Homework

Welcome to **Introduction to Data Science**!

We will have 7-8 assignments like this one. You can't learn technical subjects without hands-on practice, so homeworks are an important part of the course.

Homework 1

Homework 1 is a warming up assignment that will guide you familiar with the basics and settings for this class.

Jupyter notebooks

This webpage is called a Jupyter notebook. A notebook is a place to write programs and view their results.

Text cells

In a notebook, each rectangle containing text or code is called a *cell*.

Text cells (like this one) can be edited by double-clicking on them. They're written in a simple format called Markdown to add formatting and section headings. You don't need to learn Markdown, but you might want to.

After you edit a text cell, click the "run cell" button at the top that looks like ▶| to confirm any changes.

Code cells

Other cells contain code in the Python 3 language. Running a code cell will execute all of the code it contains.

To run the code in a code cell, first click on that cell to activate it. It'll be highlighted with a little green or blue rectangle. Next, either press ▶| or hold down the `shift` key and press `return` or `enter` .

Try running this cell:

In [ ]:
```
print("Hello, World!")
```

And this one:

In [ ]:
```
print("\N{WAVING HAND SIGN}, \N{EARTH GLOBE ASIA-AUSTRALIA}!")
```

The fundamental building block of Python code is an expression. Cells can contain multiple lines with multiple expressions. When you run a cell, the lines of code are executed in the order in which they appear. Every `print` expression prints a line. Run the next cell and notice the order of the output.

In [98]:
```
# your code here

print("First this line, ")
#print('First')
print("then the whole \N{EARTH GLOBE ASIA-AUSTRALIA},")
print("and then this one.")
```

```
First this line,
then the whole 🌏,
and then this one.
```

Question: Change the cell above so that it prints out:

```
First this line,
then the whole 🌏,
and then this one.
```

Question: Yuri Gagarin was the first person to travel through outer space. When he emerged from his capsule upon landing on Earth, he had the following conversation with a woman and girl who saw the landing:

```
The woman asked: "Can it be that you have come from outer space?"
Gagarin replied: "As a matter of fact, I have!"
```

The cell below contains unfinished code. Fill in the `...` s so that it prints out this conversation *exactly* as it appears above.

In [99]:

```
# your code here
woman_asking = 'The woman asked:'
woman_quote = '"Can it be that you have come from outer space?"'
gagarin_reply = 'Gagarin replied:'
gagarin_quote = '"As a matter of fact, I have!"'

print(woman_asking, woman_quote)
print(gagarin_reply, gagarin_quote)
```

```
The woman asked: "Can it be that you have come from outer space?"
Gagarin replied: "As a matter of fact, I have!"
```

You can also use the **print()** function to print strings and variables together, which will lead to a meaningfull string. For example:

In [10…

```
boys = 5
girls = 6
print ("There are ", boys, "boys in the class.")
print ("There are ", boys+girls, "students in the class.")
```

```
There are  5 boys in the class.
There are  11 students in the class.
```

Arithmetic

The line in the next cell subtracts. Its value is what you'd expect. Run it.

In [ ]:

```
3.25 - 1.5
```

Many basic arithmetic operations are built in to Python. The common operator that differs from typical math notation is `**` , which raises one number to the power of the other. So, `2**3` stands for 2^3 and evaluates to 8.

The order of operations is what you learned in elementary school, and Python also has parentheses. For example, compare the outputs of the cells below.

In [ ]:

```
1+6*5-6*3**2*2**3/4*7
```

In [75]:

```
1+(6*5-(6*3))**2*((2**3)/4*7)
```

Out[75]:

```
2017.0
```

In standard math notation, the first expression is

$$1 + 6 \times 5 - 6 \times 3^2 \times \frac{2^3}{4} \times 7,$$

while the second expression is

$$1 + (6 \times 5 - (6 \times 3))^2 \times (\frac{(2^3)}{4} \times 7).$$

Question: Write a Python expression in this next cell that's equal to $5 \times (3\frac{10}{11}) - 50\frac{1}{3} + 2^{.5 \times 22} - \frac{7}{33}$. That's five times three and ten elevenths, minus fifty and a third, plus two to the power of half 22, minus 7 33rds. By "$3\frac{10}{11}$" we mean $3 + \frac{10}{11}$, not $3 \times \frac{10}{11}$.

