 3. , 9.1 , 1, 0.84),
       (130, 333, 118, 5, 5. , 5. , 9.35, 1, 0.92),
       (131, 339, 114, 5, 4. , 4.5, 9.76, 1, 0.96),
       (132, 303, 105, 5, 5. , 4.5, 8.65, 0, 0.77),
       (133, 309, 105, 5, 3.5, 3.5, 8.56, 0, 0.71),
       (134, 323, 112, 5, 4. , 4.5, 8.78, 0, 0.79),
       (135, 333, 113, 5, 4. , 4. , 9.28, 1, 0.89),
       (136, 314, 109, 4, 3.5, 4. , 8.77, 1, 0.82),
       (137, 312, 103, 3, 5. , 4. , 8.45, 0, 0.76),
       (138, 316, 100, 2, 1.5, 3. , 8.16, 1, 0.71),
       (139, 326, 116, 2, 4.5, 3. , 9.08, 1, 0.8 ),
       (140, 318, 109, 1, 3.5, 3.5, 9.12, 0, 0.78),
       (141, 329, 110, 2, 4. , 3. , 9.15, 1, 0.84),
       (142, 332, 118, 2, 4.5, 3.5, 9.36, 1, 0.9 ),
       (143, 331, 115, 5, 4. , 3.5, 9.44, 1, 0.92),
       (144, 340, 120, 4, 4.5, 4. , 9.92, 1, 0.97),
       (145, 325, 112, 2, 3. , 3.5, 8.96, 1, 0.8 ),
       (146, 320, 113, 2, 2. , 2.5, 8.64, 1, 0.81),
       (147, 315, 105, 3, 2. , 2.5, 8.48, 0, 0.75),
       (148, 326, 114, 3, 3. , 3. , 9.11, 1, 0.83),
       (149, 339, 116, 4, 4. , 3.5, 9.8 , 1, 0.96),
       (150, 311, 106, 2, 3.5, 3. , 8.26, 1, 0.79),
       (151, 334, 114, 4, 4. , 4. , 9.43, 1, 0.93),
       (152, 332, 116, 5, 5. , 5. , 9.28, 1, 0.94),
       (153, 321, 112, 5, 5. , 5. , 9.06, 1, 0.86),
       (154, 324, 105, 3, 3. , 4. , 8.75, 0, 0.79),
       (155, 326, 108, 3, 3. , 3.5, 8.89, 0, 0.8 ),
       (156, 312, 109, 3, 3. , 3. , 8.69, 0, 0.77),
       (157, 315, 105, 3, 2. , 2.5, 8.34, 0, 0.7 ),
       (158, 309, 104, 2, 2. , 2.5, 8.26, 0, 0.65),
       (159, 306, 106, 2, 2. , 2.5, 8.14, 0, 0.61),
       (160, 297, 100, 1, 1.5, 2. , 7.9 , 0, 0.52),
       (161, 315, 103, 1, 1.5, 2. , 7.86, 0, 0.57),
       (162, 298,  99, 1, 1.5, 3. , 7.46, 0, 0.53),
       (163, 318, 109, 3, 3. , 3. , 8.5 , 0, 0.67),
       (164, 317, 105, 3, 3.5, 3. , 8.56, 0, 0.68),
       (165, 329, 111, 4, 4.5, 4. , 9.01, 1, 0.81),
       (166, 322, 110, 5, 4.5, 4. , 8.97, 0, 0.78),
       (167, 302, 102, 3, 3.5, 5. , 8.33, 0, 0.65),
       (168, 313, 102, 3, 2. , 3. , 8.27, 0, 0.64),
       (169, 293,  97, 2, 2. , 4. , 7.8 , 1, 0.64),
       (170, 311,  99, 2, 2.5, 3. , 7.98, 0, 0.65),
       (171, 312, 101, 2, 2.5, 3.5, 8.04, 1, 0.68),
       (172, 334, 117, 5, 4. , 4.5, 9.07, 1, 0.89),
       (173, 322, 110, 4, 4. , 5. , 9.13, 1, 0.86),
       (174, 323, 113, 4, 4. , 4.5, 9.23, 1, 0.89),
       (175, 321, 111, 4, 4. , 4. , 8.97, 1, 0.87),
       (176, 320, 111, 4, 4.5, 3.5, 8.87, 1, 0.85),
       (177, 329, 119, 4, 4.5, 4.5, 9.16, 1, 0.9 ),
       (178, 319, 110, 3, 3.5, 3.5, 9.04, 0, 0.82),
       (179, 309, 108, 3, 2.5, 3. , 8.12, 0, 0.72),
       (180, 307, 102, 3, 3. , 3. , 8.27, 0, 0.73),
       (181, 300, 104, 3, 3.5, 3. , 8.16, 0, 0.71),
       (182, 305, 107, 2, 2.5, 2.5, 8.42, 0, 0.71),
       (183, 299, 100, 2, 3. , 3.5, 7.88, 0, 0.68),
       (184, 314, 110, 3, 4. , 4. , 8.8 , 0, 0.75),
       (185, 316, 106, 2, 2.5, 4. , 8.32, 0, 0.72),
       (186, 327, 113, 4, 4.5, 4.5, 9.11, 1, 0.89),
       (187, 317, 107, 3, 3.5, 3. , 8.68, 1, 0.84),
       (188, 335, 118, 5, 4.5, 3.5, 9.44, 1, 0.93),
       (189, 331, 115, 5, 4.5, 3.5, 9.36, 1, 0.93),
       (190, 324, 112, 5, 5. , 5. , 9.08, 1, 0.88),
       (191, 324, 111, 5, 4.5, 4. , 9.16, 1, 0.9 ),
       (192, 323, 110, 5, 4. , 5. , 8.98, 1, 0.87),
       (193, 322, 114, 5, 4.5, 4. , 8.94, 1, 0.86),
       (194, 336, 118, 5, 4.5, 5. , 9.53, 1, 0.94),
       (195, 316, 109, 3, 3.5, 3. , 8.76, 0, 0.77),
       (196, 307, 107, 2, 3. , 3.5, 8.52, 1, 0.78),
       (197, 306, 105, 2, 3. , 2.5, 8.26, 0, 0.73),
       (198, 310, 106, 2, 3.5, 2.5, 8.33, 0, 0.73),
       (199, 311, 104, 3, 4.5, 4.5, 8.43, 0, 0.7 ),
       (200, 313, 107, 3, 4. , 4.5, 8.69, 0, 0.72),
       (201, 317, 103, 3, 2.5, 3. , 8.54, 1, 0.73),
       (202, 315, 110, 2, 3.5, 3. , 8.46, 1, 0.72),
       (203, 340, 120, 5, 4.5, 4.5, 9.91, 1, 0.97),
       (204, 334, 120, 5, 4. , 5. , 9.87, 1, 0.97),
       (205, 298, 105, 3, 3.5, 4. , 8.54, 0, 0.69),
       (206, 295,  99, 2, 2.5, 3. , 7.65, 0, 0.57),
       (207, 315,  99, 2, 3.5, 3. , 7.89, 0, 0.63),
       (208, 310, 102, 3, 3.5, 4. , 8.02, 1, 0.66),
       (209, 305, 106, 2, 3. , 3. , 8.16, 0, 0.64),
       (210, 301, 104, 3, 3.5, 4. , 8.12, 1, 0.68),
       (211, 325, 108, 4, 4.5, 4. , 9.06, 1, 0.79),
       (212, 328, 110, 4, 5. , 4. , 9.14, 1, 0.82),
       (213, 338, 120, 4, 5. , 5. , 9.66, 1, 0.95),
       (214, 333, 119, 5, 5. , 4.5, 9.78, 1, 0.96),
       (215, 331, 117, 4, 4.5, 5. , 9.42, 1, 0.94),
       (216, 330, 116, 5, 5. , 4.5, 9.36, 1, 0.93),
       (217, 322, 112, 4, 4.5, 4.5, 9.26, 1, 0.91),
       (218, 321, 109, 4, 4. , 4. , 9.13, 1, 0.85),
       (219, 324, 110, 4, 3. , 3.5, 8.97, 1, 0.84),
       (220, 312, 104, 3, 3.5, 3.5, 8.42, 0, 0.74),
       (221, 313, 103, 3, 4. , 4. , 8.75, 0, 0.76),
       (222, 316, 110, 3, 3.5, 4. , 8.56, 0, 0.75),
       (223, 324, 113, 4, 4.5, 4. , 8.79, 0, 0.76),
       (224, 308, 109, 2, 3. , 4. , 8.45, 0, 0.71),
       (225, 305, 105, 2, 3. , 2. , 8.23, 0, 0.67),
       (226, 296,  99, 2, 2.5, 2.5, 8.03, 0, 0.61),
       (227, 306, 110, 2, 3.5, 4. , 8.45, 0, 0.63),
       (228, 312, 110, 2, 3.5, 3. , 8.53, 0, 0.64),
       (229, 318, 112, 3, 4. , 3.5, 8.67, 0, 0.71),
       (230, 324, 111, 4, 3. , 3. , 9.01, 1, 0.82),
       (231, 313, 104, 3, 4. , 4.5, 8.65, 0, 0.73),
       (232, 319, 106, 3, 3.5, 2.5, 8.33, 1, 0.74),
