

Python For Microeconomics by Professor Khazra

In this notebook, we will learn to code with Python and solve microeconomic optimization models. We will also learn to play with parameters and use comparative statics to better understand the theories and assumptions.

In this notebook, you will experiment with the following models:

1. [Utility Maximization](#)
2. [Profit Maximization](#)
3. [Game Theory](#)

Before we start let's read some jokes and have fun with coding!

```
In [1]: ! pip install -q pyjokes
```

```
In [2]: import pyjokes

joke = pyjokes.get_joke()
print(joke)
```

Why do programmers prefer dark mode? Because light attracts bugs.

Section 0: House Keeping

In Python, a library is a collection of modules that provide additional functionality to the core language. Libraries can be used to extend the capabilities of Python in areas such as economics, statistics, mathematics, data processing, image and audio processing, and more. To use a library in a Python script, you need to import it.

There are two main ways to import libraries in Python:

1. Importing the entire library: This is done using the import statement, followed by the name of the library. For example:

```
import numpy
```

2. Importing specific parts of a library: This is done using the `from ... import ...` statement, followed by the name of the library and the specific parts to be imported. For example:

```
from numpy import array
```

In the above example, only the `array` function from the `numpy` library is imported.

Both of these methods have their own advantages and disadvantages, and the choice of which one to use depends on the specific requirements of the project.

In addition to the standard libraries that come with Python, there are also many third-party libraries available that you can use to extend the functionality of your Python scripts. To use these libraries, you first need to install them on your computer, and then you can import them in your Python scripts as described above.

```
In [3]: # We need to import some libraries first:

import sympy as sp #for symbolic mathematics
import numpy as np #for working with data
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline
```

The libraries that we will use in this notebook are the following:

1. **`sympy` : is a Python library for symbolic mathematics.**

- Main documentation:
 - <https://www.sympy.org/en/index.html>
- A cheatsheet:
 - <http://daabzlatex.s3.amazonaws.com/9065616cce623384fe5394eddf4c5>

2. **`numpy` : is a Python library for working with data.**

- Main documentation:
 - <https://numpy.org>
- Cheatsheet:
 - https://assets.datacamp.com/blog_assets/Numpy_Python_Cheat_Sheet.pdf
 - https://www.w3schools.com/python/numpy/numpy_intro.asp

3. **`matplotlib.pyplot` : is a Python library to make plots and show plots on**

your screen.

- Main documentation
 - https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.html

4. `mpl_toolkits.mplot3d` : is a Python library to make 3D plots.

- A tutorial:
 - https://matplotlib.org/2.0.2/mpl_toolkits/mplot3d/tutorial.html

Libraries vs Functions

Python has many many libraries, and the number is growing every day. The software is open sourced and many people contribute to it. In this notebook, we use a few popular libraries such as Numpy, Sympy, and Matplotlib. In these libraries, there are some functions that we will call to perform the operations that we need.

When we call these functions and methods, we should tell Python which library to use first. It's like checking out books from an actual library! If you are borrowing a book online, you should first log in to your chosen library and then check out the book you need. The same is valid for Python libraries. Pick your library first, and then check out the function!

For example, we use the `symbols` method from the `sympy` library. When you want to use this function, you should write `sp.symbols`, and as you remember `sp` is the nickname we gave to the sympy library just because we are lazy to type sympy each time we want to check out a method or a function.

Section 1: Utility Maximization

We want to define our utility function and maximize it. Below are the steps we want to follow:

1. calculate MRS
2. define $MRS = P_x/P_y$ (the tangency condition)
3. define the budget constraint
4. finally solve the two equations- two unknowns system for X^* and Y^* .

Before we dive into the maximization problem, let's do some plotting.

This code creates a 3D surface plot of a given utility function, $U(X, Y)$. The Cobb-Douglas utility function is defined as $U = 30 * \text{np.sqrt}(X) * \text{np.sqrt}(Y) = 30X^{0.5}Y^{0.5}$, where X and Y are inputs.