In [76]:
```
## your code here
5*(3+10/11)-50-1/3+2**(.5*22)-7/33
```

Out[76]:
```
2017.0
```

Question: In the next cell, write a Python expression equivalent to this math expression:

$$1 - \frac{2^2}{2^{20}}$$

In [77]:
```
## your code here
1-(2**2)/(2**20)
```

Out[77]:
```
0.9999961853027344
```

Question: A famous fact in mathematics is that

$$(1 - \frac{1}{n})^{-n}$$

gets very close to the number e, which is roughly 2.718, when n is large. Verify that it gets closer to e as n gets larger by writing this expression in Python and trying different values for n.

In [78]:
```
## your code here
## Hint: you can use n = 10, 100, 1000,... and print your results.
a = 10;b = 0;
for i in range(2,20):
    a = 10**i
    b = (1-1/a)**(-a)
    print(a,b)


##当a>10**12就会出现溢出,但是在一定范围内结果都稳定在2.71828
```

```
100 2.7319990264290284
1000 2.719642216442853
10000 2.71841775501015
100000 2.7182954199804055
1000000 2.7182831876793716
10000000 2.7182819629423656
100000000 2.7182818557091655
1000000000 2.7182817529399292
10000000000 2.7182820535066914
100000000000 2.7182820533850918
1000000000000 2.7182216960543433
10000000000000 2.7191271965359936
100000000000000 2.7161100340870363
1000000000000000 2.7161100340870363
10000000000000000 3.0350352065492636
100000000000000000 1.0
1000000000000000000 1.0
10000000000000000000 1.0
```

Question: The length of three sides of a triangle is a, b and c, try to calculate the area S of the triangle.

In [10⋯
```
# your code here
import math
a = 17
b = 16
c = 30

# your code here
# first write the expression of S in terms of a, b and c
p = (a+b+c)/2
```

```
S = math.sqrt(p*(p-a)*(p-b)*(p-c))
# then print
print ("The area of the triangle is ", S)
```

```
The area of the triangle is  103.0506550197523
```

Application: A physics experiment

On the Apollo 15 mission to the Moon, astronaut David Scott famously replicated Galileo's physics experiment in which he showed that gravity accelerates objects of different mass at the same rate. Because there is no air resistance for a falling object on the surface of the Moon, even two objects with very different masses and densities should fall at the same rate. David Scott compared a feather and a hammer.

You can run the following cell to watch a video of the experiment.

In [80]:
```
from IPython.display import Video
Video("./videoApollo.mp4")
```

Out[80]:

0:00 / 0:47

Here's the transcript of the video:

00:01 Scott: Well, in my left hand, I have a feather; in my right hand, a hammer. And I guess one of the reasons we got here today was because of a gentleman named Galileo, a long time ago, who made a rather significant discovery about falling objects in gravity fields. And we thought where would be a better place to confirm his findings than on the Moon. And so we thought we'd try it here for you. The feather happens to be, appropriately, a falcon feather for our Falcon. And I'll drop the two of them here and, hopefully, they'll hit the ground at the same time.

00:40 Scott: How about that!

00:42 Allen: How about that! (Applause in Houston)

00:43 Scott: Which proves that Mr. Galileo was correct in his findings.

Newton's Law. Using this footage, we can also attempt to confirm another famous bit of physics: Newton's law of universal gravitation. Newton's laws predict that any object dropped near the surface of the Moon should fall

$$\frac{1}{2} G \frac{M}{R^2} t^2 \text{ meters}$$

after t seconds, where G is a universal constant, M is the moon's mass in kilograms, and R is the moon's radius in meters. So if we know G, M, and R, then Newton's laws let us predict how far an object will fall over any amount of time.