       (233, 312, 107, 2, 2.5, 3.5, 8.27, 0, 0.69),
       (234, 304, 100, 2, 2.5, 3.5, 8.07, 0, 0.64),
       (235, 330, 113, 5, 5. , 4. , 9.31, 1, 0.91),
       (236, 326, 111, 5, 4.5, 4. , 9.23, 1, 0.88),
       (237, 325, 112, 4, 4. , 4.5, 9.17, 1, 0.85),
       (238, 329, 114, 5, 4.5, 5. , 9.19, 1, 0.86),
       (239, 310, 104, 3, 2. , 3.5, 8.37, 0, 0.7 ),
       (240, 299, 100, 1, 1.5, 2. , 7.89, 0, 0.59),
       (241, 296, 101, 1, 2.5, 3. , 7.68, 0, 0.6 ),
       (242, 317, 103, 2, 2.5, 2. , 8.15, 0, 0.65),
       (243, 324, 115, 3, 3.5, 3. , 8.76, 1, 0.7 ),
       (244, 325, 114, 3, 3.5, 3. , 9.04, 1, 0.76),
       (245, 314, 107, 2, 2.5, 4. , 8.56, 0, 0.63),
       (246, 328, 110, 4, 4. , 2.5, 9.02, 1, 0.81),
       (247, 316, 105, 3, 3. , 3.5, 8.73, 0, 0.72),
       (248, 311, 104, 2, 2.5, 3.5, 8.48, 0, 0.71),
       (249, 324, 110, 3, 3.5, 4. , 8.87, 1, 0.8 ),
       (250, 321, 111, 3, 3.5, 4. , 8.83, 1, 0.77),
       (251, 320, 104, 3, 3. , 2.5, 8.57, 1, 0.74),
       (252, 316,  99, 2, 2.5, 3. , 9.  , 0, 0.7 ),
       (253, 318, 100, 2, 2.5, 3.5, 8.54, 1, 0.71),
       (254, 335, 115, 4, 4.5, 4.5, 9.68, 1, 0.93),
       (255, 321, 114, 4, 4. , 5. , 9.12, 0, 0.85),
       (256, 307, 110, 4, 4. , 4.5, 8.37, 0, 0.79),
       (257, 309,  99, 3, 4. , 4. , 8.56, 0, 0.76),
       (258, 324, 100, 3, 4. , 5. , 8.64, 1, 0.78),
       (259, 326, 102, 4, 5. , 5. , 8.76, 1, 0.77),
       (260, 331, 119, 4, 5. , 4.5, 9.34, 1, 0.9 ),
       (261, 327, 108, 5, 5. , 3.5, 9.13, 1, 0.87),
       (262, 312, 104, 3, 3.5, 4. , 8.09, 0, 0.71),
       (263, 308, 103, 2, 2.5, 4. , 8.36, 1, 0.7 ),
       (264, 324, 111, 3, 2.5, 1.5, 8.79, 1, 0.7 ),
       (265, 325, 110, 2, 3. , 2.5, 8.76, 1, 0.75),
       (266, 313, 102, 3, 2.5, 2.5, 8.68, 0, 0.71),
       (267, 312, 105, 2, 2. , 2.5, 8.45, 0, 0.72),
       (268, 314, 107, 3, 3. , 3.5, 8.17, 1, 0.73),
       (269, 327, 113, 4, 4.5, 5. , 9.14, 0, 0.83),
       (270, 308, 108, 4, 4.5, 5. , 8.34, 0, 0.77),
       (271, 306, 105, 2, 2.5, 3. , 8.22, 1, 0.72),
       (272, 299,  96, 2, 1.5, 2. , 7.86, 0, 0.54),
       (273, 294,  95, 1, 1.5, 1.5, 7.64, 0, 0.49),
       (274, 312,  99, 1, 1. , 1.5, 8.01, 1, 0.52),
       (275, 315, 100, 1, 2. , 2.5, 7.95, 0, 0.58),
       (276, 322, 110, 3, 3.5, 3. , 8.96, 1, 0.78),
       (277, 329, 113, 5, 5. , 4.5, 9.45, 1, 0.89),
       (278, 320, 101, 2, 2.5, 3. , 8.62, 0, 0.7 ),
       (279, 308, 103, 2, 3. , 3.5, 8.49, 0, 0.66),
       (280, 304, 102, 2, 3. , 4. , 8.73, 0, 0.67),
       (281, 311, 102, 3, 4.5, 4. , 8.64, 1, 0.68),
       (282, 317, 110, 3, 4. , 4.5, 9.11, 1, 0.8 ),
       (283, 312, 106, 3, 4. , 3.5, 8.79, 1, 0.81),
       (284, 321, 111, 3, 2.5, 3. , 8.9 , 1, 0.8 ),
       (285, 340, 112, 4, 5. , 4.5, 9.66, 1, 0.94),
       (286, 331, 116, 5, 4. , 4. , 9.26, 1, 0.93),
       (287, 336, 118, 5, 4.5, 4. , 9.19, 1, 0.92),
       (288, 324, 114, 5, 5. , 4.5, 9.08, 1, 0.89),
       (289, 314, 104, 4, 5. , 5. , 9.02, 0, 0.82),
       (290, 313, 109, 3, 4. , 3.5, 9.  , 0, 0.79),
       (291, 307, 105, 2, 2.5, 3. , 7.65, 0, 0.58),
       (292, 300, 102, 2, 1.5, 2. , 7.87, 0, 0.56),
       (293, 302,  99, 2, 1. , 2. , 7.97, 0, 0.56),
       (294, 312,  98, 1, 3.5, 3. , 8.18, 1, 0.64),
       (295, 316, 101, 2, 2.5, 2. , 8.32, 1, 0.61),
       (296, 317, 100, 2, 3. , 2.5, 8.57, 0, 0.68),
       (297, 310, 107, 3, 3.5, 3.5, 8.67, 0, 0.76),
       (298, 320, 120, 3, 4. , 4.5, 9.11, 0, 0.86),
       (299, 330, 114, 3, 4.5, 4.5, 9.24, 1, 0.9 ),
       (300, 305, 112, 3, 3. , 3.5, 8.65, 0, 0.71),
       (301, 309, 106, 2, 2.5, 2.5, 8.  , 0, 0.62),
       (302, 319, 108, 2, 2.5, 3. , 8.76, 0, 0.66),
       (303, 322, 105, 2, 3. , 3. , 8.45, 1, 0.65),
       (304, 323, 107, 3, 3.5, 3.5, 8.55, 1, 0.73),
       (305, 313, 106, 2, 2.5, 2. , 8.43, 0, 0.62),
       (306, 321, 109, 3, 3.5, 3.5, 8.8 , 1, 0.74),
       (307, 323, 110, 3, 4. , 3.5, 9.1 , 1, 0.79),
       (308, 325, 112, 4, 4. , 4. , 9.  , 1, 0.8 ),
       (309, 312, 108, 3, 3.5, 3. , 8.53, 0, 0.69),
       (310, 308, 110, 4, 3.5, 3. , 8.6 , 0, 0.7 ),
       (311, 320, 104, 3, 3. , 3.5, 8.74, 1, 0.76),
       (312, 328, 108, 4, 4.5, 4. , 9.18, 1, 0.84),
       (313, 311, 107, 4, 4.5, 4.5, 9.  , 1, 0.78),
       (314, 301, 100, 3, 3.5, 3. , 8.04, 0, 0.67),
       (315, 305, 105, 2, 3. , 4. , 8.13, 0, 0.66),
       (316, 308, 104, 2, 2.5, 3. , 8.07, 0, 0.65),
       (317, 298, 101, 2, 1.5, 2. , 7.86, 0, 0.54),
       (318, 300,  99, 1, 1. , 2.5, 8.01, 0, 0.58),
       (319, 324, 111, 3, 2.5, 2. , 8.8 , 1, 0.79),
       (320, 327, 113, 4, 3.5, 3. , 8.69, 1, 0.8 ),
       (321, 317, 106, 3, 4. , 3.5, 8.5 , 1, 0.75),
       (322, 323, 104, 3, 4. , 4. , 8.44, 1, 0.73),
       (323, 314, 107, 2, 2.5, 4. , 8.27, 0, 0.72),
       (324, 305, 102, 2, 2. , 2.5, 8.18, 0, 0.62),
       (325, 315, 104, 3, 3. , 2.5, 8.33, 0, 0.67),
       (326, 326, 116, 3, 3.5, 4. , 9.14, 1, 0.81),
       (327, 299, 100, 3, 2. , 2. , 8.02, 0, 0.63),
       (328, 295, 101, 2, 2.5, 2. , 7.86, 0, 0.69),
       (329, 324, 112, 4, 4. , 3.5, 8.77, 1, 0.8 ),
       (330, 297,  96, 2, 2.5, 1.5, 7.89, 0, 0.43),
       (331, 327, 113, 3, 3.5, 3. , 8.66, 1, 0.8 ),
       (332, 311, 105, 2, 3. , 2. , 8.12, 1, 0.73),
       (333, 308, 106, 3, 3.5, 2.5, 8.21, 1, 0.75),
       (334, 319, 108, 3, 3. , 3.5, 8.54, 1, 0.71),
       (335, 312, 107, 4, 4.5, 4. , 8.65, 1, 0.73),