Here are the steps of the code:

- We first create a mesh grid for the X, Y space. This is done using the `linspace` function from the NumPy library, which creates an array of equally spaced values over a specified range. The `linspace` function creates 100 equally spaced values between 0 and 10 for both X and Y , and stores them in the `x_mesh` and `Y_mesh` arrays, respectively.
- The `meshgrid` function from NumPy is then used to create a grid of all X, Y pairs. The X and Y arrays are the inputs to the `meshgrid` function, which returns two arrays of the same shape, X and Y , which represent the `X` and `Y` values of all the points in the mesh grid.
- Now we define our Cobb-Douglas utility function `U = 30 * np.sqrt(X) * np.sqrt(Y)`. This is a simple mathematical formula that takes the input X and Y arrays, calculates the square root of each value, multiplies the two square roots together, and multiplies the result by 30. This formula is just an example, you can use other functions or even have more than two goods!!
- We need to tell Python that we want a 3D plot and also we should specify the length of each grid. The "111" means 1 by 1 grids in the first subplot.
- We then create a 3D plot of the utility function using the `plot_surface` function from the Matplotlib library. This function takes three arguments: the X, Y , and Z arrays, which represent the X and Y values of the points in the mesh grid and the corresponding Z values of the utility function.
- At last, we want to set the labels for the X, Y , and Z axes using the `set_xlabel`, `set_ylabel`, and `set_zlabel` functions. It also sets the limits for the X and Y axes using the `set_xlim` and `set_ylim` functions.
- The color map is set to `viridis`, which is a perceptual color map that is color-blind friendly.
- `tight_layout` is used to adjust the subplot parameters for a better fit in the figure.

Finally, we show the plot using the `plt.show` function from the Matplotlib library.

```
In [4]: #define a mesh grid for the X, Y space
x_mesh = np.linspace(0.0, 10, 100)
Y_mesh = np.linspace(0.0, 10, 100)
X,Y = np.meshgrid(x_mesh, Y_mesh)

#define the utility function
```

```

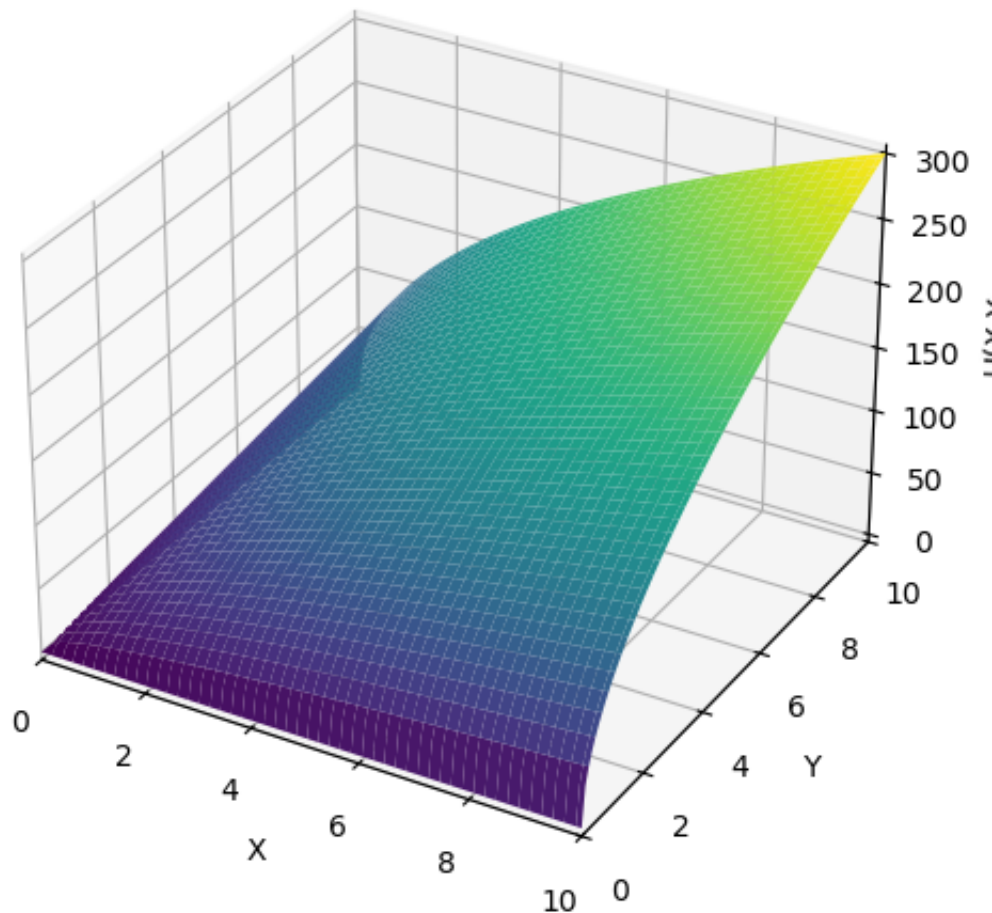
U = 30 * np.sqrt(X) * np.sqrt(Y)

#simple plotting
fig = plt.figure()
ax = fig.add_subplot(111,projection='3d')

ax.plot_surface(X, Y, U, cmap='viridis')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('U(X,Y)')
ax.set_xlim([0,10])
ax.set_ylim([0,10])

plt.tight_layout()
plt.show()