To verify the accuracy of this law, we will calculate the difference between the predicted distance the hammer drops and the actual distance. (If they are different, it might be because Newton's laws are wrong, or because our measurements are imprecise, or because there are other factors affecting the hammer for which we haven't accounted.)

Someone studied the video and estimated that the hammer was dropped 113 cm from the surface. Counting frames in the video, the hammer falls for 1.2 seconds (36 frames).

Question: Complete the code in the next cell to fill in the *data* from the experiment.

In [81]:
```
# t, the duration of the fall in the experiment, in seconds.
# Fill this in.
time = 1.2

# The estimated distance the hammer actually fell, in meters.
# Fill this in.
estimated_distance_m = 1.13
```

Question: Now, complete the code in the next cell to compute the difference between the predicted and estimated distances (in meters) that the hammer fell in this experiment.

This just means translating the formula above ($\frac{1}{2}G\frac{M}{R^2}t^2$) into Python code. You'll have to replace each variable in the math formula with the name we gave that number in Python code.

In [82]:
```
# First, we've written down the values of the 3 universal
# constants that show up in Newton's formula.

# G, the universal constant measuring the strength of gravity.
gravity_constant = 6.674 * 10**-11

# M, the moon's mass, in kilograms.
moon_mass_kg = 7.34767309 * 10**22

# R, the radius of the moon, in meters.
moon_radius_m = 1.737 * 10**6

# The distance the hammer should have fallen over the
# duration of the fall, in meters, according to Newton's
# law of gravity.  The text above describes the formula
# for this distance given by Newton's law.
# **YOU FILL THIS PART IN.**
predicted_distance_m = .5*gravity_constant*moon_mass_kg*time**2/moon_radius_m**2

# Here we've computed the difference between the predicted
# fall distance and the distance we actually measured.
# If you've filled in the above code, this should just work.
difference = predicted_distance_m - estimated_distance_m
difference
```

Out[82]:
```
0.040223694659304865
```

Calling functions

The most common way to combine or manipulate values in Python is by calling functions. Python comes with many built-in functions that perform common operations.

For example, the `abs` function takes a single number as its argument and returns the absolute value of that number. The absolute value of a number is its distance from 0 on the number line, so `abs(5)` is 5 and `abs(-5)` is also 5.

In [83]:
```
abs(5)
```

Out[83]:
```
5
```

In [84]:
```
abs(-5)
```

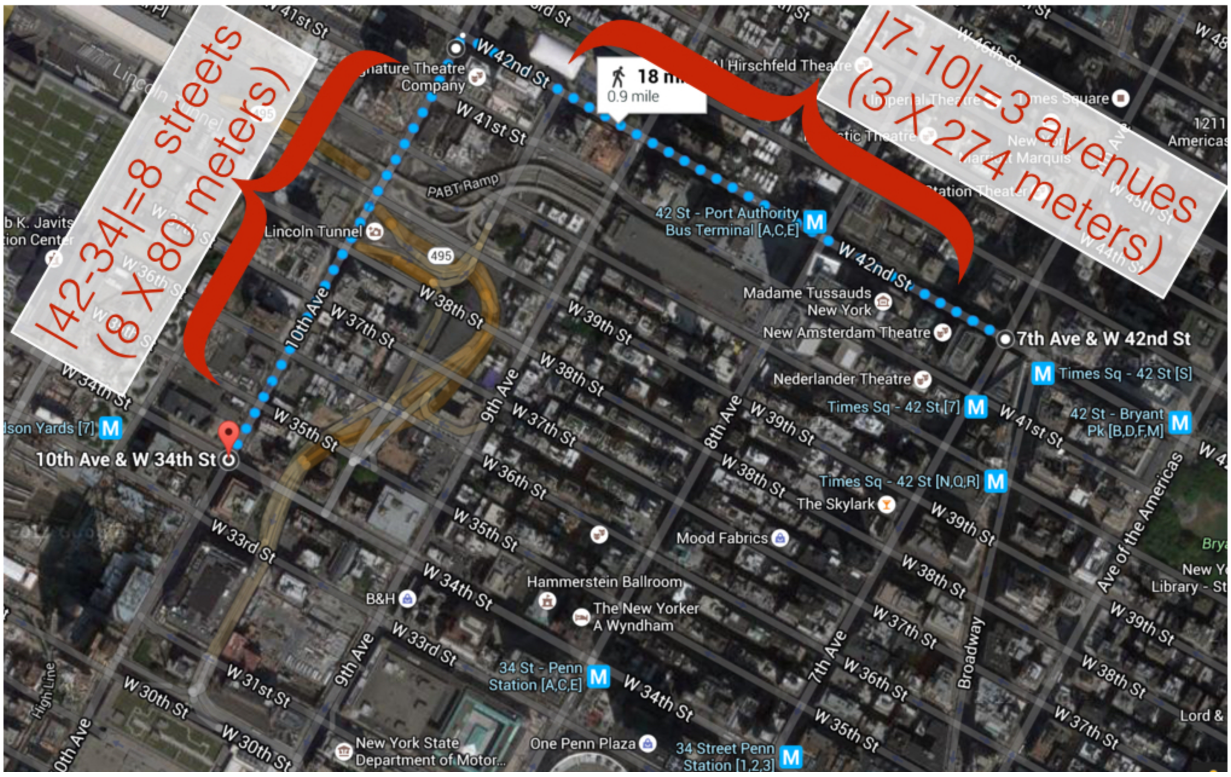
Out[84]:
```
5
```

Application: Computing walking distances

Chunhua is on the corner of 7th Avenue and 42nd Street in Midtown Manhattan, and she wants to know far she'd have to walk to get to Gramercy School on the corner of 10th Avenue and 34th Street.

She can't cut across blocks diagonally, since there are buildings in the way. She has to walk along the sidewalks. Using the map below, she sees she'd have to walk 3 avenues (long blocks) and 8 streets (short blocks). In terms of the given numbers, she computed 3 as the difference between 7 and 10, *in absolute value*, and 8 similarly.

Chunhua also knows that blocks in Manhattan are all about 80m by 274m (avenues are farther apart than streets). So in total, she'd have to walk $(80 \times |42 - 34| + 274 \times |7 - 10|)$ meters to get to the park.



Question: Finish the line `num_avenues_away = ...` in the next cell so that the cell calculates the distance Chunhua must walk and gives it the name `manhattan_distance` . Everything else has been filled in for you. **Use the `abs` function.**

In [85]:

```
# Here's the number of streets away:
num_streets_away = abs(42-34)

# Compute the number of avenues away in a similar way:
num_avenues_away = abs(7-10)

street_length_m = 80
avenue_length_m = 274

# Now we compute the total distance Chunhua must walk.
manhattan_distance = street_length_m*num_streets_away + avenue_length_m*num_avenues_away

# We've included this line so that you see the distance
# you've computed when you run this cell.  You don't need
# to change it, but you can if you want.
manhattan_distance
```

Out[85]:

```
1462
```

In [ ]:

String Methods

Strings can be transformed using **methods**, which are functions that involve an existing string and some other arguments. One example is the `replace` method, which replaces all instances of some part of a string with some alternative.

A method is invoked on a string by placing a `.` after the string value, then the name of the method, and finally parentheses containing the arguments. Here's a sketch, where the `<` and `>` symbols aren't part of the syntax; they just mark the boundaries of sub-expressions.

```
<expression that evaluates to a string>.<method name>(<argument>, <argument>, ...)
```

Try to predict the output of these examples, then execute them.

In [86]:
```
# Replace one letter
'Hello'.replace('o', 'a')
```

Out[86]:
```
'Hella'
```

In [87]:
```
# Replace a sequence of letters, which appears twice
'hitchhiker'.replace('hi', 'ma')
```

Out[87]:
```
'matchmaker'
```

Once a name is bound to a string value, methods can be invoked on that name as well. The name doesn't change in this case, so a new name is needed to capture the result.