       (336, 325, 111, 4, 4. , 4.5, 9.11, 1, 0.83),
       (337, 319, 110, 3, 3. , 2.5, 8.79, 0, 0.72),
       (338, 332, 118, 5, 5. , 5. , 9.47, 1, 0.94),
       (339, 323, 108, 5, 4. , 4. , 8.74, 1, 0.81),
       (340, 324, 107, 5, 3.5, 4. , 8.66, 1, 0.81),
       (341, 312, 107, 3, 3. , 3. , 8.46, 1, 0.75),
       (342, 326, 110, 3, 3.5, 3.5, 8.76, 1, 0.79),
       (343, 308, 106, 3, 3. , 3. , 8.24, 0, 0.58),
       (344, 305, 103, 2, 2.5, 3.5, 8.13, 0, 0.59),
       (345, 295,  96, 2, 1.5, 2. , 7.34, 0, 0.47),
       (346, 316,  98, 1, 1.5, 2. , 7.43, 0, 0.49),
       (347, 304,  97, 2, 1.5, 2. , 7.64, 0, 0.47),
       (348, 299,  94, 1, 1. , 1. , 7.34, 0, 0.42),
       (349, 302,  99, 1, 2. , 2. , 7.25, 0, 0.57),
       (350, 313, 101, 3, 2.5, 3. , 8.04, 0, 0.62),
       (351, 318, 107, 3, 3. , 3.5, 8.27, 1, 0.74),
       (352, 325, 110, 4, 3.5, 4. , 8.67, 1, 0.73),
       (353, 303, 100, 2, 3. , 3.5, 8.06, 1, 0.64),
       (354, 300, 102, 3, 3.5, 2.5, 8.17, 0, 0.63),
       (355, 297,  98, 2, 2.5, 3. , 7.67, 0, 0.59),
       (356, 317, 106, 2, 2. , 3.5, 8.12, 0, 0.73),
       (357, 327, 109, 3, 3.5, 4. , 8.77, 1, 0.79),
              (358, 301, 104, 2, 3.5, 3.5, 7.89, 1, 0.68),
              (359, 314, 105, 2, 2.5, 2. , 7.64, 0, 0.7 ),
              (360, 321, 107, 2, 2. , 1.5, 8.44, 0, 0.81),
              (361, 322, 110, 3, 4. , 5. , 8.64, 1, 0.85),
              (362, 334, 116, 4, 4. , 3.5, 9.54, 1, 0.93),
              (363, 338, 115, 5, 4.5, 5. , 9.23, 1, 0.91),
              (364, 306, 103, 2, 2.5, 3. , 8.36, 0, 0.69),
              (365, 313, 102, 3, 3.5, 4. , 8.9 , 1, 0.77),
              (366, 330, 114, 4, 4.5, 3. , 9.17, 1, 0.86),
              (367, 320, 104, 3, 3.5, 4.5, 8.34, 1, 0.74),
              (368, 311,  98, 1, 1. , 2.5, 7.46, 0, 0.57),
              (369, 298,  92, 1, 2. , 2. , 7.88, 0, 0.51),
              (370, 301,  98, 1, 2. , 3. , 8.03, 1, 0.67),
              (371, 310, 103, 2, 2.5, 2.5, 8.24, 0, 0.72),
              (372, 324, 110, 3, 3.5, 3. , 9.22, 1, 0.89),
              (373, 336, 119, 4, 4.5, 4. , 9.62, 1, 0.95),
              (374, 321, 109, 3, 3. , 3. , 8.54, 1, 0.79),
              (375, 315, 105, 2, 2. , 2.5, 7.65, 0, 0.39),
              (376, 304, 101, 2, 2. , 2.5, 7.66, 0, 0.38),
              (377, 297,  96, 2, 2.5, 2. , 7.43, 0, 0.34),
              (378, 290, 100, 1, 1.5, 2. , 7.56, 0, 0.47),
              (379, 303,  98, 1, 2. , 2.5, 7.65, 0, 0.56),
              (380, 311,  99, 1, 2.5, 3. , 8.43, 1, 0.71),
              (381, 322, 104, 3, 3.5, 4. , 8.84, 1, 0.78),
              (382, 319, 105, 3, 3. , 3.5, 8.67, 1, 0.73),
              (383, 324, 110, 4, 4.5, 4. , 9.15, 1, 0.82),
              (384, 300, 100, 3, 3. , 3.5, 8.26, 0, 0.62),
              (385, 340, 113, 4, 5. , 5. , 9.74, 1, 0.96),
              (386, 335, 117, 5, 5. , 5. , 9.82, 1, 0.96),
              (387, 302, 101, 2, 2.5, 3.5, 7.96, 0, 0.46),
              (388, 307, 105, 2, 2. , 3.5, 8.1 , 0, 0.53),
              (389, 296,  97, 2, 1.5, 2. , 7.8 , 0, 0.49),
              (390, 320, 108, 3, 3.5, 4. , 8.44, 1, 0.76),
              (391, 314, 102, 2, 2. , 2.5, 8.24, 0, 0.64),
              (392, 318, 106, 3, 2. , 3. , 8.65, 0, 0.71),
              (393, 326, 112, 4, 4. , 3.5, 9.12, 1, 0.84),
              (394, 317, 104, 2, 3. , 3. , 8.76, 0, 0.77),
              (395, 329, 111, 4, 4.5, 4. , 9.23, 1, 0.89),
              (396, 324, 110, 3, 3.5, 3.5, 9.04, 1, 0.82),
              (397, 325, 107, 3, 3. , 3.5, 9.11, 1, 0.84),
              (398, 330, 116, 4, 5. , 4.5, 9.45, 1, 0.91),
              (399, 312, 103, 3, 3.5, 4. , 8.78, 0, 0.67),
              (400, 333, 117, 4, 5. , 4. , 9.66, 1, 0.95)],
             dtype=[('Serial_No', '<i8'), ('GRE_Score', '<i8'), ('TOEFL_Score', '<i8'),
       ('University_Rating', '<i8'), ('SOP', '<f8'), ('LOR', '<f8'), ('CGPA', '<f8'),
       ('Research', '<i8'), ('Chance_of_Admit', '<f8')])
```

[57]:
```
# Notice that the resulting array is actually a one-dimensional array with 400␣
↪tuples
graduate_admission.shape
```

[57]:
```
(400,)
```

[58]:
```
# We can retrieve a column from the array using the column's name for example,␣
↪let's get the CGPA column and
# only the first five values.
graduate_admission['CGPA'][0:5]
```

[58]:
```
array([9.65, 8.87, 8.  , 8.67, 8.21])
```

[59]:
```
# Since the GPA in the dataset range from 1 to 10, and in the US it's more␣
↪common to use a scale of up to 4,
# a common task might be to convert the GPA by dividing by 10 and then␣
↪multiplying by 4
graduate_admission['CGPA'] = graduate_admission['CGPA'] /10 *4
graduate_admission['CGPA'][0:20] #let's get 20 values
```

[59]:
```
array([3.86 , 3.548, 3.2  , 3.468, 3.284, 3.736, 3.28 , 3.16 , 3.2  ,
       3.44 , 3.36 , 3.6  , 3.64 , 3.2  , 3.28 , 3.32 , 3.48 , 3.2  ,
       3.52 , 3.4  ])
```

[60]:
```
# Recall boolean masking. We can use this to find out how many students have␣
↪had research experience by
# creating a boolean mask and passing it to the array indexing operator
len(graduate_admission[graduate_admission['Research'] == 1])
```

[60]:
```
219
```

[61]:
```
# Since we have the data field chance of admission, which ranges from 0 to 1,␣
↪we can try to see if students
# with high chance of admission (>0.8) on average have higher GRE score than␣
↪those with lower chance of
# admission (<0.4)