```



```

In [5]: # rewrite the above code as a reusable function:
def plot3d(A,b,c):
    x_mesh = np.linspace(0.0, 10, 100)
    Y_mesh = np.linspace(0.0, 10, 100)
    X,Y = np.meshgrid(x_mesh, Y_mesh)

    #define the utility function

```

```

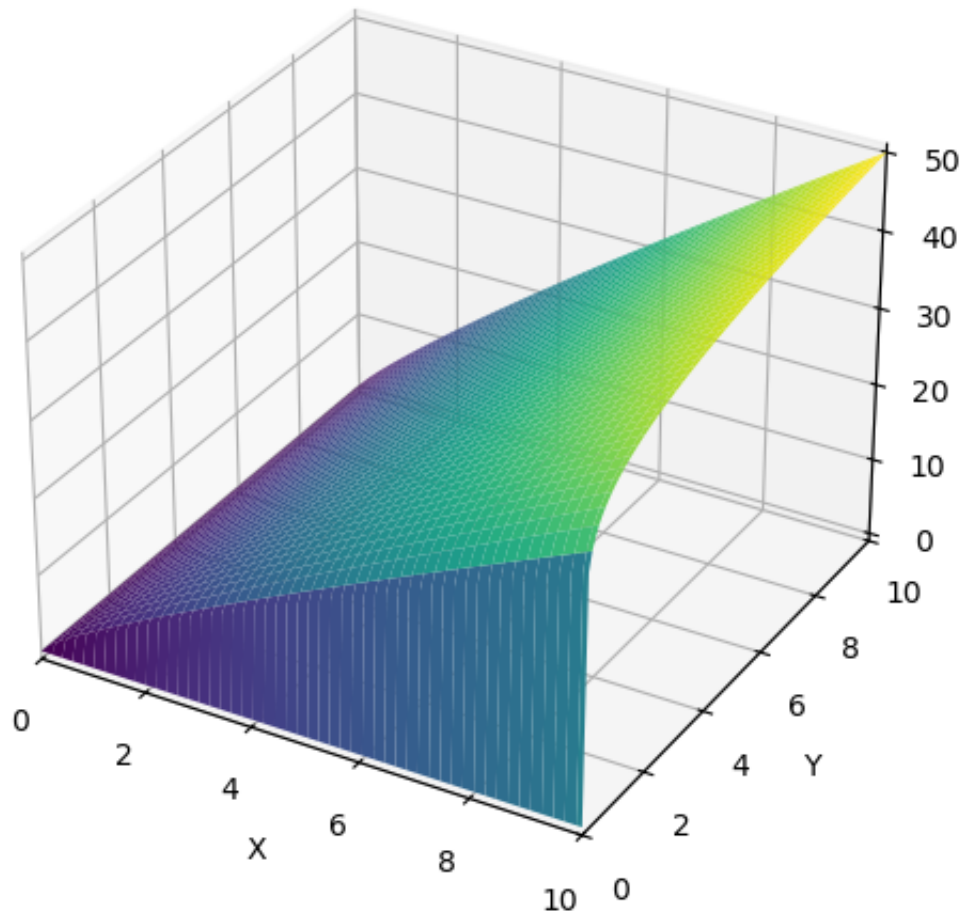
U = A * X**b * Y**c

#simple plotting
fig = plt.figure()
ax = fig.add_subplot(111,projection='3d')

ax.plot_surface(X, Y, U, cmap='viridis')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('U(X,Y)')
ax.set_xlim([0,10])
ax.set_ylim([0,10])
plt.tight_layout()
return plt.show()

```

In [6]: `plot3d(5,0.9,0.1)`



The code below is defining a function for computing the Marginal Rate of Substitution (MRS). The MRS measures the rate at which a consumer is willing to give up one good in exchange for another, while keeping their level of utility constant.

We first want to define symbols for X and Y, indicating that they will be used as variables in the rest of the code.