In [88]:
```
sharp = 'edged'
hot = sharp.replace('ed', 'ma')##attention! it will change all 'ed'
print('sharp:', sharp)
print('hot:', hot)
```

```
sharp: edged
hot: magma
```

You can call functions on the results of other functions. For example,

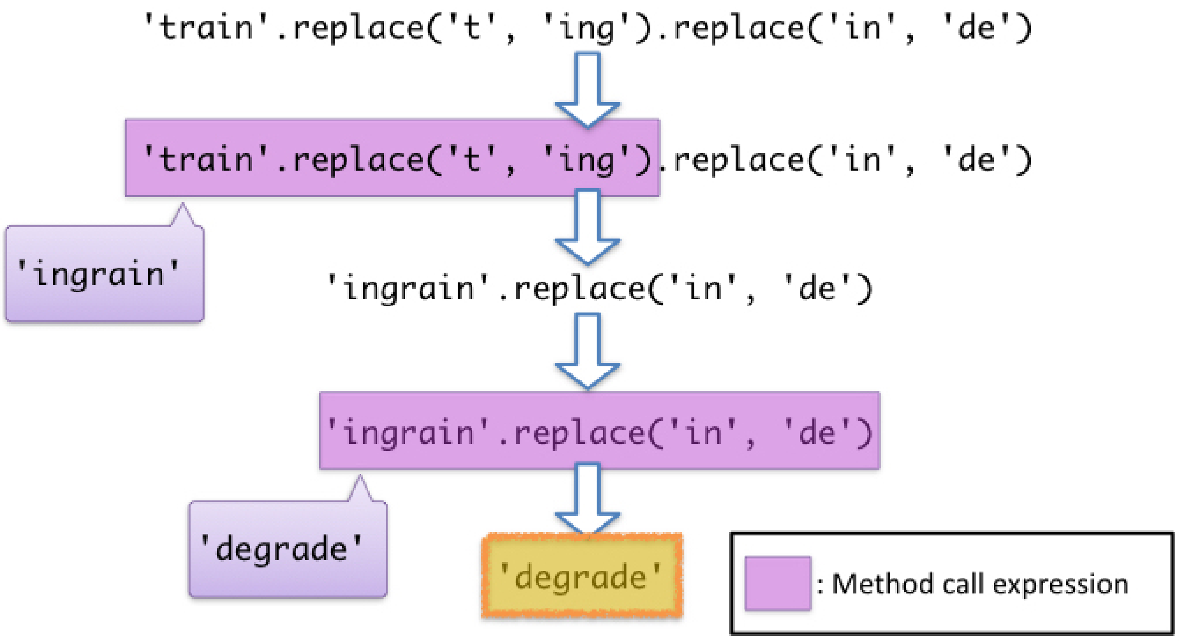
```
max(abs(-5), abs(3))
```

has value 5. Similarly, you can invoke methods on the results of other method (or function) calls.

In [89]:
```
# Calling replace on the output of another call to
# replace
'train'.replace('t', 'ing').replace('in', 'de')
```

Out[89]:
```
'degrade'
```

Here's a picture of how Python evaluates a "chained" method call like that:


'train'.replace('t', 'ing').replace('in', 'de')
'train'.replace('t', 'ing').replace('in', 'de')
'ingrain'
'ingrain'.replace('in', 'de')
'ingrain'.replace('in', 'de')
'degrade'
'degrade'
: Method call expression

Question: Assign strings to the names `you` and `this` so that the final expression evaluates to a 10-letter English word with three double letters in a row.

Hint: After you guess at some values for `you` and `this`, it's helpful to see the value of the variable `the`. Try printing the value of `the` by adding a line like this:

```
print(the)
```

Hint 2: If you're stuck. **Only for this question, you can ask for help in our QQ group!**

In [90]:
```
# your code here
you = 'bee'
this = 'acca'
a = 'beeper'
the = a.replace('p', you)
print(the)
the = the.replace('bee', this)#remember to store 'the' value
print(the)
```

```
beebeeer
accaaccaer
```

Importing code

> What has been will be again,
> what has been done will be done again;
> there is nothing new under the sun.

Most programming involves work that is very similar to work that has been done before. Since writing code is time-consuming, it's good to rely on others' published code when you can. Rather than copy-pasting, Python allows us to **import** other code, creating a **module** that contains all of the names created by that code.

Python includes many useful modules that are just an `import` away. We'll look at the `math` module as a first example. The `math` module is extremely useful in computing mathematical expressions in Python.

Suppose we want to very accurately compute the area of a circle with radius 5 meters. For that, we need the constant π, which is roughly 3.14. Conveniently, the `math` module has `pi` defined for us:

In [91]:

```
import math
radius = 5
area_of_circle = radius**2 * math.pi
area_of_circle
```

```
Out[91]: 78.53981633974483
```

`pi` is defined inside `math`, and the way that we access names that are inside modules is by writing the module's name, then a dot, then the name of the thing we want:

```
<module name>.<name>
```

In order to use a module at all, we must first write the statement `import <module name>`. That statement creates a module object with things like `pi` in it and then assigns the name `math` to that module. Above we have done that for `math`.

Question: `math` also provides the name `e` for the base of the natural logarithm, which is roughly 2.71. Compute $e^{\pi} - \pi$, giving it the name `near_twenty`.

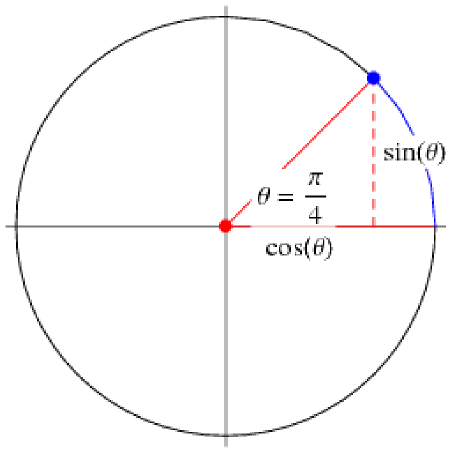
```
In  [92]:
import math
e = math.e
pi = math.pi
near_twenty = e**pi-pi
near_twenty
```

```
Out[92]: 19.999099979189447
```

Importing functions

Modules can provide other named things, including **functions**. For example, `math` provides the name `sin` for the sine function. Having imported `math` already, we can write `math.sin(3)` to compute the sine of 3. (Note that this sine function considers its argument to be in radians, not degrees. 180 degrees are equivalent to π radians.)

Question: A $\frac{\pi}{4}$-radian (45-degree) angle forms a right triangle with equal base and height, pictured below. If the hypotenuse (the radius of the circle in the picture) is 1, then the height is $\sin(\frac{\pi}{4})$. Compute that using `sin` and `pi` from the `math` module. Give the result the name `sine_of_pi_over_four`.



(Source: Wolfram MathWorld)

```
In  [93]:
sine_of_pi_over_four = math.sin(math.pi/4)
#print(math.sqrt(1/2))
sine_of_pi_over_four
```

```
Out[93]: 0.7071067811865476
```

For your reference, here are some more examples of functions from the `math` module.

Note how different methods take in different number of arguments. Often, the documentation of the module will provide information on how many arguments is required for each method.

In [94]:
```
# Calculating factorials.#阶乘
math.factorial(5)
```

Out[94]:
```
120
```

In [95]:
```
# Calculating logarithms (the logarithm of 8 in base 2).
# The result is 3 because 2 to the power of 3 is 8.
math.log(8, 2)
```

Out[95]:
```
3.0
```

In [96]:
```
# Calculating square roots.
math.sqrt(5)
```

Out[96]:
```
2.23606797749979
```

In [97]:
```
# Calculating cosines.
math.cos(math.pi)
```

Out[97]:
```
-1.0
```

In [ ]:
```

```