# So first we use boolean masking to pull out only those students we are␣
↪interested in based on their chance
# of admission, then we pull out only their GPA scores, then we print the mean␣
↪values.
print(graduate_admission[graduate_admission['Chance_of_Admit'] > 0.
↪8]['GRE_Score'].mean())
print(graduate_admission[graduate_admission['Chance_of_Admit'] < 0.
↪4]['GRE_Score'].mean())
```

```
328.7350427350427
302.2857142857143
```

[62]:
```
# Take a moment to reflect here, do you understand what is happening in these␣
↪calls?

# When we do the boolean masking we are left with an array with tuples in it␣
↪still, and numpy holds underneath
# this a list of the columns we specified and their name and indexes
graduate_admission[graduate_admission['Chance_of_Admit'] > 0.8]
```

[62]:
```
array([(  1, 337, 118, 4, 4.5, 4.5, 3.86 , 1, 0.92),
       (  6, 330, 115, 5, 4.5, 3. , 3.736, 1, 0.9 ),
       ( 12, 327, 111, 4, 4. , 4.5, 3.6  , 1, 0.84),
       ( 23, 328, 116, 5, 5. , 5. , 3.8  , 1, 0.94),
       ( 24, 334, 119, 5, 5. , 4.5, 3.88 , 1, 0.95),
       ( 25, 336, 119, 5, 4. , 3.5, 3.92 , 1, 0.97),
       ( 26, 340, 120, 5, 4.5, 4.5, 3.84 , 1, 0.94),
       ( 33, 338, 118, 4, 3. , 4.5, 3.76 , 1, 0.91),
       ( 34, 340, 114, 5, 4. , 4. , 3.84 , 1, 0.9 ),
       ( 35, 331, 112, 5, 4. , 5. , 3.92 , 1, 0.94),
       ( 36, 320, 110, 5, 5. , 5. , 3.68 , 1, 0.88),
       ( 44, 332, 117, 4, 4.5, 4. , 3.64 , 0, 0.87),
       ( 45, 326, 113, 5, 4.5, 4. , 3.76 , 1, 0.91),
       ( 46, 322, 110, 5, 5. , 4. , 3.64 , 1, 0.88),
       ( 47, 329, 114, 5, 4. , 5. , 3.72 , 1, 0.86),
       ( 48, 339, 119, 5, 4.5, 4. , 3.88 , 0, 0.89),
       ( 49, 321, 110, 3, 3.5, 5. , 3.54 , 1, 0.82),
       ( 71, 332, 118, 5, 5. , 5. , 3.856, 1, 0.94),
       ( 72, 336, 112, 5, 5. , 5. , 3.904, 1, 0.96),
       ( 73, 321, 111, 5, 5. , 5. , 3.78 , 1, 0.93),
       ( 74, 314, 108, 4, 4.5, 4. , 3.616, 1, 0.84),
       ( 82, 340, 120, 4, 5. , 5. , 3.8  , 1, 0.96),
       ( 83, 320, 110, 5, 5. , 4.5, 3.688, 1, 0.92),
       ( 84, 322, 115, 5, 4. , 4.5, 3.744, 1, 0.92),
       ( 85, 340, 115, 5, 4.5, 4.5, 3.78 , 1, 0.94),
       ( 98, 331, 120, 3, 4. , 4. , 3.584, 1, 0.86),
       ( 99, 332, 119, 4, 5. , 4.5, 3.696, 1, 0.9 ),
       (107, 329, 111, 4, 4.5, 4.5, 3.672, 1, 0.87),
       (108, 338, 117, 4, 3.5, 4.5, 3.784, 1, 0.91),
       (109, 331, 116, 5, 5. , 5. , 3.752, 1, 0.93),
       (121, 335, 117, 5, 5. , 5. , 3.824, 1, 0.94),
       (122, 334, 119, 5, 4.5, 4.5, 3.792, 1, 0.94),
       (127, 323, 113, 3, 4. , 3. , 3.728, 1, 0.85),
       (129, 326, 112, 3, 3.5, 3. , 3.64 , 1, 0.84),
       (130, 333, 118, 5, 5. , 5. , 3.74 , 1, 0.92),
       (131, 339, 114, 5, 4. , 4.5, 3.904, 1, 0.96),
       (135, 333, 113, 5, 4. , 4. , 3.712, 1, 0.89),
       (136, 314, 109, 4, 3.5, 4. , 3.508, 1, 0.82),
       (141, 329, 110, 2, 4. , 3. , 3.66 , 1, 0.84),
       (142, 332, 118, 2, 4.5, 3.5, 3.744, 1, 0.9 ),
       (143, 331, 115, 5, 4. , 3.5, 3.776, 1, 0.92),
       (144, 340, 120, 4, 4.5, 4. , 3.968, 1, 0.97),
       (146, 320, 113, 2, 2. , 2.5, 3.456, 1, 0.81),
       (148, 326, 114, 3, 3. , 3. , 3.644, 1, 0.83),
       (149, 339, 116, 4, 4. , 3.5, 3.92 , 1, 0.96),
       (151, 334, 114, 4, 4. , 4. , 3.772, 1, 0.93),
       (152, 332, 116, 5, 5. , 5. , 3.712, 1, 0.94),
       (153, 321, 112, 5, 5. , 5. , 3.624, 1, 0.86),
       (165, 329, 111, 4, 4.5, 4. , 3.604, 1, 0.81),
       (172, 334, 117, 5, 4. , 4.5, 3.628, 1, 0.89),
       (173, 322, 110, 4, 4. , 5. , 3.652, 1, 0.86),
       (174, 323, 113, 4, 4. , 4.5, 3.692, 1, 0.89),
       (175, 321, 111, 4, 4. , 4. , 3.588, 1, 0.87),
       (176, 320, 111, 4, 4.5, 3.5, 3.548, 1, 0.85),
       (177, 329, 119, 4, 4.5, 4.5, 3.664, 1, 0.9 ),
       (178, 319, 110, 3, 3.5, 3.5, 3.616, 0, 0.82),
       (186, 327, 113, 4, 4.5, 4.5, 3.644, 1, 0.89),
       (187, 317, 107, 3, 3.5, 3. , 3.472, 1, 0.84),
       (188, 335, 118, 5, 4.5, 3.5, 3.776, 1, 0.93),
       (189, 331, 115, 5, 4.5, 3.5, 3.744, 1, 0.93),
       (190, 324, 112, 5, 5. , 5. , 3.632, 1, 0.88),
       (191, 324, 111, 5, 4.5, 4. , 3.664, 1, 0.9 ),
       (192, 323, 110, 5, 4. , 5. , 3.592, 1, 0.87),
       (193, 322, 114, 5, 4.5, 4. , 3.576, 1, 0.86),
       (194, 336, 118, 5, 4.5, 5. , 3.812, 1, 0.94),
       (203, 340, 120, 5, 4.5, 4.5, 3.964, 1, 0.97),
       (204, 334, 120, 5, 4. , 5. , 3.948, 1, 0.97),
       (212, 328, 110, 4, 5. , 4. , 3.656, 1, 0.82),
       (213, 338, 120, 4, 5. , 5. , 3.864, 1, 0.95),
       (214, 333, 119, 5, 5. , 4.5, 3.912, 1, 0.96),
       (215, 331, 117, 4, 4.5, 5. , 3.768, 1, 0.94),
       (216, 330, 116, 5, 5. , 4.5, 3.744, 1, 0.93),
       (217, 322, 112, 4, 4.5, 4.5, 3.704, 1, 0.91),
       (218, 321, 109, 4, 4. , 4. , 3.652, 1, 0.85),
       (219, 324, 110, 4, 3. , 3.5, 3.588, 1, 0.84),
       (230, 324, 111, 4, 3. , 3. , 3.604, 1, 0.82),
       (235, 330, 113, 5, 5. , 4. , 3.724, 1, 0.91),
       (236, 326, 111, 5, 4.5, 4. , 3.692, 1, 0.88),
       (237, 325, 112, 4, 4. , 4.5, 3.668, 1, 0.85),
       (238, 329, 114, 5, 4.5, 5. , 3.676, 1, 0.86),
       (246, 328, 110, 4, 4. , 2.5, 3.608, 1, 0.81),
       (254, 335, 115, 4, 4.5, 4.5, 3.872, 1, 0.93),
       (255, 321, 114, 4, 4. , 5. , 3.648, 0, 0.85),
       (260, 331, 119, 4, 5. , 4.5, 3.736, 1, 0.9 ),
       (261, 327, 108, 5, 5. , 3.5, 3.652, 1, 0.87),
       (269, 327, 113, 4, 4.5, 5. , 3.656, 0, 0.83),
               (277, 329, 113, 5, 5. , 4.5, 3.78 , 1, 0.89),
               (283, 312, 106, 3, 4. , 3.5, 3.516, 1, 0.81),
               (285, 340, 112, 4, 5. , 4.5, 3.864, 1, 0.94),
               (286, 331, 116, 5, 4. , 4. , 3.704, 1, 0.93),
               (287, 336, 118, 5, 4.5, 4. , 3.676, 1, 0.92),
               (288, 324, 114, 5, 5. , 4.5, 3.632, 1, 0.89),
               (289, 314, 104, 4, 5. , 5. , 3.608, 0, 0.82),
               (298, 320, 120, 3, 4. , 4.5, 3.644, 0, 0.86),
               (299, 330, 114, 3, 4.5, 4.5, 3.696, 1, 0.9 ),
               (312, 328, 108, 4, 4.5, 4. , 3.672, 1, 0.84),
               (326, 326, 116, 3, 3.5, 4. , 3.656, 1, 0.81),
               (336, 325, 111, 4, 4. , 4.5, 3.644, 1, 0.83),
               (338, 332, 118, 5, 5. , 5. , 3.788, 1, 0.94),
               (339, 323, 108, 5, 4. , 4. , 3.496, 1, 0.81),
               (340, 324, 107, 5, 3.5, 4. , 3.464, 1, 0.81),
               (360, 321, 107, 2, 2. , 1.5, 3.376, 0, 0.81),
               (361, 322, 110, 3, 4. , 5. , 3.456, 1, 0.85),
               (362, 334, 116, 4, 4. , 3.5, 3.816, 1, 0.93),
               (363, 338, 115, 5, 4.5, 5. , 3.692, 1, 0.91),
               (366, 330, 114, 4, 4.5, 3. , 3.668, 1, 0.86),
               (372, 324, 110, 3, 3.5, 3. , 3.688, 1, 0.89),
               (373, 336, 119, 4, 4.5, 4. , 3.848, 1, 0.95),
               (383, 324, 110, 4, 4.5, 4. , 3.66 , 1, 0.82),
               (385, 340, 113, 4, 5. , 5. , 3.896, 1, 0.96),
               (386, 335, 117, 5, 5. , 5. , 3.928, 1, 0.96),
               (393, 326, 112, 4, 4. , 3.5, 3.648, 1, 0.84),
               (395, 329, 111, 4, 4.5, 4. , 3.692, 1, 0.89),
               (396, 324, 110, 3, 3.5, 3.5, 3.616, 1, 0.82),
               (397, 325, 107, 3, 3. , 3.5, 3.644, 1, 0.84),
               (398, 330, 116, 4, 5. , 4.5, 3.78 , 1, 0.91),
               (400, 333, 117, 4, 5. , 4. , 3.864, 1, 0.95)],
              dtype=[('Serial_No', '<i8'), ('GRE_Score', '<i8'), ('TOEFL_Score', '<i8'),
       ('University_Rating', '<i8'), ('SOP', '<f8'), ('LOR', '<f8'), ('CGPA', '<f8'),
       ('Research', '<i8'), ('Chance_of_Admit', '<f8')])
```

[63]:
```
# Let's also do this with GPA
print(graduate_admission[graduate_admission['Chance_of_Admit'] > 0.8]['CGPA'].
 ↪mean())
print(graduate_admission[graduate_admission['Chance_of_Admit'] < 0.4]['CGPA'].
 ↪mean())
```