Next, the function MRS is defined, which takes in a utility function as input, `U`, and calculates the `MRS` as the ratio of the marginal utilities of X and Y. The marginal utility of a good is the change in utility that results from a small increase in the consumption of that good.

Finally, a specific example of a utility function, `U`, is defined as the square root of X times the square root of Y, and the MRS is calculated by calling the MRS function on this utility function.

```
In [7]: X,Y= sp.symbols('X', real = True), sp.symbols('Y', real = True)

def Myfunction(U):
    MU_X = sp.diff(U, X)
    MU_Y = sp.diff(U, Y)
    myvalue = MU_X/MU_Y
    return myvalue

U = X**0.5 * Y**0.5
myvalue(U)
```

```
Out[7]: 1.0Y1.0
        X1.0
```

Here are the steps:

1. Find the Marginal Rate of Substitution: $MRS_{X,Y} = \frac{MU_X}{MU_Y}$
2. Set up the tangency equation: $MRS_{X,Y} = \frac{P_X}{P_Y}$
3. Set up the budget constraint: $Income = P_X * X + P_Y * Y$
4. Solve the tangency equation and the budget constraint to find the optimal bundle of X and Y

We finished step 1, let's move to step 2 now.

In the code below, the equality `Tangency = sp.Eq(MRS(U), P_X/P_Y)` sets the `MRS` equal to the ratio of the price of X and Y. `sp` is calling the `sympy` library and then asks for the function `Eq` which is equation. It reads like this: from the `sympy` library, get the `Eq` function and create an equation that sets my MRS equal to `P_X/P_Y`

```
In [8]: # Step 2: set it equal to P_X/P_Y
P_X = 1 #these numbers are examples, you can change these numbers to t
P_Y = 1
Tangency = sp.Eq(MRS(U), P_X/P_Y)
```

Tangency

Out[8]: $\frac{1.0Y^{1.0}}{X^{1.0}} = 1.0$

```
In [9]: # Step 3: set up the budget constraint equation
I = 1000
Budget_Constraint = sp.Eq(P_X*X + P_Y*Y, I)
Budget_Constraint
```

Out[9]: $X + Y = 1000$

Below, we use `sp.solve` function from the `SymPy` library to solve for X and Y in the given tangency condition and budget constraint.

The first argument to the `sp.solve` function is a tuple of equations. In this case, it contains `Tangency` and `Budget_Constraint`. The second argument is a tuple of variables, which in this case is `(X, Y)`.

The `sp.solve` function is used to find the solution of the equations represented by `Tangency` and `Budget_Constraint` with respect to the variables `X` and `Y`. This can be interpreted as finding the values of X and Y such that the tangency condition and budget constraint are satisfied.

```
In [10]: # Step 4: solve the MRS equation and the budget constraint and find X*
sp.solve((Tangency, Budget_Constraint), (X, Y))
```

Out[10]: {X: 500.0000000000000, Y: 500.0000000000000}

Now let's look at everything in one cell and play with the functions.

Change the utility function to the following and give me the X^* and Y^*

1. $U = 10X^{1/3}Y^{2/3}$
2. $U = 3X + 4Y$
3. $U = \log(X) + \log(Y)$
4. $U = \min(X, Y)$

```
In [11]: X,Y= sp.symbols('X', real = True), sp.symbols('Y', real = True)

def MRS(U):
    MU_X = sp.diff(U, X)
    MU_Y = sp.diff(U, Y)
    mrs = MU_X/MU_Y
```

```

    return mrs

U = X**0.5 * Y**0.5
P_X = 1
P_Y = 1
I = 1000

Tangency = sp.Eq(MRS(U), P_X/P_Y)
Tangency

Budget_Constraint = sp.Eq(P_X*X + P_Y*Y, I)
Budget_Constraint

sp.solve((Tangency, Budget_Constraint), (X, Y))

```

Out[11]: {X: 500.0000000000000, Y: 500.0000000000000}

Take home exercise:

Can you define a Cobb-Douglas function that takes the elasticities and returns the utility function?

```

In [12]: #answer:
# def Cobb_Douglas(A, alpha, beta):
#     U = A* X**alpha * Y**beta
#     return U

```

Section 2: Profit Maximization

1. Define TR function
2. Define TC function
3. Take the derivative
4. Plot the curves
5. Experiment and have fun!!

```

In [13]: # define the total revenue function
P = 5
Q = 1000000
TR = P * Q
TR

print("The firms's revenue is", TR, "dollars")

# change P and Q and experiment with the TR function