```
3.7106666666666666
3.0222857142857142
```

[64]:
```
# Hrm, well, I guess one could have expected this. The GPA and GRE for students␣
 ↪who have a higher chance of
# being admitted, at least based on our cursory look here, seems to be higher.
```

So that’s a bit of a whirlwing tour of numpy, the core scientific computing library in python. Now, you’re going to see a lot more of this kind of discussion, as the library we’ll be focusing on in this course is pandas, which is built on top of numpy. Don’t worry if it didn’t all sink in the first time, we’re going to dig in to most of these topics again with pandas. However, it’s useful to know that many of the functions and capabilities of numpy are available to you within pandas.

Regex_ed

November 15, 2021

In this lecture we're going to talk about pattern matching in strings using regular expressions. Regular expressions, or regexes, are written in a condensed formatting language. In general, you can think of a regular expression as a pattern which you give to a regex processor with some source data. The processor then parses that source data using that pattern, and returns chunks of text back to the a data scientist or programmer for further manipulation. There's really three main reasons you would want to do this - to check whether a pattern exists within some source data, to get all instances of a complex pattern from some source data, or to clean your source data using a pattern generally through string splitting. Regexes are not trivial, but they are a foundational technique for data cleaning in data science applications, and a solid understanding of regexs will help you quickly and efficiently manipulate text data for further data science application.

Now, you could teach a whole course on regular expressions alone, especially if you wanted to demystify how the regex parsing engine works and efficient mechanisms for parsing text. In this lecture I want to give you basic understanding of how regex works - enough knowledge that, with a little directed sleuthing, you'll be able to make sense of the regex patterns you see others use, and you can build up your practical knowledge of how to use regexes to improve your data cleaning. By the end of this lecture, you will understand the basics of regular expressions, how to define patterns for matching, how to apply these patterns to strings, and how to use the results of those patterns in data processing.

Finally, a note that in order to best learn regexes you need to write regexes. I encourage you to stop the video at any time and try out new patterns or syntax you learn at any time.

[1]:
```
# First we'll import the re module, which is where python stores regular␣
 ↪expression libraries.
import re
```

[2]:
```
# There are several main processing functions in re that you might use. The␣
 ↪first, match() checks for a match
# that is at the beginning of the string and returns a boolean. Similarly,␣
 ↪search(), checks for a match
# anywhere in the string, and returns a boolean.

# Lets create some text for an example
text = "This is a good day."

# Now, lets see if it's a good day or not:
if re.search("good", text): # the first parameter here is the pattern
    print("Wonderful!")
else:
    print("Alas :(")
```

```
Wonderful!
```

[3]:
```
# In addition to checking for conditionals, we can segment a string. The work␣
↪that regex does here is called
# tokenizing, where the string is separated into substrings based on patterns.␣
↪Tokenizing is a core activity
# in natural language processing, which we won't talk much about here but that␣
↪you will study in the future

# The findall() and split() functions will parse the string for us and return␣
↪chunks. Lets try and example
text = "Amy works diligently. Amy gets good grades. Our student Amy is␣
↪succesful."

# This is a bit of a fabricated example, but lets split this on all instances␣
↪of Amy
re.split("Amy", text)
```

[3]:
```
['',
 ' works diligently. ',
 ' gets good grades. Our student ',
 ' is succesful.']
```

[4]:
```
# You'll notice that split has returned an empty string, followed by a number␣
↪of statements about Amy, all as
# elements of a list. If we wanted to count how many times we have talked about␣
↪Amy, we could use findall()
re.findall("Amy", text)
```

[4]:
```
['Amy', 'Amy', 'Amy']
```

[5]:
```
# Ok, so we've seen that .search() looks for some pattern and returns a␣
↪boolean, that .split() will use a
# pattern for creating a list of substrings, and that .findall() will look for␣
↪a pattern and pull out all
# occurences.
```

[6]:
```
# Now that we know how the python regex API works, lets talk about more complex␣
↪patterns. The regex
# specification standard defines a markup language to describe patterns in text.
↪ Lets start with anchors.
# Anchors specify the start and/or the end of the string that you are trying to␣
↪match. The caret character ^
# means start and the dollar sign character $ means end. If you put ^ before a␣
↪string, it means that the text
# the regex processor retrieves must start with the string you specify. For␣
↪ending, you have to put the $
```

```
# character after the string, it means that the text Regex retrieves must end␣
↪with the string you specify.