```

The firms's revenue is 5000000 dollars

```
In [14]: # or better, uncomment the following code and write a TR function! fan
# def TR(P, Q):
#     rev = P*Q
#     return rev
```

Let's now define our demand function and total revenue and then take the first derivative to get marginal revenue. We will experiment with linear and non-linear functions.

```
In [15]: # define inverse demand
Q = sp.symbols('Q') # define Q as a symbol to then use it in the demand
P = 50 - Q
P
```

Out[15]: $50 - Q$

```
In [16]: # define total revenue
TR = P*Q
TR
```

Out[16]: $Q(50 - Q)$

```
In [17]: # find marginal revenue
MR = sp.diff(TR, Q)
MR
```

Out[17]: $50 - 2Q$

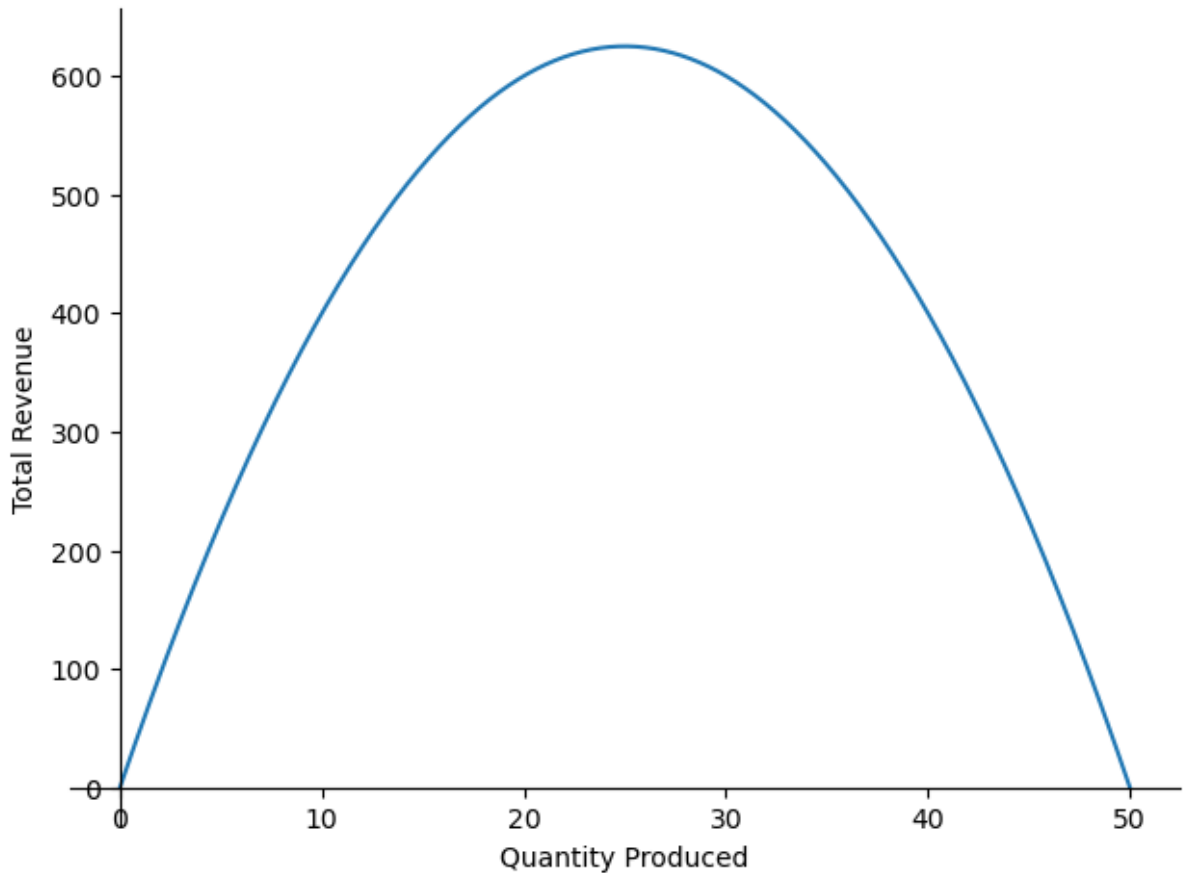
```
In [18]: max_TR = sp.solve(sp.diff(TR, Q), Q) # find the Max of the TR function
max_TR
```

Out[18]: [25]

```
In [19]: sp.solve(TR, Q) #find the limits of Q for the plot. When does the TR f
```