# Here's an example
text = "Amy works diligently. Amy gets good grades. Our student Amy is␣
↪succesful."

# Lets see if this begins with Amy
re.search("^Amy",text)
```

[6]:
```
<re.Match object; span=(0, 3), match='Amy'>
```

[7]:
```
# Notice that re.search() actually returned to us a new object, called re.Match␣
↪object. An re.Match object
# always has a boolean value of True, as something was found, so you can always␣
↪evaluate it in an if statement
# as we did earlier. The rendering of the match object also tells you what␣
↪pattern was matched, in this case
# the word Amy, and the location the match was in, as the span.
```

1 Patterns and Character Classes

[8]:
```
# Let's talk more about patterns and start with character classes. Let's create␣
↪a string of a single learners'
# grades over a semester in one course across all of their assignments
                                grades="ACAAAABCBCBAA"

# If we want to answer the question "How many B's were in the grade list?" we␣
↪would just use B
re.findall("B",grades)
```

[8]:
```
['B', 'B', 'B']
```

[9]:
```
# If we wanted to count the number of A's or B's in the list, we can't use "AB"␣
↪since this is used to match
# all A's followed immediately by a B. Instead, we put the characters A and B␣
↪inside square brackets
re.findall("[AB]",grades)
```

[9]:
```
['A', 'A', 'A', 'A', 'A', 'B', 'B', 'B', 'A', 'A']
```

[10]:
```
# This is called the set operator. You can also include a range of characters,␣
↪which are ordered
# alphanumerically. For instance, if we want to refer to all lower case letters␣
↪we could use [a-z] Lets build
# a simple regex to parse out all instances where this student receive an A␣
↪followed by a B or a C
re.findall("[A][B-C]",grades)
```

[10]:
```
['AC', 'AB']
```

[11]:
```
# Notice how the [AB] pattern describes a set of possible characters which␣
↪could be either (A OR B), while the
# [A][B-C] pattern denoted two sets of characters which must have been matched␣
↪back to back. You can write
# this pattern by using the pipe operator, which means OR
re.findall("AB|AC",grades)
```

[11]:
```
['AC', 'AB']
```

[12]:
```
# We can use the caret with the set operator to negate our results. For␣
↪instance, if we want to parse out only
# the grades which were not A's
re.findall("[^A]",grades)
```

[12]:
```
['C', 'B', 'C', 'B', 'C', 'B']
```

[13]:
```
# Note this carefully - the caret was previously matched to the beginning of a␣
↪string as an anchor point, but
# inside of the set operator the caret, and the other special characters we␣
↪will be talking about, lose their
# meaning. This can be a bit confusing. What do you think the result would be␣
↪of this?
re.findall("^[^A]",grades)
```

[13]:
```
[]
```

[14]:
```
# It's an empty list, because the regex says that we want to match any value at␣
↪the beginning of the string
# which is not an A. Our string though starts with an A, so there is no match␣
↪found. And remember when you are
# using the set operator you are doing character-based matching. So you are␣
↪matching individual characters in
# an OR method.
```

2 Quantifiers

[15]:
```
# Ok, so we've talked about anchors and matching to the beginning and end of␣
↪patterns. And we've talked about
# characters and using sets with the [] notation. We've also talked about␣
↪character negation, and how the pipe
# | character allows us to or operations. Lets move on to quantifiers.
```

[16]:
```
# Quantifiers are the number of times you want a pattern to be matched in order␣
↪to match. The most basic
# quantifier is expressed as e{m,n}, where e is the expression or character we␣
↪are matching, m is the minimum
```

```
# number of times you want it to matched, and n is the maximum number of times␣
↪the item could be matched.

# Let's use these grades as an example. How many times has this student been on␣
↪a back-to-back A's streak?
re.findall("A{2,10}",grades) # we'll use 2 as our min, but ten as our max
```

[16]:
```
['AAAA', 'AA']
```

[17]:
```
# So we see that there were two streaks, one where the student had four A's,␣
↪and one where they had only two
# A's

# We might try and do this using single values and just repeating the pattern
re.findall("A{1,1}A{1,1}",grades)
```

[17]:
```
['AA', 'AA', 'AA']
```

[18]:
```
# As you can see, this is different than the first example. The first pattern␣
↪is looking for any combination
# of two A's up to ten A's in a row. So it sees four A's as a single streak.␣
↪The second pattern is looking for
# two A's back to back, so it sees two A's followed immediately by two more A's.
↪ We say that the regex
# processor begins at the start of the string and consumes variables which␣
↪match patterns as it does.

# It's important to note that the regex quantifier syntax does not allow you to␣
↪deviate from the {m,n}
# pattern. In particular, if you have an extra space in between the braces␣
↪you'll get an empty result
re.findall("A{2, 2}",grades)
```

[18]:
```
[]
```

[19]:
```
# And as we have already seen, if we don't include a quantifier then the␣
↪default is {1,1}
re.findall("AA",grades)
```

[19]:
```
['AA', 'AA', 'AA']
```

[20]:
```
# Oh, and if you just have one number in the braces, it's considered to be both␣
↪m and n
re.findall("A{2}",grades)
```

[20]:
```
['AA', 'AA', 'AA']
```

[21]:
```
# Using this, we could find a decreasing trend in a student's grades
re.findall("A{1,10}B{1,10}C{1,10}",grades)
```

[21]:
```
['AAAABC']
```

[22]:
```python
# Now, that's a bit of a hack, because we included a maximum that was just␣
↪arbitrarily large. There are three
# other quantifiers that are used as short hand, an asterix * to match 0 or␣
↪more times, a question mark ? to
# match one or more times, or a + plus sign to match one or more times. Lets␣
↪look at a more complex example,
# and load some data scraped from wikipedia
with open("datasets/ferpa.txt","r") as file:
    # we'll read that into a variable called wiki
    wiki=file.read()
# and lets print that variable out to the screen
wiki
```

[22]:
```
'Overview[edit]\nFERPA gives parents access to their child\'s education records,
an opportunity to seek to have the records amended, and some control over the
disclosure of information from the records. With several exceptions, schools
must have a student\'s consent prior to the disclosure of education records
after that student is 18 years old. The law applies only to educational agencies
and institutions that receive funds under a program administered by the U.S.
Department of Education.\n\nOther regulations under this act, effective starting
January 3, 2012, allow for greater disclosures of personal and directory student
identifying information and regulate student IDs and e-mail addresses.[2] For
example, schools may provide external companies with a student\'s personally
identifiable information without the student\'s consent.[2]\n\nExamples of
situations affected by FERPA include school employees divulging information to
anyone other than the student about the student\'s grades or behavior, and
school work posted on a bulletin board with a grade. Generally, schools must
have written permission from the parent or eligible student in order to release
any information from a student\'s education record.\n\nThis privacy policy also
governs how state agencies transmit testing data to federal agencies, such as
the Education Data Exchange Network.\n\nThis U.S. federal law also gave students
18 years of age or older, or students of any age if enrolled in any post-
secondary educational institution, the right of privacy regarding grades,
enrollment, and even billing information unless the school has specific
permission from the student to share that specific type of information.\n\nFERPA
also permits a school to disclose personally identifiable information from
education records of an "eligible student" (a student age 18 or older or
enrolled in a postsecondary institution at any age) to his or her parents if the
student is a "dependent student" as that term is defined in Section 152 of the
Internal Revenue Code. Generally, if either parent has claimed the student as a
dependent on the parent\'s most recent income tax statement, the school may non-
consensually disclose the student\'s education records to both
parents.[3]\n\nThe law allowed students who apply to an educational institution
such as graduate school permission to view recommendations submitted by others
as part of the application. However, on standard application forms, students are
given the option to waive this right.\n\nFERPA specifically excludes employees
of an educational institution if they are not students.\n\nThe act is also
```

```
referred to as the Buckley Amendment, for one of its proponents, Senator James
L. Buckley of New York.\n\nAccess to public records[edit]\nThe citing of FERPA
to conceal public records that are not "educational" in nature has been widely
criticized, including by the act\'s primary Senate sponsor.[4] For example, in
the Owasso Independent School District v. Falvo case, an important part of the
debate was determining the relationship between peer-grading and "education
records" as defined in FERPA. In the Court of Appeals, it was ruled that
students placing grades on the work of other students made such work into an
"education record." Thus, peer-grading was determined as a violation of FERPA
privacy policies because students had access to other students\' academic
performance without full consent.[5] However, when the case went to the Supreme
Court, it was officially ruled that peer-grading was not a violation of FERPA.
This is because a grade written on a student\'s work does not become an
"education record" until the teacher writes the final grade into a grade
book.[6]\n\nStudent medical records[edit]\nLegal experts have debated the issue
of whether student medical records (for example records of therapy sessions with
a therapist at an on-campus counseling center) might be released to the school
administration under certain triggering events, such as when a student sued his
college or university.[7][8]\n\nUsually, student medical treatment records will
remain under the protection of FERPA, not the Health Insurance Portability and
Accountability Act (HIPAA). This is due to the "FERPA Exception" written within
HIPAA.[9]'
```

[23]:
```
# Scanning through this document one of the things we notice is that the␣
 ↪headers all have the words [edit]
# behind them, followed by a newline character. So if we wanted to get a list␣
 ↪of all of the headers in this
# article we could do so using re.findall
re.findall("[a-zA-Z]{1,100}\[edit\]",wiki)
```

[23]:
```
['Overview[edit]', 'records[edit]', 'records[edit]']
```

[24]:
```
# Ok, that didn't quite work. It got all of the headers, but only the last word␣
 ↪of the header, and it really
# was quite clunky. Lets iteratively improve this. First, we can use \w to␣
 ↪match any letter, including digits
# and numbers.
re.findall("[\w]{1,100}\[edit\]",wiki)
```

[24]:
```
['Overview[edit]', 'records[edit]', 'records[edit]']
```

[25]:
```
# This is something new. \w is a metacharacter, and indicates a special pattern␣
 ↪of any letter or digit. There
# are actually a number of different metacharacters listed in the documentation.
 ↪ For instance, \s matches any
# whitespace character.

# Next, there are three other quantifiers we can use which shorten up the curly␣
 ↪brace syntax. We can use an
```

```
# asterix * to match 0 or more times, so let's try that.
re.findall("[\w]*\[edit\]",wiki)
```

[25]:
```
['Overview[edit]', 'records[edit]', 'records[edit]']
```

[26]:
```
# Now that we have shortened the regex, let's improve it a little bit. We can␣
 ↪add in a spaces using the space
# character
re.findall("[\w ]*\[edit\]",wiki)
```

[26]:
```
['Overview[edit]',
 'Access to public records[edit]',
 'Student medical records[edit]']
```

[27]:
```
# Ok, so this gets us the list of section titles in the wikipedia page! You can␣
 ↪now create a list of titles by
# iterating through this and applying another regex
for title in re.findall("[\w ]*\[edit\]",wiki):
    # Now we will take that intermediate result and split on the square bracket␣
 ↪[ just taking the first result
    print(re.split("[\[]",title)[0])
```

```
Overview
Access to public records
Student medical records
```

3 Groups

[28]:
```
# Ok, this works, but it's a bit of a pain. To this point we have been talking␣
 ↪about a regex as a single
# pattern which is matched. But, you can actually match different patterns,␣
 ↪called groups, at the same time,
# and then refer to the groups you want. To group patterns together you use␣
 ↪parentheses, which is actually
# pretty natural. Lets rewrite our findall using groups
re.findall("([\w ]*)(\[edit\])",wiki)
```

[28]:
```
[('Overview', '[edit]'),
 ('Access to public records', '[edit]'),
 ('Student medical records', '[edit]')]
```

[29]:
```
# Nice - we see that the python re module breaks out the result by group. We␣
 ↪can actually refer to groups by
# number as well with the match objects that are returned. But, how do we get␣
 ↪back a list of match objects?
# Thus far we've seen that findall() returns strings, and search() and match()␣
 ↪return individual Match
# objects. But what do we do if we want a list of Match objects? In this case,␣
 ↪we use the function finditer()
```

```
for item in re.finditer("([\w ]*)(\[edit\])",wiki):
    print(item.groups())
```

```
('Overview', '[edit]')
('Access to public records', '[edit]')
('Student medical records', '[edit]')
```

[30]:
```
# We see here that the groups() method returns a tuple of the group. We can get␣
↪an individual group using
# group(number), where group(0) is the whole match, and each other number is␣
↪the portion of the match we are
# interested in. In this case, we want group(1)
for item in re.finditer("([\w ]*)(\[edit\])",wiki):
    print(item.group(1))
```

```
Overview
Access to public records
Student medical records
```

[31]:
```
# One more piece to regex groups that I rarely use but is a good idea is␣
↪labeling or naming groups. In the
# previous example I showed you how you can use the position of the group. But␣
↪giving them a label and looking
# at the results as a dictionary is pretty useful. For that we use the syntax (?
↪P<name>), where the parethesis
# starts the group, the ?P indicates that this is an extension to basic␣
↪regexes, and <name> is the dictionary
# key we want to use wrapped in <>.
for item in re.finditer("(?P<title>[\w ]*)(?P<edit_link>\[edit\])",wiki):
    # We can get the dictionary returned for the item with .groupdict()
    print(item.groupdict()['title'])
```

```
Overview
Access to public records
Student medical records
```

[32]:
```
# Of course, we can print out the whole dictionary for the item too, and see␣
↪that the [edit] string is still
# in there. Here's the dictionary kept for the last match
print(item.groupdict())
```

```
{'title': 'Student medical records', 'edit_link': '[edit]'}
```

[33]:
```
# Ok, we have seen how we can match individual character patterns with [], how␣
↪we can group matches together
```

```
# using (), and how we can use quantifiers such as *, ?, or m{n} to describe␣
↪patterns. Something I glossed
# over in the previous example was the \w, which standards for any word␣
↪character. There are a number of
# short hands which are used with regexes for different kinds of characters,␣
↪including:
# a . for any single character which is not a newline
# a \d for any digit
# and \s for any whitespace character, like spaces and tabs
# There are more, and a full list can be found in the python documentation for␣
↪regexes
```

4 Look-ahead and Look-behind

[34]:
```
# One more concept to be familiar with is called "look ahead" and "look behind"␣
↪matching. In this case, the
# pattern being given to the regex engine is for text either before or after␣
↪the text we are trying to
# isolate. For example, in our headers we want to isolate text which  comes␣
↪before the [edit] rendering, but
# we actually don't care about the [edit] text itself. Thus far we have been␣
↪throwing the [edit] away, but if
# we want to use them to match but don't want to capture them we could put them␣
↪in a group and use look ahead
# instead with ?= syntax
for item in re.finditer("(?P<title>[\w ]+)(?=\[edit\])",wiki):
    # What this regex says is match two groups, the first will be named and␣
↪called title, will have any amount
    # of whitespace or regular word characters, the second will be the␣
↪characters [edit] but we don't actually
    # want this edit put in our output match objects
    print(item)
```

```
<re.Match object; span=(0, 8), match='Overview'>
<re.Match object; span=(2715, 2739), match='Access to public records'>
<re.Match object; span=(3692, 3715), match='Student medical records'>
```

5 Example: Wikipedia Data

[35]:
```
# Let's look at some more wikipedia data. Here's some data on universities in␣
↪the US which are buddhist-based
with open("datasets/buddhist.txt","r") as file:
    # we'll read that into a variable called wiki
    wiki=file.read()
```

```
# and lets print that variable out to the screen
wiki
```

[35]:
```
'Buddhist universities and colleges in the United States\nFrom Wikipedia, the
free encyclopedia\nJump to navigationJump to search\n\nThis article needs
additional citations for verification. Please help improve this article by
adding citations to reliable sources. Unsourced material may be challenged and
removed.\nFind sources: "Buddhist universities and colleges in the United
States"  news û newspapers û books û scholar û JSTOR (December 2009) (Learn how
and when to remove this template message)\nThere are several Buddhist
universities in the United States. Some of these have existed for decades and
are accredited. Others are relatively new and are either in the process of being
accredited or else have no formal accreditation. The list
includes:\n\nDhammakaya Open University  located in Azusa, California, part of
the Thai Wat Phra Dhammakaya[1]\nDharmakirti College  located in Tucson,
Arizona Now called Awam Tibetan Buddhist Institute
(http://awaminstitute.org/)\nDharma Realm Buddhist University  located in
Ukiah, California\nEwam Buddhist Institute  located in Arlee, Montana\nNaropa
University is located in Boulder, Colorado (Accredited by the Higher Learning
Commission)\nInstitute of Buddhist Studies  located in Berkeley,
California\nMaitripa College  located in Portland, Oregon\nSoka University of
America  located in Aliso Viejo, California\nUniversity of the West  located
in Rosemead, California\nWon Institute of Graduate Studies  located in
Glenside, Pennsylvania\nReferences[edit]\n^ Banchanon, Phongphiphat (3 February
2015).  "" [Getting to know the Dhammakaya network].
Forbes Thailand (in Thai). Retrieved 11 November 2016.\nExternal
links[edit]\nList of Buddhist Universities and Colleges in the world\n'
```

[36]:
```
# We can see that each university follows a fairly similar pattern, with the␣
↪name followed by an  then the
# words "located in" followed by the city and state

# I'll actually use this example to show you the verbose mode of python regexes.
↪ The verbose mode allows you
# to write multi-line regexes and increases readability. For this mode, we have␣
↪to explicitly indicate all
# whitespace characters, either by prepending them with a \ or by using the \s␣
↪special value. However, this
# means we can write our regex a bit more like code, and can even include␣
↪comments with #
pattern="""
(?P<title>.*)        #the university title
(\ located\ in\ )   #an indicator of the location
(?P<city>\w*)        #city the university is in
(,\ )                #separator for the state
(?P<state>\w*)       #the state the city is located in"""
```

assignment1

November 15, 2021

1 Assignment 1

For this assignment you are welcomed to use other regex resources such a regex “cheat sheets” you find on the web.

Before start working on the problems, here is a small example to help you understand how to write your own answers. In short, the solution should be written within the function body given, and the final result should be returned. Then the autograder will try to call the function and validate your returned result accordingly.

[1]:
```
def example_word_count():
    # This example question requires counting words in the example_string below.
    example_string = "Amy is 5 years old"

    # YOUR CODE HERE.
    # You should write your solution here, and return your result, you can␣
↪comment out or delete the
    # NotImplementedError below.
    result = example_string.split(" ")
    return len(result)

    #raise NotImplementedError()
```

1.1 Part A

Find a list of all of the names in the following string using regex.

[2]:
```
import re
def names():
    simple_string = """Amy is 5 years old, and her sister Mary is 2 years old.
    Ruth and Peter, their parents, have 3 kids."""

    # YOUR CODE HERE
    #raise NotImplementedError()
    names=re.findall("[A-Z]\w+",simple_string)
    #I would have written assert len(names)==4 afterwards, without the␣
↪parenthesis
    return names
names()
```

[2]:
```
['Amy', 'Mary', 'Ruth', 'Peter']
```

[ ]:
```
assert len(names()) == 4, "There are four names in the simple_string"
```

1.2 Part B

The dataset file in `assets/grades.txt` contains a line separated list of people with their grade in a class. Create a regex to generate a list of just those students who received a B in the course.

[10]:
```
import re
def grades():
    with open ("assets/grades.txt", "r") as file:
        grades = file.read()

    # YOUR CODE HERE
    liste=re.findall("[A-Z][\w ]+:\sB",grades)
    liste2=[]
    for i in liste:
        j=i.split(": ")
        liste2.append(j[0])
    return liste2
grades()
```

[10]:
```
['Bell Kassulke',
 'Simon Loidl',
 'Elias Jovanovic',
 'Hakim Botros',
 'Emilie Lorentsen',
 'Jake Wood',
 'Fatemeh Akhtar',
 'Kim Weston',
 'Yasmin Dar',
 'Viswamitra Upandhye',
 'Killian Kaufman',
 'Elwood Page',
 'Elodie Booker',
 'Adnan Chen',
 'Hank Spinka',
 'Hannah Bayer']
```

[ ]:
```
assert len(grades()) == 16
```

1.3 Part C

Consider the standard web log file in `assets/logdata.txt`. This file records the access a user makes when visiting a web page (like this one!). Each line of the log has the following items: * a host (e.g., '146.204.224.152') * a user_name (e.g., 'feest6811' **note: sometimes the user name is missing! In this case, use '-' as the value for the username.**) * the time a request was made (e.g., ‘21/Jun/2019:15:45:24 -0700’) * the post request type (e.g., ‘POST /incentivize HTTP/1.1’ **note: not everything is a POST!**)

Your task is to convert this into a list of dictionaries, where each dictionary looks like the following:

```
example_dict = {"host":"146.204.224.152",
                "user_name":"feest6811",
                "time":"21/Jun/2019:15:45:24 -0700",
                "request":"POST /incentivize HTTP/1.1"}
```

[1]:
```
import re
def logs():
    with open("assets/logdata.txt", "r") as file:
        logdata = file.read()
    pattern="""
    (?P<host>\w+\.\w+\.\w+\.\w+)
    (\s-\s)
    (?P<user_name>.*)
    (\s\[)
    (?P<time>.*)
    (\]\s)
    (")
    (?P<request>.*)
    (")
    """

    iter=re.finditer(pattern,logdata,re.VERBOSE)
    liste=[]
    for i in iter:
        liste.append(i.groupdict())
    print (len(liste))
    return liste
    # YOUR CODE HERE
    raise NotImplementedError()
logs()
```

```
979
```

[1]:
```
[{'host': '146.204.224.152',
  'user_name': 'feest6811',
  'time': '21/Jun/2019:15:45:24 -0700',
  'request': 'POST /incentivize HTTP/1.1'},
 {'host': '197.109.77.178',
  'user_name': 'kertzmann3129',
  'time': '21/Jun/2019:15:45:25 -0700',
  'request': 'DELETE /virtual/solutions/target/web+services HTTP/2.0'},
 {'host': '156.127.178.177',
  'user_name': 'okuneva5222',
 'time': '21/Jun/2019:15:45:27 -0700',
 'request': 'DELETE /interactive/transparent/niches/revolutionize HTTP/1.1'},
{'host': '100.32.205.59',
 'user_name': 'ortiz8891',
 'time': '21/Jun/2019:15:45:28 -0700',
 'request': 'PATCH /architectures HTTP/1.0'},
{'host': '168.95.156.240',
 'user_name': 'stark2413',
 'time': '21/Jun/2019:15:45:31 -0700',
 'request': 'GET /engage HTTP/2.0'},
{'host': '71.172.239.195',
 'user_name': 'dooley1853',
 'time': '21/Jun/2019:15:45:32 -0700',
 'request': 'PUT /cutting-edge HTTP/2.0'},
{'host': '180.95.121.94',
 'user_name': 'mohr6893',
 'time': '21/Jun/2019:15:45:34 -0700',
 'request': 'PATCH /extensible/reinvent HTTP/1.1'},
{'host': '144.23.247.108',
 'user_name': 'auer7552',
 'time': '21/Jun/2019:15:45:35 -0700',
 'request': 'POST /extensible/infrastructures/one-to-one/enterprise HTTP/1.1'},
{'host': '2.179.103.97',
 'user_name': 'lind8584',
 'time': '21/Jun/2019:15:45:36 -0700',
 'request': 'POST /grow/front-end/e-commerce/robust HTTP/2.0'},
{'host': '241.114.184.133',
 'user_name': 'tromp8355',
 'time': '21/Jun/2019:15:45:37 -0700',
 'request': 'GET /redefine/orchestrate HTTP/1.0'},
{'host': '224.188.38.4',
 'user_name': 'keebler1423',
 'time': '21/Jun/2019:15:45:40 -0700',
 'request': 'PUT /orchestrate/out-of-the-box/unleash/syndicate HTTP/1.1'},
{'host': '94.11.36.112',
 'user_name': 'klein8508',
 'time': '21/Jun/2019:15:45:41 -0700',
 'request': 'POST /enhance/solutions/bricks-and-clicks HTTP/1.1'},
{'host': '126.196.238.197',
 'user_name': 'gusikowski9864',
 'time': '21/Jun/2019:15:45:45 -0700',
 'request': 'DELETE /rich/reinvent HTTP/2.0'},
{'host': '103.247.168.212',
 'user_name': 'medhurst2732',
 'time': '21/Jun/2019:15:45:49 -0700',
 'request': 'HEAD /scale/global/leverage HTTP/1.0'},
{'host': '57.86.153.68',
```