Out[19]: [0, 50]

```
In [20]: sp.plot(TR, (Q,0,50), ylabel= "Total Revenue", xlabel = "Quantity Prod
```



Out[20]: <sympy.plotting.backends.matplotlibbackend.matplotlib.MatplotlibBackend at 0x7fae46fdf6b0>

Let's now put everything together and create a more generalizable version to be applied to ANY demand function.

```
In [21]: Q = sp.symbols('Q')
P = sp.Function('P') #sp is what we called the sympy library, and Func
# the sp.Function tells Python that 'P' is going to be a function in m

# This is a better way to define a marginal revenue function
def MR(P):
    TR = P*Q
    marginal_revenue = sp.diff(TR, Q)
    return marginal_revenue
```

Let's test out our "MR" (marginal revenue) function, uncomment each line and check the MR for each demand:

1. $P = 300 - Q$
2. $P = \ln(Q)$
3. $P = 5000 - \sqrt{Q}$
4. $P = 600 - (Q - 10)^{1/3}$

```
In [22]: # practice: uncomment whichever you want to test!
```

```
#MR(300-Q)
#MR(ln(Q))
#MR(5000-Q**(1/2)) #note that ** means exponentiation in Python not ^
#MR(600-(Q-10)**(1/3))
```

Just beautiful!

We know how to code TR functions and drive MR. Let's now do more. Let's define cost functions and then profit functions and maximize the profits! Yooohoo! I'm going to first define the cost function and then define the profit function. You know the rest of the story.

```
In [23]: # Define the Profit function

TC = sp.Function('TC') #tell Python a function is coming its way!

def Profit(P,TC):
    TR = P*Q
    pi = TR-TC
    return pi
```

Now let's test our profit function. Here is what you see in the next block:

1. Lay out the function using the 'Profit' function defined above
2. Take the first derivative using the `diff` method
3. Solve for Q^* using the `solve` method

Here is the profit function in mathematical form:

$$\pi = TR - TC = PQ - TC(Q)$$

If the inverse demand function and the cost function are the following;
 $P = 500 - Q$ and $TC(Q) = Q^3 - 21Q^2 + 400Q$, then profit will be:

$$\pi = (500Q - Q^2) - (Q^3 - 21Q^2 + 400Q) = -Q^3 + 20Q^2 + 100Q$$

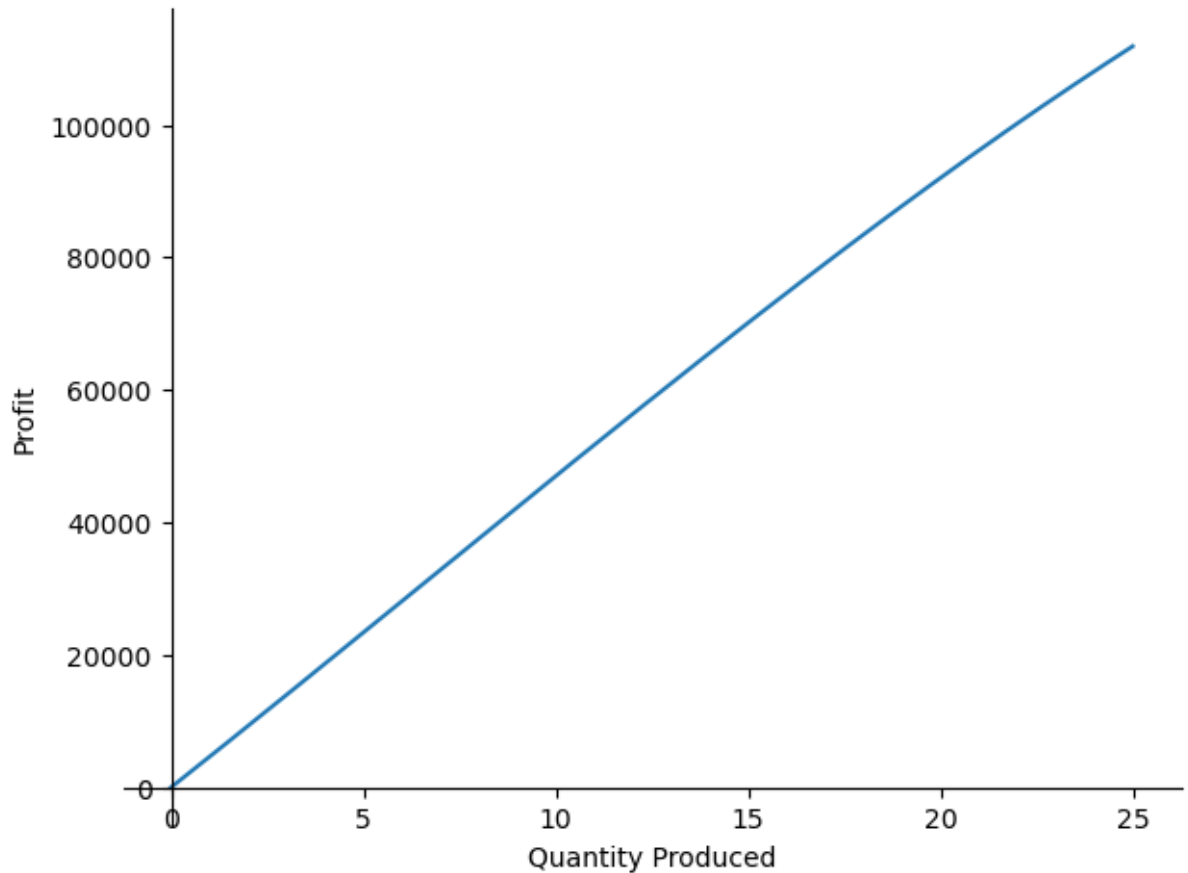
```
In [24]: pi = Profit(5000-Q, Q**3-21*Q**2+400*Q) #notice that ** means 'to the
sp.solve(sp.diff(pi,Q),Q) #this gives you the profit maximizing quanti
```

```
Out[24]: [20/3 - 10*sqrt(142)/3, 20/3 + 10*sqrt(142)/3]
```

```
In [25]: float(20/3 + 10*np.sqrt(142)/3)
```

```
Out[25]: 46.387917626043276
```

```
In [26]: sp.plot(pi, (Q,0,25), ylabel = "Profit", xlabel = "Quantity Produced")
```



Out[26]: <sympy.plotting.backends.matplotlibbackend.matplotlib.MatplotlibBackend at 0x7fae36b6f230>

POLL

In [27]: *# can you find the profit maximizing quantity if demand goes up to $P =$*

Section 3: Game Theory!

This section is for all the geeks! Oh, I love this part so much.

We want to solve for pure strategy and mixed strategy Nash equilibria with just a few lines of code! Let's do this.

Here is the breakdown:

1. Pure strategy NE
2. Mixed strategy NE

Some more resoueces for you:

https://nashpy.readthedocs.io/_/downloads/en/stable/pdf/

<https://github.com/Nikoleta-v3/Game-Theory-and-Python/blob/master/1.%20Introduction.ipynb>

<https://towardsdatascience.com/game-theory-666abe63215e>

<https://www.theoj.org/jose-papers/jose.00078/10.21105.jose.00078.pdf>

<https://towardsdatascience.com/an-introduction-to-game-theory-using-python-358c63e36e02>

In the next block of code, we will do the following:

1. install and then import the `pynash` library
2. Define a 2-player, 2-strategies simultaneous game (normal form)
3. Find the pure strategy Nash Equilibrium

```
In [28]: # you may need to install this package first, if so, uncomment the line
import sys

!{sys.executable} -m pip install nashpy

#this may work too
#!pip install nashpy
```

Collecting nashpy

Downloading nashpy-0.0.43-py3-none-any.whl.metadata (6.7 kB)

Requirement already satisfied: numpy>=1.21.0 in /opt/conda/lib/python3.12/site-packages (from nashpy) (2.2.6)

Requirement already satisfied: scipy>=0.19.0 in /opt/conda/lib/python3.12/site-packages (from nashpy) (1.16.0)

Requirement already satisfied: networkx>=3.0.0 in /opt/conda/lib/python3.12/site-packages (from nashpy) (3.5)

Collecting deprecated>=1.2.14 (from nashpy)

Downloading deprecated-1.3.1-py2.py3-none-any.whl.metadata (5.9 kB)

Requirement already satisfied: wrapt<3,>=1.10 in /opt/conda/lib/python3.12/site-packages (from deprecated>=1.2.14->nashpy) (1.17.3)

Downloading nashpy-0.0.43-py3-none-any.whl (32 kB)

Downloading deprecated-1.3.1-py2.py3-none-any.whl (11 kB)

Installing collected packages: deprecated, nashpy

Successfully installed deprecated 1.3.1 nashpy 0.0.43

```
In [29]: # 1. import the library and give it a nickname
import nashpy as nash
```

The code below defines a two-player, two-strategy (2x2) game in game theory using the `nashpy` library. The game consists of two players, Player 1 and Player

2, who each have two strategies, A and B. The payoffs for each player for each strategy are represented as a 2x2 matrix. In this case, both players have the same payoffs.

The `nashpy` library is used to find the Nash Equilibrium of the game, which is a pair of strategies, one for each player, such that neither player can increase their payoff by unilaterally changing their strategy. In other words, the Nash Equilibrium is a stable state in which neither player has an incentive to change their strategy.

The `nash.Game` method is used to create a game object, which allows us to use the functions in the `nashpy` library to analyze the game.

```
In [30]: # 2. define your 2 by 2 game
Player1 = np.array([[ -1, -3], [0, -2]]) #[-1,-3] are player1's payoff for
                                         #[0,-2] are player 1's payoff for
Player2 = np.array([[ -1, -3], [0, -2]])
mygame = nash.Game(Player1, Player2) #nash is what we called the nashpy

mygame
#can you draw the normal form?
```

Out[30]: Bi matrix game with payoff matrices:

```
Row player:
[[-1 -3]
 [ 0 -2]]
```

```
Column player:
[[-1 -3]
 [ 0 -2]]
```

```
In [31]: # 3. find the equilirbium!
equilibria = mygame.support_enumeration()
for eq in equilibria:
    print(eq)
```

```
(array([0., 1.]), array([1., 0.]))
```

Let's try to see if we can find any mixed strategy nash equilibria!

In the output, you can see all equilirbia, there are two pure strategy NEs and one mixed strategy NE. How awesome is this?

The final line simply prints the game object, but the results of the code would be the Nash Equilibria of the game, if any exist. In this particular game, there are two Nash Equilibria: (A, A) and (B, B), meaning that both players playing strategy A or both players playing strategy B are both Nash Equilibria.

```
In [32]: # define your 2 by 2 game (two strategies: A and B, two players: player 1 and player 2)
Player1 = np.array([[4,1],[0,3]]) # 4 and 1 are player 1's outcomes for strategies A and B
Player2 = np.array([[3,0],[1,4]])
newgame = nash.Game(Player1,Player2) #nash is what we called the nashpy library

newgame
#can you draw the normal form?
```

Out[32]: Bi matrix game with payoff matrices:

```
Row player:
[[4 1]
 [0 3]]
```

```
Column player:
[[3 0]
 [1 4]]
```

```
In [33]: # get the NEs
equilibria = newgame.support_enumeration()
for eq in equilibria:
    print(eq)

(array([1., 0.]), array([1., 0.]))
(array([0., 1.]), array([0., 1.]))
(array([0.5, 0.5]), array([0.33333333, 0.66666667]))
```

Learning in repeated games

Here we perform fictitious play on a repeated game in game theory using the `nashpy` library.

Fictitious play is a learning algorithm that models how players adapt their strategies over time in a repeated game. The idea behind fictitious play is that players update their strategies based on the strategies that they observe their opponents playing in the past.

Here I am setting the number of iterations of the game to 10, and the random seed to 0 to ensure that the same results are produced each time the code is run. The `fictitious_play` method is used to perform fictitious play on the game object, and the `iterations` argument is used to specify the number of iterations to run.

The `play_counts` variable stores the count of the number of times each strategy was played by each player over the course of the iterations. The `for` loop then prints each row and column play count for each iteration.

The algorithm starts with a random initial guess for the play counts and updates it

at each iteration based on the opponent's previous play. The play counts converge to a steady state, which is a Nash Equilibrium. In other words, the output of this code would show the number of times each player played each strategy in each iteration of the repeated game, allowing us to see how the strategies evolved over time.

```
In [34]: iterations = 10
         np.random.seed(0)
         play_counts = newgame.fictitious_play(iterations=iterations)
         for row_play_count, column_play_count in play_counts:
             print(row_play_count, column_play_count)
```

```
[0 0] [0 0]
[1. 0.] [0. 1.]
[1. 1.] [1. 1.]
[2. 1.] [1. 2.]
[3. 1.] [2. 2.]
[4. 1.] [3. 2.]
[5. 1.] [4. 2.]
[6. 1.] [5. 2.]
[7. 1.] [6. 2.]
[8. 1.] [7. 2.]
[9. 1.] [8. 2.]
```

In []:

In []:

In []: