The 46-Page Ultimate Guide to Pricing Options and Implied Volatility With Python (with code)

PyQuant News | https://pyquantnews.com | @pyquantnews

The Ultimate Guide to Pricing Options and Implied Volatility With Python

By PyQuant News

Python is an important tool for quantitative and algorithmic trading and research.

Maybe you're an options investor looking to improve your analysis. Or a Python developer interested in investing.

I know quant traders making $1,000,000 per year using Python. I also know 18 year olds trading in their dorm rooms using Python. From data science to rocket science, Python is everywhere.

I built this ebook based on my experience both trading and using Python.

It covers pricing options and dives into computing and analyzing implied volatility.

I only assume you want to learn how to analyze options with Python. While I cover some of the math, you don't need to know it for this ebook to be useful.

You can use the code in this ebook anyway you want to: as is or dive deeper into the analysis.

I'm glad you decided to invest time in learning with me!

Contents

This ultimate guide has everything you need to price options with Python.

Important Jargon

Let's start with the vocabulary.

Terms to know

- **Call option**. Derivative contract which conveys the right (not the obligation) to the buyer of the option to purchase the underlying stock at a specific price on a certain date (European style) or before a certain date (American style).
- **Put option**. Derivative contract which conveys the right (not the obligation) to the buyer of the option to sell the underlying stock at a specific price on a certain date.
- **In the money**. Condition where the stock price is greater than the strike price for a call option or lower than the strike price for a put option
- **Out the money**. Opposite of in the money.
- **At the money**. The stock price is equal to (or very near) the strike price. Usually considered the option with the strike price that is closest to the money.
- **Realized volatility**. Also called statistical or historical volatility, this is usually the annualized standard deviation of the log returns of the underlying over a past whindow of time.
- **Implied volatility**. Volatiltiy parameter that sets the pricing model equal to the market price of the option. (Much more on this later.)

Input parameters to price options

Options priced with Black-Scholes have five input variables, four of which can be observed in the market and one of which is latent (unobservable, more on this later).

- **Underlying stock price (S)**. Price of the underlying stock upon which the option derives its value.
- **Strike price (K)**. The price at which the owner of the option can buy the underlying in the case of a call or sell the underlying in the case of a put.
- **Time to expiration (t)**. Time (as a fraction of a 365 day year) until the expiration of the option.
- **Risk free rate (r)**. So-called risk free rate (this is better described as the rate of interest you would earn in a riskless security over the holding period of the option)
- **Volatility (vol)**. Volatility (or more usually implied volatility) is a latent value not actually observed in the market.

Let's import our modules and declare these variables in Python for later use.

```
# magic function to plot inline
%matplotlib inline

# python standard modules
import time
import datetime as dt
from math import sqrt, pi


# import numpy, pyplot and scipy
import numpy as np
import pandas as pd
from pandas_datareader.yahoo.options import Options
from pandas_datareader.yahoo.daily import YahooDailyReader
import matplotlib as mat

mat.style.use("ggplot")
import matplotlib.pyplot as plt

# for plotting the vol surface
from mpl_toolkits.mplot3d import Axes3D
import scipy
from scipy.stats import norm
from scipy.optimize import brentq
from scipy.interpolate import interp1d

print(
    f"Numpy {np.__version__}\nMatplotlib {mat.__version__}\nScipy {scipy.__version__}\nPandas {pd.__versior
)
```

```
Numpy 1.20.3
Matplotlib 3.4.3
Scipy 1.7.1
Pandas 1.3.4
```

What Are Options?

Options are standardized derivatives contracts that convey the buyer the right (but not obligation) to buy the underlying security in the case of a call option or sell the underlying security in the case of a put option at a given price before a given date (for American style options) or on a given date (European style options).

There are many types of options of which only a few are available for retail traders. "Exotics" exist that are generally large dollar trades (millions of US$) and traded among investment banks. These are generally custom built by quants to help solve a customer's specific financial problem.

We will focus on plain vanilla, european style, equity options so we can use the famous Black-Scholes pricing formula. More on Black-Scholes a bit later.

Most derivatives have a payoff function which describes the value at the end of the life of the contract (expiration). For a call option, the payoff can be expressed simply as:

$$C(S, K) = max(S - K, 0)$$

While the payoff for a put option can be expressed as:

$$P(K, S) = max(K - S, 0)$$

Where S and K are defined above. In this case, K remains fixed during the life of a contract while the underlying stock price, S, fluctuates..

Across a range of potential S values, we can form what is commonly known as the payoff (or risk profile or PnL chart) for an option at expiration.

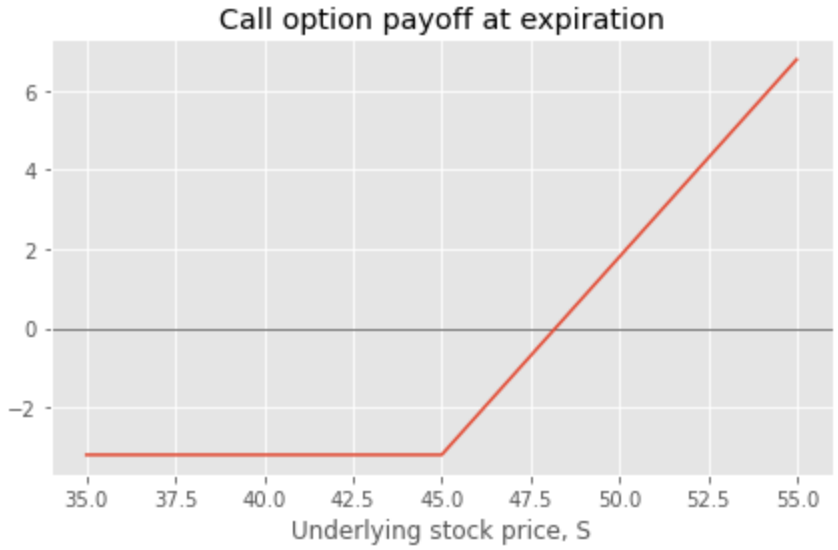
The challenge for quants is to figure out what the price of the option is *before* expiration. This is where option pricing models like Black-Scholes come into play.

```
# underlying stock price
S = 45.0

# series of underlying stock prices to demonstrate a payoff profile
S_ = np.arange(35.0, 55.0, 0.01)

# strike price
K = 45.0

# time to expiration (you'll see this as T-t in the equation)
t = 164.0 / 365.0

# risk free rate (there's nuance to this which we'll describe later)
r = 0.02

# volatility (latent variable which is the topic of this talk)
vol = 0.25

# black scholes prices for demonstrating trades
atm_call_premium = 3.20
atm_put_premium = 2.79

otm_call_premium = 1.39
otm_put_premium = 0.92
```

```python
# use a lambda for a payoff functions
# equivelant to:
#
# def call_payoff(S, K):
#     return np.maximum(S - K, 0.0)
call_payoff = lambda S, K: np.maximum(S_ - K, 0.0)

# equivelant to:
#
# def put_payoff(S, K):
#     return np.maximum(K - S, 0.0)
put_payoff = lambda S, K: np.maximum(K - S_, 0.0)
```

```
# plot the call payoff
plt.figure(1, figsize=(7, 4))
plt.title("Call option payoff at expiration")
plt.xlabel("Underlying stock price, S")
plt.axhline(y=0, lw=1, c="grey")
plt.plot(S_, -atm_call_premium + call_payoff(S_, K))
```

Out[163… [<matplotlib.lines.Line2D at 0x7f94b04d9190>]

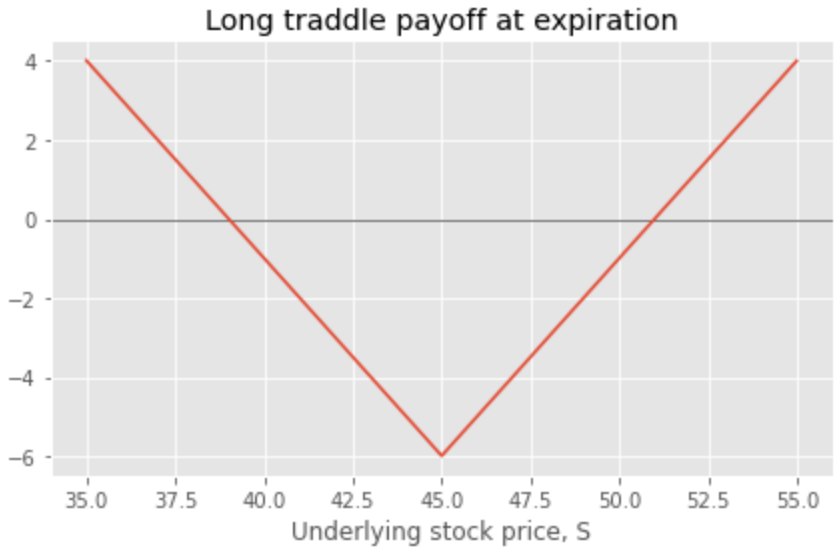



```
# plot the put payoff
plt.figure(2, figsize=(7, 4))
plt.title("Put option payoff at expiration")
plt.xlabel("Underlying stock price, S")
plt.axhline(y=0, lw=1, c="grey")
plt.plot(S_, -atm_put_premium + put_payoff(S_, K))
```

Out[164… [<matplotlib.lines.Line2D at 0x7f94b06f69d0>]

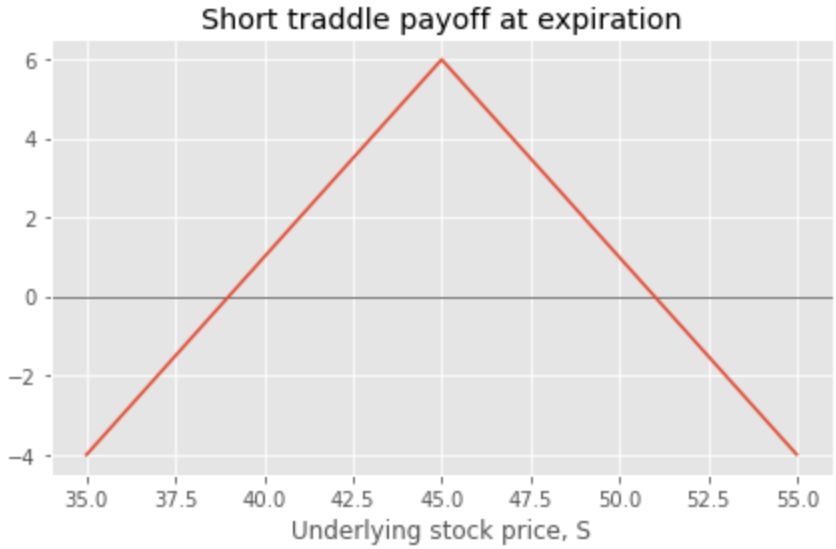
```
# plot a long straddle payoff
long_straddle = call_payoff(S_, K) + put_payoff(S_, K)
long_straddle_premium = -atm_call_premium - atm_put_premium
plt.figure(3, figsize=(7, 4))
plt.title("Long traddle payoff at expiration")
plt.xlabel("Underlying stock price, S")
plt.axhline(y=0, lw=1, c="grey")
plt.plot(S_, long_straddle_premium + long_straddle)
```

Out[165… [<matplotlib.lines.Line2D at 0x7f94d1f0e8b0>]



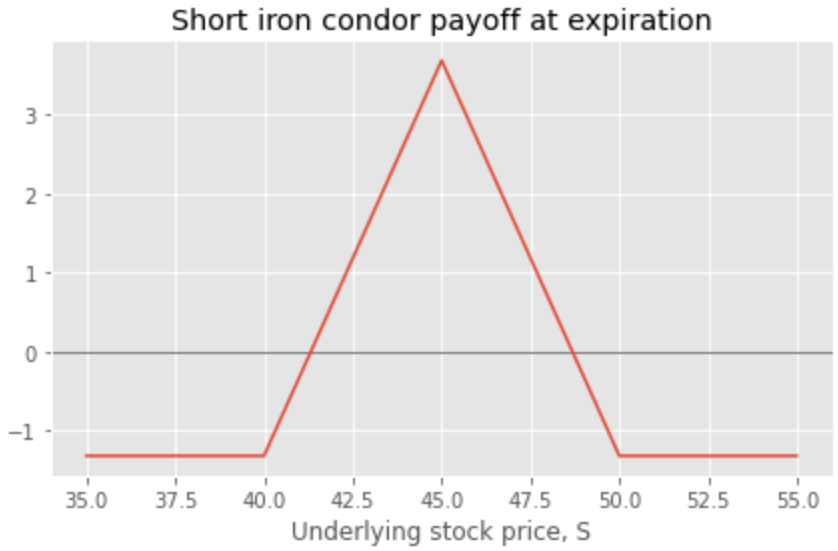

```
# plot a short straddle payoff
short_straddle = -call_payoff(S_, K) - put_payoff(S_, K)
short_straddle_premium = atm_call_premium + atm_put_premium
plt.figure(4, figsize=(7, 4))
plt.title("Short traddle payoff at expiration")
plt.xlabel("Underlying stock price, S")
plt.axhline(y=0, lw=1, c="grey")
plt.plot(S_, short_straddle_premium - long_straddle)
```

Out[166… [<matplotlib.lines.Line2D at 0x7f94e28aec70>]



```
# plot a short iron condor payoff
short_iron_condor = (
    call_payoff(S_, K + 5)
    - call_payoff(S_, K)
    - put_payoff(S_, K)
    + put_payoff(S_, K - 5)
)
short_iron_condor_premium = (
    -otm_call_premium + atm_call_premium + atm_put_premium - otm_put_premium
)
plt.figure(5, figsize=(7, 4))
plt.title("Short iron condor payoff at expiration")
plt.xlabel("Underlying stock price, S")
plt.axhline(y=0, lw=1, c="grey")
plt.plot(S_, short_iron_condor_premium + short_iron_condor)
```

Out[167… [<matplotlib.lines.Line2D at 0x7f94a06fbf70>]



What Is the Black-Scholes Option Pricing Model?

Black-Scholes is largely understood as an options pricing formula but it is really a framework that models a financial market. Within this framework the options pricing formula is be derived through stochasitc differentiation.

The Black–Scholes model assumes that the market consists of at least one risky asset, usually called the stock, and one riskless asset, usually called the money market, cash, or bond.

Assumptions on the market:

- The rate of return on the riskless asset is constant and thus called the risk-free interest rate
- The instantaneous log returns of the stock price is an infinitesimal random walk with drift and volatility (geometric Brownian motion) and assumes this drift and volatility are constant
- The stock does not pay a dividend

Assumptions on the underlying security:

- There is no arbitrage opportunity (i.e. there is no way to make a riskless profit)
- It is possible to borrow and lend any amount, even fractional, of cash at the riskless rate
- It is possible to buy and sell any amount, even fractional, of the stock (this includes short selling)
- The above transactions do not incur any fees or costs (i.e. frictionless market).

As any trader knows, not a single one of these assumptions are actually true in reality. Probably the most imporant is the assumption of constant volatility which motivates this presentation.

Assume these assumptions *do* hold and suppose there is a derivative security also trading in this market. We specify that this security will have a certain payoff at a specified date in the future, depending on the value(s) taken by the stock up to that date. It is a surprising fact that the derivative's price is completely determined at the current time, even though we do not know what path the stock price will take in the future. (These are the payoffs we saw above.) For the special case of a European call or put option, Black and Scholes showed that *"it is possible to create a hedged position, consisting of a long position in the stock and a short position in the option, whose value will not depend on the price of the stock"*. Their dynamic hedging strategy led to a partial differential equation which governed the price of the option. Its solution is given by the Black–Scholes formula.

(Some of) the Math

The famous Black-Scholes model:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

The key financial insight behind the equation is that one can perfectly hedge the option by buying and selling the underlying asset in just the right way and consequently create an arbitrage free environment. This hedge, in turn, implies that there is only one right price for the option, as returned by the Black–Scholes formula. For this dynamic hedging to work, one must continuously hedge the movement in the underlying stock with the option which in reality, is prohibitive due to transaction costs.

I'll skip the derivation of the option pricing formulas from the partial differential equation but the result solves for a call option

$$C(S,t) = N(d_1)S - N(d_2)Ke^{-r(T-t)}$$

and put option

$$P(S,t) = N(-d_2)Ke^{-r(T-t)} - N(-d_1)S$$

where

$$d_1 = \frac{ln(\frac{S}{K}) + (r + \frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}}$$

and

$$d_2 = d_1 - \sigma\sqrt{T-t}$$

and finally (as we discussed above)

- $N(\cdot)$ is the cumulative distribution function of the standard normal distribution
- $T-t$ is the time to maturity (I use t in the code)
- S is the spot price of the underlying asset
- K is the strike price
- r is the risk free rate (annual rate, expressed in terms of continuous compounding)
- σ is the volatility of returns of the underlying asset (I use vol in the code)

Black-Scholes Formula in Python

Finally, enough math and some code. We'll be using Numpy so we vectorize the function which will help us later.

```
def N(z):
    """ Normal cumulative density function

    :param z: point at which cumulative density is calculated
    :return: cumulative density under normal curve
    """
    return norm.cdf(z)


def black_scholes_call_value(S, K, r, t, vol):
    """ Black-Scholes call option

    :param S: underlying
    :param K: strike price
    :param r: rate
    :param t: time to expiration
    :param vol: volatility
    :return: BS call option value
    """
    d1 = (1.0 / (vol * np.sqrt(t))) * (np.log(S / K) + (r + 0.5 * vol ** 2.0) * t)
    d2 = d1 - (vol * np.sqrt(t))

    return N(d1) * S - N(d2) * K * np.exp(-r * t)


def black_scholes_put_value(S, K, r, t, vol):
    """ Black-Scholes put option

    :param S: underlying
    :param K: strike price
    :param r: rate
    :param t: time to expiration
    :param vol: volatility
    :return: BS put option value
    """
    d1 = (1.0 / (vol * np.sqrt(t))) * (np.log(S / K) + (r + 0.5 * vol ** 2.0) * t)
    d2 = d1 - (vol * np.sqrt(t))

    return N(-d2) * K * np.exp(-r * t) - N(-d1) * S
```

```
call_value = black_scholes_call_value(S, K, r, t, vol)
put_value = black_scholes_put_value(S, K, r, t, vol)

print(f"Black-Scholes call value {call_value:.2f}")
print(f"Black-Scholes put value {put_value:.2f}")
```

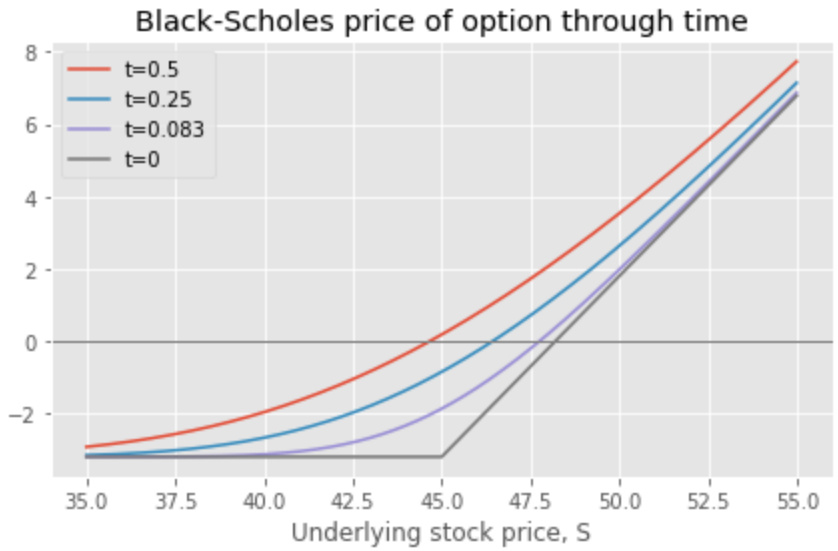
```
Black-Scholes call value 3.20
Black-Scholes put value 2.79
```

Because we vectorized the function, we can plot the Black-Scholes call value along a range of underlying stock prices. Going back to our example of the option value at expiration, let's see what it looks like with six, three and one month left to expiration.

```
# get the value of the option with six months to expiration
black_scholes_call_value_six_months = (
    black_scholes_call_value(S_, K, r, 0.5, vol) - atm_call_premium
)

# get the value of the option with three months to expiration
black_scholes_call_value_three_months = (
    black_scholes_call_value(S_, K, r, 0.25, vol) - atm_call_premium
)

# get the value of the option with one month to expiration
black_scholes_call_value_one_month = (
    black_scholes_call_value(S_, K, r, 1.0 / 12.0, vol) - atm_call_premium
)

# get payoff value at expiration
call_payoff_at_expiration = call_payoff(S_, K) - atm_call_premium
```

```
# plot the call payoffs
plt.figure(3, figsize=(7, 4))
plt.plot(S_, black_scholes_call_value_six_months)
plt.plot(S_, black_scholes_call_value_three_months)
plt.plot(S_, black_scholes_call_value_one_month)
plt.plot(S_, call_payoff_at_expiration)
plt.axhline(y=0, lw=1, c="grey")
plt.title("Black-Scholes price of option through time")
plt.xlabel("Underlying stock price, S")
plt.legend(["t=0.5", "t=0.25", "t=0.083", "t=0"], loc=2)
```

Out[171… `<matplotlib.legend.Legend at 0x7f94e29bd7f0>`



As you can see from the plot above, as the time to expiration decreases, the value of the option collapses towards the value of the option at expiration, or the payoff value.

The Greeks

A discussion about Black-Scholes would not be complete without a discussion of the Greeks. In mathematical terms, the Greeks are simply the partial derivatives of the option pricing formula. In other words, these are the sensitivities in the movement of the option price relative to the movement of other aspects of the input variables.

Because the assumptions required for the Black-Scholes model do not hold true in practice, the plain vanilla formula presented above is not usually used for pricing options for the purposes of trading. (In fact if one were to rely on Black-Scholes to manage a portfolio of options, that trader would surely lose money.) It is however, largely used for understanding the sensitivities of the option with respect to it's parameters and to hedge. The most common Greeks are as follows.

- **Delta**. Partial derivative of the option value with respect to the change in the underlying stock price. Delta measures how the underlying option moves with respect to moves in the underlying stock. The formula for delta is different for calls and puts.

  > Call delta
  >
  > $$\frac{\partial V}{\partial S} = N(d_1)$$
  >
  > Put delta
  >
  > $$\frac{\partial V}{\partial S} = N(d_1) - 1$$

- **Gamma**. Second partial derivative of the option value with respect to the change in the underlying stock price. Gamma measures movements in delta or the convexity in the value of the option with respect to the underlying.

  > Gamma (same for call and put)
  >
  > $$\frac{\partial^2 V}{\partial S^2} = \frac{\phi(d_1)}{S\sigma\sqrt{T-t}}$$

- **Vega**. Partial derivative of the option value with respect to the change in the volatility of the underling. Vega measures how the option price moves with respect to the volatility of the underlying. (Note vega is not a real greek letter.)

  > Vega (same for call and put)
  >
  > $$\frac{\partial V}{\partial \sigma} = S\phi(d_1)\sqrt{T-t}$$

- **Theta**. Partial derivative of the option value with respect to the change in time. Theta measures how the value of the option decays as time passes. This was demonstrated in the plot above.

  > Call theta

> $$\frac{\partial V}{\partial t} = -\frac{S\phi(d_1)\sigma}{2\sqrt{T-t}} - rKe^{-r(T-t)}N(d_2)$$
>
> Put theta
>
> $$\frac{\partial V}{\partial t} = -\frac{S\phi(d_1)\sigma}{2\sqrt{T-t}} + rKe^{-r(T-t)}N(-d_2)$$

- **Rho**. Partial derivative of the option value with respect to change in the risk-free interest rate. Rho measures how the option value changes as the interest rate changes.

  > Call rho
  >
  > $$\frac{\partial V}{\partial r} = K(T-t)e^{-r(T-t)}N(d_2)$$
  >
  > Put rho
  >
  > $$\frac{\partial V}{\partial r} = -K(T-t)e^{-r(T-t)}N(-d_2)$$

Where ϕ

> $$\phi(x) = \frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}}$$

and $N(\cdot)$, d_1 and d_2 are defined above.

The Greeks in Python

Although we won't be discussing the greeks any further in this talk, let's code them up for completeness.

```
# helper function phi
def phi(x):
    """ Phi helper function

    """
    return np.exp(-0.5 * x * x) / (sqrt(2.0 * pi))


# shared
def gamma(S, K, r, t, vol):
    """ Black-Scholes gamma

    :param S: underlying
    :param K: strike price
    :param r: rate
    :param t: time to expiration
    :param vol: volatility
    :return: gamma
    """
    d1 = (1.0 / (vol * np.sqrt(t))) * (np.log(S / K) + (r + 0.5 * vol ** 2.0) * t)

    return phi(d1) / (S * vol * sqrt(t))


def vega(S, K, r, t, vol):
    """ Black-Scholes vega

    :param S: underlying
    :param K: strike price
    :param r: rate
    :param t: time to expiration
    :param vol: volatility
    :return: vega
    """
    d1 = (1.0 / (vol * np.sqrt(t))) * (np.log(S / K) + (r + 0.5 * vol ** 2.0) * t)

    return (S * phi(d1) * sqrt(t)) / 100.0


# call options
def call_delta(S, K, r, t, vol):
    """ Black-Scholes call delta

    :param S: underlying
    :param K: strike price
    :param r: rate
    :param t: time to expiration
    :param vol: volatility
    :return: call delta
    """
    d1 = (1.0 / (vol * np.sqrt(t))) * (np.log(S / K) + (r + 0.5 * vol ** 2.0) * t)

    return N(d1)
```

```
def call_theta(S, K, r, t, vol):
    """ Black-Scholes call theta

    :param S: underlying
    :param K: strike price
    :param r: rate
    :param t: time to expiration
    :param vol: volatility
    :return: call theta
    """
    d1 = (1.0 / (vol * np.sqrt(t))) * (np.log(S / K) + (r + 0.5 * vol ** 2.0) * t)
    d2 = d1 - (vol * np.sqrt(t))

    theta = -((S * phi(d1) * vol) / (2.0 * np.sqrt(t))) - (
        r * K * np.exp(-r * t) * N(d2)
    )
    return theta / 365.0


def call_rho(S, K, r, t, vol):
    """ Black-Scholes call rho

    :param S: underlying
    :param K: strike price
    :param r: rate
    :param t: time to expiration
    :param vol: volatility
    :return: call rho
    """
    d1 = (1.0 / (vol * np.sqrt(t))) * (np.log(S / K) + (r + 0.5 * vol ** 2.0) * t)
    d2 = d1 - (vol * np.sqrt(t))

    rho = K * t * np.exp(-r * t) * N(d2)
    return rho / 100.0


# put options
def put_delta(S, K, r, t, vol):
    """ Black-Scholes put delta

    :param S: underlying
    :param K: strike price
    :param r: rate
    :param t: time to expiration
    :param vol: volatility
    :return: put delta
    """
    d1 = (1.0 / (vol * np.sqrt(t))) * (np.log(S / K) + (r + 0.5 * vol ** 2.0) * t)

    return N(d1) - 1.0
```

```
def put_theta(S, K, r, t, vol):
    """ Black-Scholes put theta

    :param S: underlying
    :param K: strike price
    :param r: rate
    :param t: time to expiration
    :param vol: volatility
    :return: put theta
    """
    d1 = (1.0 / (vol * np.sqrt(t))) * (np.log(S / K) + (r + 0.5 * vol ** 2.0) * t)
    d2 = d1 - (vol * np.sqrt(t))

    theta = -((S * phi(d1) * vol) / (2.0 * np.sqrt(t))) + (
        r * K * np.exp(-r * t) * N(-d2)
    )
    return theta / 365.0


def put_rho(S, K, r, t, vol):
    """ Black-Scholes put rho

    :param S: underlying
    :param K: strike price
    :param r: rate
    :param t: time to expiration
    :param vol: volatility
    :return: put rho
    """
    d1 = (1.0 / (vol * np.sqrt(t))) * (np.log(S / K) + (r + 0.5 * vol ** 2.0) * t)
    d2 = d1 - (vol * np.sqrt(t))

    rho = -K * t * np.exp(-r * t) * N(-d2)
    return rho / 100.0
```

```
# print each of the results
print("Black-Scholes call delta %0.4f" % call_delta(S, K, r, t, vol))
print("Black-Scholes put delta %0.4f" % put_delta(S, K, r, t, vol))
print("Black-Scholes gamma %0.4f" % gamma(S, K, r, t, vol))
print("Black-Scholes vega %0.4f" % vega(S, K, r, t, vol))
print("Black-Scholes call theta %0.4f" % call_theta(S, K, r, t, vol))
print("Black-Scholes put theta %0.4f" % put_theta(S, K, r, t, vol))
print("Black-Scholes call rho %0.4f" % call_rho(S, K, r, t, vol))
print("Black-Scholes put rho %0.4f" % put_rho(S, K, r, t, vol))
```

```
Black-Scholes call delta 0.5546
Black-Scholes put delta -0.4454
Black-Scholes gamma 0.0524
Black-Scholes vega 0.1192
Black-Scholes call theta -0.0103
Black-Scholes put theta -0.0078
Black-Scholes call rho 0.0978
Black-Scholes put rho -0.1026
```

Realized Volatility

Also refered to as historical or statistical volatility, realized volatility estimates the volatility of the underlying price over a defined period of time. It's generally used as an input into a pricing model such as the Black-Scholes model to calibrate the value to then solve for implied volatiltiy. Realized volatility is not generally used for making trading decisions but can be informative in understanding how the underlying has moved in the past.

The bonus lecture presents several realized volatility estimators. Here, we'll use the most common estimator which is simple standard deviation of log returns. A popular analysis technique, volatility cones, is also presented.

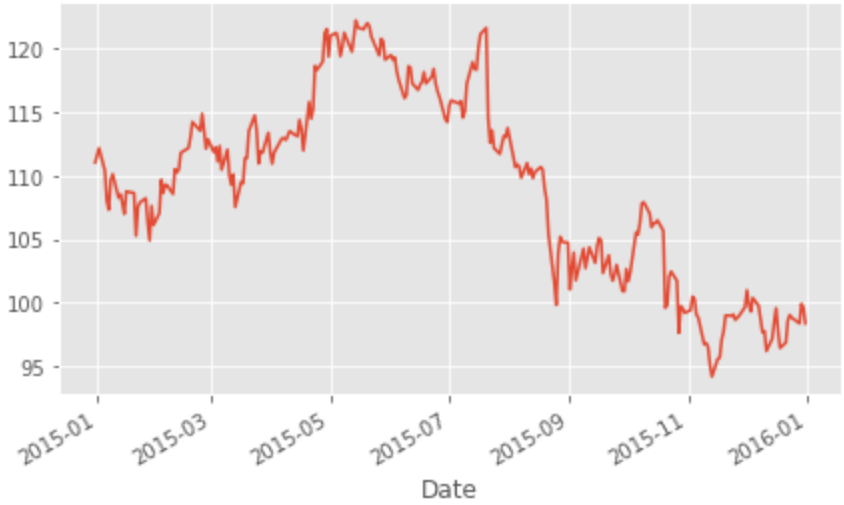
We'll use `pandas_datareader` to get historical prices for the underlying and compute a 30-day, annualized volatility measure.

```
# define a stock symbol
underlying_symbol = "IBM"

# define a YahooDailyReader object
price_obj = YahooDailyReader(underlying_symbol, start="2015-01-01", end="2015-12-31")

# request historical data
prices = price_obj.read()

# yahoo returns prices in ascending order, sort to descending order
prices.sort_index(ascending=False, inplace=True)

# let's pickle the dataframe so we don't have to hit the network every time
prices.to_pickle("underlying_prices.pickle")
```

```
# read the original frame in from cache (pickle)
prices = pd.read_pickle("underlying_prices.pickle")
```

```
prices["Adj Close"].plot(figsize=(7, 4))
```

Out[178… <AxesSubplot:xlabel='Date'>



Use the last month (roughly 22 trading days) of the data set to compute

```
# compute the log returns from the adjusted closing price
log_return = (prices["Adj Close"] / prices["Adj Close"].shift(-1)).apply(np.log)

# take the standard deviation of the last month of data (22 trading days)
sd_of_log_returns = log_return.head(22).std()

# multiply by the square root of the number of trading days in a year (252) to annualize
realized_volatility = sd_of_log_returns * sqrt(252)
```

```
print("One month, annualized realized volatility %0.4f" % realized_volatility)
```

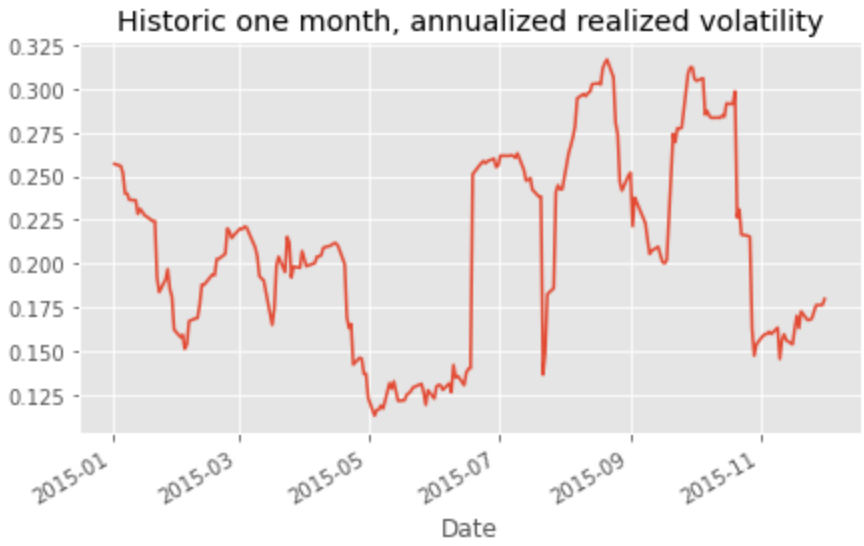
```
One month, annualized realized volatility 0.1798
```

```
# use the pandas DataFrame.rolling method to create a rolling standard deviation of
# log returns, then multiply by square root of 252 to annualize
rolling_realized_volatility = log_return.rolling(window=22, center=False).std() * sqrt(
    252
)
```

```
# note that we plot plot directly from a pandas data frame!
rolling_realized_volatility.plot(
    title="Historic one month, annualized realized volatility", figsize=(7, 4)
)
```

Out[182…

```
<AxesSubplot:title={'center':'Historic one month, annualized realized volatility'},
xlabel='Date'>
```



Implied Volatility

The Black-Scholes pricing formula is usually used to solve for implied volatility. Quite simply, this means setting the Black-Scholes pricing formula equal to the market observed price and using a root finding algorithm to solve for the volatility parameter which sets the difference (between model and market price) to zero.

The implied volatility is generally used for making trading decisions, calibrating other more exotic securities and researching market anomolies. For example, many traders use the so called volatility skew to understand the relative value of an option to other options trading in the market. We'll see examples of the skew later.

First we create an objective function which we'll pass to the Scipy implementation of the Brentq algorithm. Then we'll create an implied volatilty function to return the volatility parameter that sets the observed market price to the model price.

```
def call_implied_volatility_objective_function(
    S, K, r, t, vol, call_option_market_price
):
    """ Objective function which sets market and model prices to zero

    :param S: underlying
    :param K: strike price
    :param r: rate
    :param t: time to expiration
    :param vol: volatility
    :param call_option_market_price: market observed option price
    :return: error between market and model price
    """
    return call_option_market_price - black_scholes_call_value(S, K, r, t, vol)
```

```python
def call_implied_volatility(
    S, K, r, t, call_option_market_price, a=-2.0, b=2.0, xtol=1e-6
):
    """ Call implied volatility function

    :param S: underlying
    :param K: strike price
    :param r: rate
    :param t: time to expiration
    :param call_option_market_price: market observed option price
    :param a: lower bound for brentq method
    :param b: upper gound for brentq method
    :param xtol: tolerance which is considered good enough
    :return: volatility to sets the difference between market and model price to zero

    """
    # avoid mirroring outer scope
    _S, _K, _r, _t, _call_option_market_price = S, K, r, t, call_option_market_price

    # define a nested function that takes our target param as the input
    def fcn(vol):

        # returns the difference between market and model price at given volatility
        return call_implied_volatility_objective_function(
            _S, _K, _r, _t, vol, _call_option_market_price
        )

    # first we try to return the results from the brentq algorithm
    try:
        result = brentq(fcn, a=a, b=b, xtol=xtol)

        # if the results are *too* small, sent to np.nan so we can later interpolate
        return np.nan if result <= 1.0e-6 else result

    # if it fails then we return np.nan so we can later interpolate the results
    except ValueError:
        return np.nan


def put_implied_volatility_objective_function(S, K, r, t, vol, put_option_market_price):
    """ Objective function which sets market and model prices to zero

    :param S: underlying
    :param K: strike price
    :param r: rate
    :param t: time to expiration
    :param vol: volatility
    :param call_option_market_price: market observed option price
    :return: error between market and model price
    """
    return put_option_market_price - black_scholes_put_value(S, K, r, t, vol)
```

```python
def put_implied_volatility(
    S, K, r, t, put_option_market_price, a=-2.0, b=2.0, xtol=1e-6
):
    """ Put implied volatility function

    :param S: underlying
    :param K: strike price
    :param r: rate
    :param t: time to expiration
    :param call_option_market_price: market observed option price
    :param a: lower bound for brentq method
    :param b: upper gound for brentq method
    :param xtol: tolerance which is considered good enough
    :return: volatility to sets the difference between market and model price to zero

    """

    # avoid mirroring out scope
    _S, _K, _r, _t, _put_option_market_price = S, K, r, t, put_option_market_price

    # define a nsted function that takes our target param as the input
    def fcn(vol):

        # returns the difference between market and model price at given volatility
        return put_implied_volatility_objective_function(
            _S, _K, _r, _t, vol, _put_option_market_price
        )

    # first we try to return the results from the brentq algorithm
    try:
        result = brentq(fcn, a=a, b=b, xtol=xtol)

        # if the results are *too* small, sent to np.nan so we can later interpolate
        return np.nan if result <= 1.0e-6 else result

    # if it fails then we return np.nan so we can later interpolate the results
    except ValueError:
        return np.nan
```

```python
# get the call and put values to test the implied volatility output
call_model_price = black_scholes_call_value(S, K, r, t, vol)
print(
    "Call implied volatility if market and model were equal (should be close to 0.25) %0.6f"
    % call_implied_volatility(S, K, r, t, call_model_price)
)

put_model_price = black_scholes_put_value(S, K, r, t, vol)
print(
    "Put implied volatility if market and model were equal (should be close to 0.25) %0.6f"
    % put_implied_volatility(S, K, r, t, put_model_price)
)
```

```
Call implied volatility if market and model were equal (should be close to 0.25) 0.
250000
Put implied volatility if market and model were equal (should be close to 0.25) 0.2
50000
```

Getting Real Options Market Data

> Yahoo discontinued use of their API for options data and the $Options$ call no longer works.
> There's no known way to acquire options data for free. This example uses cached data.

With the excellent Pandas library (you are using Pandas, right?) we can grab entire options chains directly from Yahoo!. This is slow (it has to scrape several pages), but very useful. Chains include all options of all strikes for and expirations for a particular underlying stock.

The chains are returned in a very handy `MultiIndex pandas.DataFrame` object. Let's see how this works.

```
# define a stock symbol
underlying_symbol = "IBM"

# # define a Options object
# options_obj = Options('IBM')

# # request all chains for the underlying symbol
# options_frame_live = options_obj.get_all_data()

# let's pickle the dataframe so we don't have to hit the network every time
# options_frame_live.to_pickle('options_frame.pickle')
```

```
# read the original frame in from cache (pickle)
options_frame = pd.read_pickle("options_frame.pickle")
```

```
# take a quick look at the DataFrame that returned
options_frame.info()
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 519 entries, (70.0, Timestamp('2016-07-15 00:00:00'), 'put', 'IBM160715
P00070000') to (280.0, Timestamp('2017-01-20 00:00:00'), 'put', 'IBM170120P00280000
')
Data columns (total 13 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   Last              519 non-null    float64
 1   Bid               519 non-null    float64
 2   Ask               519 non-null    float64
 3   Chg               519 non-null    float64
 4   PctChg            519 non-null    object
 5   Vol               519 non-null    int64
 6   Open_Int          519 non-null    int64
 7   IV                519 non-null    object
 8   Root              519 non-null    object
 9   IsNonstandard     519 non-null    bool
 10  Underlying        519 non-null    object
 11  Underlying_Price  519 non-null    float64
 12  Quote_Time        519 non-null    datetime64[ns]
dtypes: bool(1), datetime64[ns](1), float64(5), int64(2), object(4)
memory usage: 75.2+ KB
```

```
# and the first ten records
options_frame.head()
```

Out[190…

	Last	Bid	Ask	Chg	PctChg	Vol	Open_Int

Strike	Expiry	Type	Symbol								
70.0	2016-07-15	put	IBM160715P00070000	0.30	0.21	0.35	0.08	36.36%	15	10	4
	2017-01-20	put	IBM170120P00070000	1.30	0.97	1.40	0.55	73.33%	48	118	4
	2018-01-19	call	IBM180119C00070000	63.00	57.50	62.50	0.00	0.00%	15	4	3
		put	IBM180119P00070000	2.30	0.96	2.50	0.32	16.16%	2	97	3
75.0	2016-07-15	put	IBM160715P00075000	0.41	0.30	0.45	0.01	2.50%	166	3	4

We're going to do some work to the `DataFrame` so that we can apply the Black-Scholes price and implied volatility to each of the options. So first let's do some clean up.

```
# reset the index so the strike and expiration become columns
options_frame.reset_index(inplace=True)

# rename the columns for consistency
columns = {
    "Expiry": "Expiration",
    "Type": "OptionType",
    "Symbol": "OptionSymbol",
    "Vol": "Volume",
    "Open_Int": "OpenInterest",
    "Underlying_Price": "UnderlyingPrice",
    "Quote_Time": "QuoteDatetime",
    "Underlying": "UnderlyingSymbol",
    "Chg": "OptionChange",
}

options_frame.rename(columns=columns, inplace=True)
```

Let's define some helper functions that we will apply to the `DataFrame` . These will provide some of the inputs to the option valuation functions.

```
def _get_days_until_expiration(series):
    """ Return the number of days until expiration

    :param series: row of the dataframe, accessible by label
    :return: days until expiration

    """

    expiration = series["Expiration"]

    # add the hours to the expiration date so we get the math correct
    date_str = expiration.strftime("%Y-%m-%d") + " 23:59:59"

    # convert date string into datetime object
    expiry = dt.datetime.strptime(date_str, "%Y-%m-%d %H:%M:%S")

    # get today
    #   since we need to use a cached data source, revert to that date
    #   today = dt.datetime.today()
    today = dt.datetime(2016, 1, 18)

    # return the difference and add one to count for today
    return (expiry - today).days + 1
```

```python
def _get_time_fraction_until_expiration(series):
    """ Return the fraction of a year until expiration

    You don't always have to be this precise. The difference in price
    based on a few hours for long dated options or far OTM options
    will not be affected. However for liquid, ATM options with short
    expiration windows, every second counts!

    :param series: row of the dataframe, accessible by label
    :return: fraction of a year until expiration

    """

    expiration = series["Expiration"]

    # add the hours to the expiration date so we get the math correct
    date_str = expiration.strftime("%Y-%m-%d") + " 23:59:59"

    # convert date string into datetime object
    time_tuple = time.strptime(date_str, "%Y-%m-%d %H:%M:%S")

    # get the number of seconds from the epoch until expiration
    expiry_in_seconds_from_epoch = time.mktime(time_tuple)

    # get the number of seconds from the epoch to right now
    #   since we need to use a cached data source, revert to that date
    #   today = dt.datetime.today()
    #   right_now_in_seconds_from_epoch = time.time()

    right_now_in_seconds_from_epoch = dt.datetime(2016, 1, 18).timestamp()

    # get the total number of seconds to expiration
    seconds_until_expiration = (
        expiry_in_seconds_from_epoch - right_now_in_seconds_from_epoch
    )

    # seconds in year
    seconds_in_year = 31536000.0

    # fraction of seconds to expiration to total in year, rounded
    return max(seconds_until_expiration / seconds_in_year, 1e-10)
```

```
# define terms and associated rates, these should coincide with our options
# these rates are taken from the yield curve
terms = [30, 3 * 30, 6 * 30, 12 * 30, 24 * 30, 36 * 30, 60 * 30]
rates = [0.0001, 0.0009, 0.0032, 0.0067, 0.0097, 0.0144, 0.0184]


def _get_rate(series):
    """ Interpolate rates out to 30 years
        Note computing rates like this is not strictly theoretically
        correct but works for illustrative purposes

    :param series: row of the dataframe, accessable by label
    :return interpolated interest rate based on term structure

    """
    days = series["DaysUntilExpiration"]

    # generate terms for every thirty days up until our longest expiration
    new_terms = [i for i in range(30, (60 * 30) + 1)]

    # create linear interpolation model
    f = interp1d(terms, rates, kind="linear")

    # interpolate the values based on the new terms we created above
    ff = f(new_terms)

    # return the interpolated rate given the days to expiration
    return round(ff[max(days, 30) - 30], 8)


def _get_mid(series):
    """ Get the mid price between bid and ask

    :param series: row of the dataframe, accessable by label
    :return mid price

    """
    bid = series["Bid"]
    ask = series["Ask"]
    last = series["Last"]

    # if the bid or ask doesn't exist, return 0.0
    if np.isnan(ask) or np.isnan(bid):
        return 0.0

    # if the bid or ask are 0.0, return the last traded price
    elif ask == 0.0 or bid == 0.0:
        return last
    else:
        return (ask + bid) / 2.0
```

We can apply these functions to each row of the `DataFrame` by setting the `axis` argument to `1` `apply` method.

```
# use the apply method to pass each row as a series to the various methods, returns a series in this case
options_frame["DaysUntilExpiration"] = options_frame.apply(
    _get_days_until_expiration, axis=1
)
options_frame["TimeUntilExpiration"] = options_frame.apply(
    _get_time_fraction_until_expiration, axis=1
)
options_frame["InterestRate"] = options_frame.apply(_get_rate, axis=1)
options_frame["Mid"] = options_frame.apply(_get_mid, axis=1)
```

Let's see what we've done.

```
options_frame.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 519 entries, 0 to 518
Data columns (total 21 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   Strike               519 non-null    float64
 1   Expiration           519 non-null    datetime64[ns]
 2   OptionType           519 non-null    object
 3   OptionSymbol         519 non-null    object
 4   Last                 519 non-null    float64
 5   Bid                  519 non-null    float64
 6   Ask                  519 non-null    float64
 7   OptionChange         519 non-null    float64
 8   PctChg               519 non-null    object
 9   Volume               519 non-null    int64
 10  OpenInterest         519 non-null    int64
 11  IV                   519 non-null    object
 12  Root                 519 non-null    object
 13  IsNonstandard        519 non-null    bool
 14  UnderlyingSymbol     519 non-null    object
 15  UnderlyingPrice      519 non-null    float64
 16  QuoteDatetime        519 non-null    datetime64[ns]
 17  DaysUntilExpiration  519 non-null    int64
 18  TimeUntilExpiration  519 non-null    float64
 19  InterestRate         519 non-null    float64
 20  Mid                  519 non-null    float64
dtypes: bool(1), datetime64[ns](2), float64(9), int64(3), object(6)
memory usage: 81.7+ KB
```

```
options_frame.head()
```

Out[197…

	Strike	Expiration	OptionType	OptionSymbol	Last	Bid	Ask	OptionChange	PctChg
0	70.0	2016-07-15	put	IBM160715P00070000	0.30	0.21	0.35	0.08	36.36%
1	70.0	2017-01-20	put	IBM170120P00070000	1.30	0.97	1.40	0.55	73.33%
2	70.0	2018-01-19	call	IBM180119C00070000	63.00	57.50	62.50	0.00	0.00%
3	70.0	2018-01-19	put	IBM180119P00070000	2.30	0.96	2.50	0.32	16.16%
4	75.0	2016-07-15	put	IBM160715P00075000	0.41	0.30	0.45	0.01	2.50%

5 rows × 21 columns

Applying Implied Volatility

We now have all the input parameters we need to follow the same procedure to add the Black-Scholes price and the associated greeks. We'll begin as before by defining the functions we'll apply. Note that we have to compute the implied volatiltiy first which we'll use as the vol input to the Black-Scholes formula. It is common practice to use the Black-Scholes formula to compute the implied volalatility then to use that result to compute the greeks.

We'll take the mid price (price between bid and ask) as this is generally considered the most representative price to use. As we'll see, there will be issues with the result which we'll have to clean.

```
def _get_implied_vol_mid(series):
    """

    """
    option_type = series["OptionType"]
    S = series["UnderlyingPrice"]
    K = series["Strike"]
    r = series["InterestRate"]
    t = series["TimeUntilExpiration"]
    mid = series["Mid"]

    # build method name
    meth_name = "{0}_implied_volatility".format(option_type)

    # call from globals()
    return float(globals().get(meth_name)(S, K, r, t, mid))
```

```
# apply the function to the dataframe rowwise
options_frame["ImpliedVolatilityMid"] = options_frame.apply(
    _get_implied_vol_mid, axis=1
)
```

```
# again, let's take a look
options_frame.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 519 entries, 0 to 518
Data columns (total 22 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   Strike               519 non-null    float64
 1   Expiration           519 non-null    datetime64[ns]
 2   OptionType           519 non-null    object
 3   OptionSymbol         519 non-null    object
 4   Last                 519 non-null    float64
 5   Bid                  519 non-null    float64
 6   Ask                  519 non-null    float64
 7   OptionChange         519 non-null    float64
 8   PctChg               519 non-null    object
 9   Volume               519 non-null    int64
 10  OpenInterest         519 non-null    int64
 11  IV                   519 non-null    object
 12  Root                 519 non-null    object
 13  IsNonstandard        519 non-null    bool
 14  UnderlyingSymbol     519 non-null    object
 15  UnderlyingPrice      519 non-null    float64
 16  QuoteDatetime        519 non-null    datetime64[ns]
```

```
 17  DaysUntilExpiration   519 non-null    int64
 18  TimeUntilExpiration   519 non-null    float64
 19  InterestRate          519 non-null    float64
 20  Mid                   519 non-null    float64
 21  ImpliedVolatilityMid  507 non-null    float64
dtypes: bool(1), datetime64[ns](2), float64(10), int64(3), object(6)
memory usage: 85.8+ KB
```

```
options_frame.head()
```

Out[201…

	Strike	Expiration	OptionType	OptionSymbol	Last	Bid	Ask	OptionChange	PctChg
0	70.0	2016-07-15	put	IBM160715P00070000	0.30	0.21	0.35	0.08	36.36%
1	70.0	2017-01-20	put	IBM170120P00070000	1.30	0.97	1.40	0.55	73.33%
2	70.0	2018-01-19	call	IBM180119C00070000	63.00	57.50	62.50	0.00	0.00%
3	70.0	2018-01-19	put	IBM180119P00070000	2.30	0.96	2.50	0.32	16.16%
4	75.0	2016-07-15	put	IBM160715P00075000	0.41	0.30	0.45	0.01	2.50%

5 rows × 22 columns

Interpolating Missing and Bad Implied Volatility Values

In the `call_implied_volatility` and `put_implied_volatility` functions we test for the Brentq solver blowing up. The solver will blow up if the option's mid price leads to a negative implied volatility. This happens with deep in or out of the money options where market makers keep the bid-ask spread wide.

Let's find the options where the implied volatility is `np.nan`.

```
bad_iv = options_frame[np.isnan(options_frame["ImpliedVolatilityMid"])]
```

```
# map the count function to each strike where there is a nan implied volatility
bad_iv.groupby(["Strike"]).count()["Expiration"]
```

Out[203…

```
Strike
70.0     1
75.0     1
80.0     3
85.0     2
90.0     2
100.0    2
105.0    1
Name: Expiration, dtype: int64
```

Now we'll interpolate the missing implied volatility values. There are *much* more sophisticated ways of doing this (see here for example). In fact this is where professional options traders make their money. Firms will spend untold millions of dollars to build better models than the competition.

We'll use simple linear interpolation to prove the technique.

First we'll define the function which we'll pass the frame to.

```python
def _interp_implied_volatility(options_frame):
    """ Interpolate missing (np.nan) values of implied volatility
    We first need to split the chains into expiration and type because we cannot
    interpolate across the entire chain, rather within these two groups

    :param options_frame: DataFrame containing options data
    :return original DataFrame with ImpliedVolatilityMid column containing interpolated values

    """
    # create a MultiIndex with Expiration, OptionType, the Strike as index, then sort
    frame = options_frame.set_index(["Expiration", "OptionType", "Strike"]).sort_index()

    # pivot the frame with ImpliedVolatilityMid as the values within the table
    # this has Strikes along the rows and Expirations along the columns
    # the level=1 unstack pivots on Expiration and level=0 unstack pivots on OptionType
    unstacked = frame["ImpliedVolatilityMid"].unstack(level=1).unstack(level=0)

    # this line does three things:
    #    first interpolates across each Expiration date down the strikes for np.nan values
    #    second forward fills values which keeps the last interpolated value as the value to fill
    #    third back fills values which keeps the first interpolated value as the value to fill
    unstacked_interp = unstacked.interpolate().ffill().bfill()

    # restack into shape of original DataFrame
    unstacked_interp_indexed = (
        unstacked_interp.stack(level=0).stack(level=0).reset_index()
    )

    # replace old column with the new column with interpolated and filled values
    frame["ImpliedVolatilityMid"] = unstacked_interp_indexed.set_index(
        ["Expiration", "OptionType", "Strike"]
    )

    # give our index back
    frame.reset_index(inplace=True)

    # return
    return frame
```

```python
# get the completed frame
options_frame = _interp_implied_volatility(options_frame)
```

```python
# check to see if there are any np.nans
bad_iv_post = options_frame[np.isnan(options_frame["ImpliedVolatilityMid"])]
```

There should now be no missing data for implied volatility.

```python
bad_iv_post.groupby(["Strike"]).count()["Expiration"]
```

```
Out[207… Series([], Name: Expiration, dtype: int64)
```

Applying Black-Scholes and the greeks

Before we take a look at the implied volatility, let's apply the Black-Scholes formula and the associated greeks. First we define the functions we'll apply to the frame.

```
def _get_option_value(series):
    """ Return the option value given the OptionType

    :param series: row of the dataframe, accessible by label
    :return: Black-Scholes option value

    """
    option_type = series["OptionType"]
    S = series["UnderlyingPrice"]
    K = series["Strike"]
    r = series["InterestRate"]
    t = series["TimeUntilExpiration"]
    vol = series["ImpliedVolatilityMid"]

    meth_name = "black_scholes_{0}_value".format(option_type)
    return float(globals().get(meth_name)(S, K, r, t, vol))


def _get_delta(series):
    """ Return the option delta given the OptionType

    :param series: row of the dataframe, accessible by label
    :return: option delta

    """
    option_type = series["OptionType"]
    S = series["UnderlyingPrice"]
    K = series["Strike"]
    r = series["InterestRate"]
    t = series["TimeUntilExpiration"]
    vol = series["ImpliedVolatilityMid"]

    meth_name = "{0}_delta".format(option_type)
    return float(globals().get(meth_name)(S, K, r, t, vol))


def _get_gamma(series):
    """ Return the option gamma

    :param series: row of the dataframe, accessible by label
    :return: option gamma

    """
    S = series["UnderlyingPrice"]
    K = series["Strike"]
    r = series["InterestRate"]
    t = series["TimeUntilExpiration"]
    vol = series["ImpliedVolatilityMid"]

    return float(gamma(S, K, r, t, vol))
```

```
def _get_vega(series):
    """ Return the option vega

    :param series: row of the dataframe, accessible by label
    :return: option vega

    """
    S = series["UnderlyingPrice"]
    K = series["Strike"]
    r = series["InterestRate"]
    t = series["TimeUntilExpiration"]
    vol = series["ImpliedVolatilityMid"]

    return float(vega(S, K, r, t, vol))


def _get_theta(series):
    """ Return the option theta given the OptionType

    :param series: row of the dataframe, accessible by label
    :return: option theta

    """
    option_type = series["OptionType"]
    S = series["UnderlyingPrice"]
    K = series["Strike"]
    r = series["InterestRate"]
    t = series["TimeUntilExpiration"]
    vol = series["ImpliedVolatilityMid"]

    meth_name = "{0}_theta".format(option_type)
    return float(globals().get(meth_name)(S, K, r, t, vol))


def _get_rho(series):
    """ Return the option rho given the OptionType

    :param series: row of the dataframe, accessible by label
    :return: option rho

    """
    option_type = series["OptionType"]
    S = series["UnderlyingPrice"]
    K = series["Strike"]
    r = series["InterestRate"]
    t = series["TimeUntilExpiration"]
    vol = series["ImpliedVolatilityMid"]

    meth_name = "{0}_rho".format(option_type)
    return float(globals().get(meth_name)(S, K, r, t, vol))
```

```python
def _get_model_error(series):
    """ Return the error between mid price and model price

    :param series: row of the dataframe, accessible by label
    :return: error between mid price and model price

    """
    option_mid = series["Mid"]

    return option_mid - _get_option_value(series)

# use the apply method to pass each row as a series to the various methods, returns a series in this case
options_frame["TheoreticalValue"] = options_frame.apply(_get_option_value, axis=1)
options_frame["Delta"] = options_frame.apply(_get_delta, axis=1)
options_frame["Gamma"] = options_frame.apply(_get_gamma, axis=1)
options_frame["Vega"] = options_frame.apply(_get_vega, axis=1)
options_frame["Theta"] = options_frame.apply(_get_theta, axis=1)
options_frame["Rho"] = options_frame.apply(_get_rho, axis=1)
options_frame["ModelError"] = options_frame.apply(_get_model_error, axis=1)
```

```python
options_frame.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 519 entries, 0 to 518
Data columns (total 29 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Expiration            519 non-null    datetime64[ns]
 1   OptionType            519 non-null    object
 2   Strike                519 non-null    float64
 3   OptionSymbol          519 non-null    object
 4   Last                  519 non-null    float64
 5   Bid                   519 non-null    float64
 6   Ask                   519 non-null    float64
 7   OptionChange          519 non-null    float64
 8   PctChg                519 non-null    object
 9   Volume                519 non-null    int64
 10  OpenInterest          519 non-null    int64
 11  IV                    519 non-null    object
 12  Root                  519 non-null    object
 13  IsNonstandard         519 non-null    bool
 14  UnderlyingSymbol      519 non-null    object
 15  UnderlyingPrice       519 non-null    float64
 16  QuoteDatetime         519 non-null    datetime64[ns]
 17  DaysUntilExpiration   519 non-null    int64
 18  TimeUntilExpiration   519 non-null    float64
 19  InterestRate          519 non-null    float64
 20  Mid                   519 non-null    float64
 21  ImpliedVolatilityMid  519 non-null    float64
 22  TheoreticalValue      519 non-null    float64
 23  Delta                 519 non-null    float64
 24  Gamma                 519 non-null    float64
 25  Vega                  519 non-null    float64
 26  Theta                 519 non-null    float64
 27  Rho                   519 non-null    float64
 28  ModelError            519 non-null    float64
dtypes: bool(1), datetime64[ns](2), float64(17), int64(3), object(6)
memory usage: 114.2+ KB
```

```python
options_frame.head()
```

Out[212…

	Expiration	OptionType	Strike	OptionSymbol	Last	Bid	Ask	OptionChange	PctChg	Vol
0	2016-01-22	call	124.0	IBM160122C00124000	7.03	6.9	7.95	0.00	0.00%	
1	2016-01-22	call	125.0	IBM160122C00125000	9.50	6.5	7.20	0.00	0.00%	
2	2016-01-22	call	126.0	IBM160122C00126000	6.05	5.8	6.30	0.25	4.31%	
3	2016-01-22	call	127.0	IBM160122C00127000	5.40	5.2	5.65	0.00	0.00%	
4	2016-01-22	call	129.0	IBM160122C00129000	4.19	4.0	4.40	-1.71	-28.98%	

5 rows × 29 columns

Analyzing the Model Error

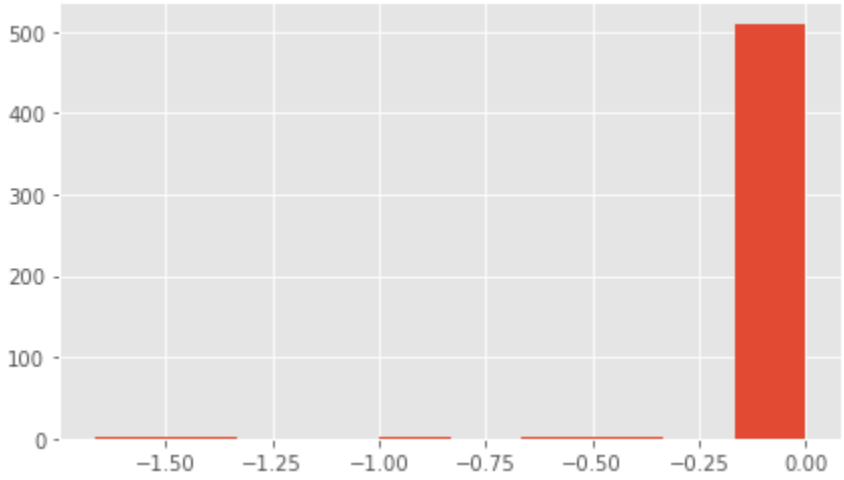
It's useful to visualize some of the data we've created. A quantitative analyst has to essentially be a full stack analyst/developer understanding the implications of the valuation models and metrics. Three years ago this was called quantitative research. Now it's called data science.

We'll start by simply ploting the model error. Because we computed implied volatility from the Black-Scholes formula then used that result to price the option, you may expect there to be no model error.

As you can see from the histogram, this is not the case. In fact there are several occasions where the model error is relatively large.

```
# plot the model error
options_frame["ModelError"].hist(figsize=(7, 4))
```

```
Out[213…  <AxesSubplot:>
```

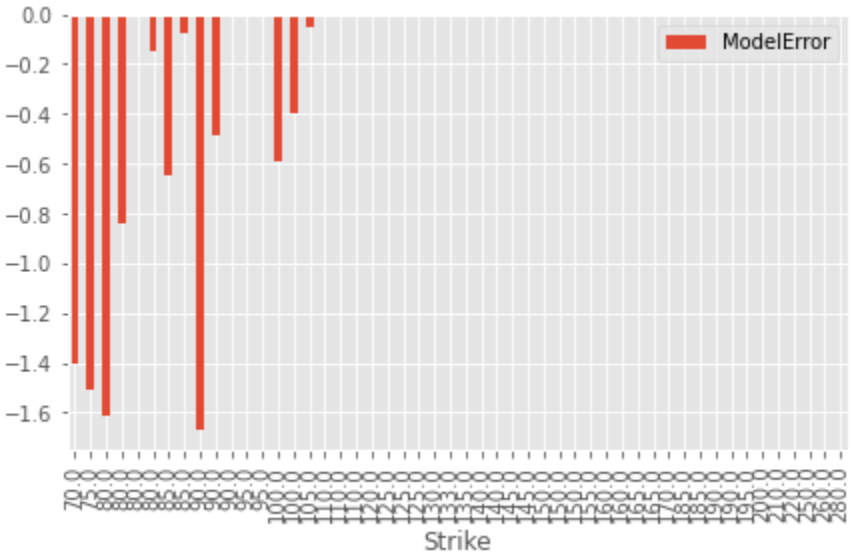


What's the explanation?

Based on some of my comments above, our hypothesis should be that deep in or out of the money options largely have wide bid-ask spreads leading to mis-pricing of the options. Let's plot strike v. model error.

```
# grab the index of the 50 largest abs(errors)
sorted_errors_idx = (
    options_frame["ModelError"].map(abs).sort_values(ascending=False).head(50)
)

# get the rest of the details from the frame
errors_20_largest_by_strike = options_frame.iloc[sorted_errors_idx.index]

# plot model error against strike
errors_20_largest_by_strike[["Strike", "ModelError"]].sort_values(by="Strike").plot(
    kind="bar", x="Strike", figsize=(7, 4)
)
```

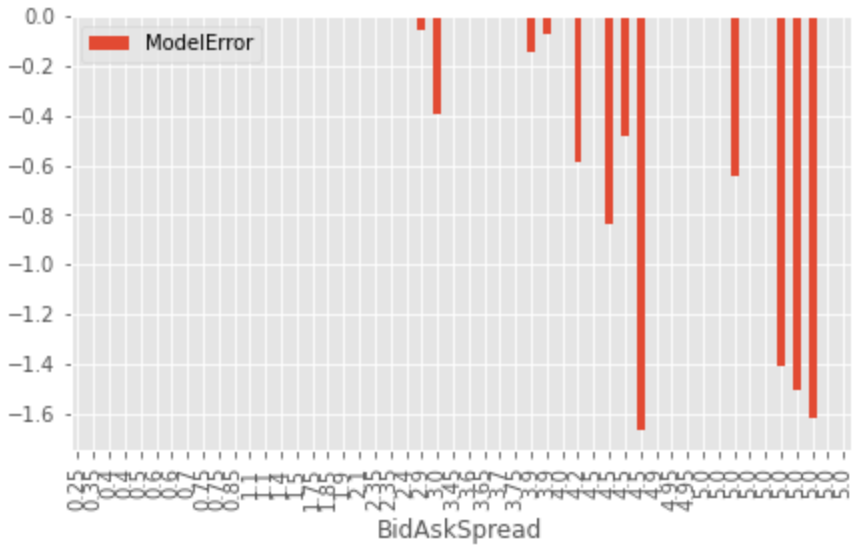
Out[214… <AxesSubplot:xlabel='Strike'>



We can also explore how the bid-ask spread contributes to model error.

```
# add a new column
options_frame["BidAskSpread"] = (options_frame["Ask"] - options_frame["Bid"]).round(4)

# plot model error by bid-ask spread
errors_20_largest_by_spread = options_frame.iloc[sorted_errors_idx.index]

# plot model error against strike, many expirations included
errors_20_largest_by_spread[["BidAskSpread", "ModelError"]].sort_values(
    by="BidAskSpread"
).plot(kind="bar", x="BidAskSpread", figsize=(7, 4))
```

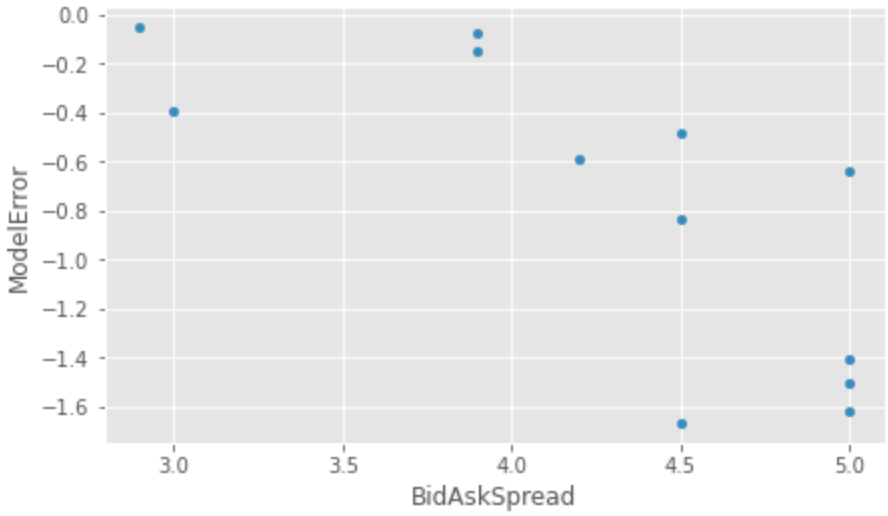
Out[215… <AxesSubplot:xlabel='BidAskSpread'>



Finally, let's take a look at the relationship between model error and bid-ask spread.

```
# plot a scatter plot of all errors > 1.0e-4
options_frame[abs(options_frame["ModelError"]) >= 1.0e-4].plot(
    kind="scatter", x="BidAskSpread", y="ModelError", figsize=(7, 4)
)
```

Out[216… <AxesSubplot:xlabel='BidAskSpread', ylabel='ModelError'>



There are not enough samples to get any sort of statistical relationship but we can clearly see that as the bid-ask spread increases the model error expands as well.

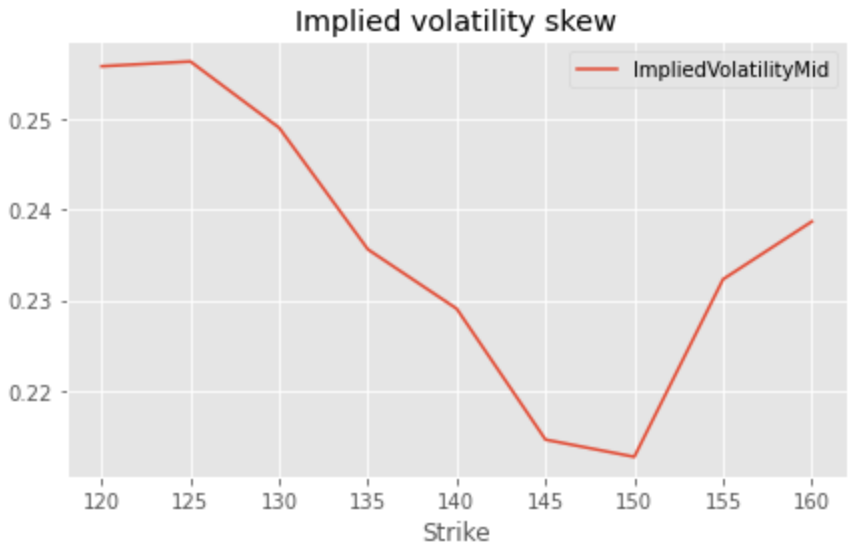
Analyzing Implied Volatility

First we'll look at the so-called implied volatility skew. This plots implied volatility for one expiration across strike prices. Given all else equal, an option with a higher implied volatility will be more expensive. Generally deep in and out of the money strikes command higher prices (and by extension higher implied volatilites) than the Black-Scholes formula implies. This skew only began to appear after the crash of US stocks in October 1987. It is assumed that this is a result of assumptions in the underlying distribution assumption having "fat tails" rather than being completely normal.

Let's take a look.

```
# select an expiration to plot
iv = options_frame[options_frame["Expiration"] == "2016-03-18"]

# get the call options
iv_call = iv[iv["OptionType"] == "call"]

# set the strike as the index so pandas plots nicely
iv_call[["Strike", "ImpliedVolatilityMid"]].set_index("Strike").plot(
    title="Implied volatility skew", figsize=(7, 4)
)
```

```
Out[217… <AxesSubplot:title={'center':'Implied volatility skew'}, xlabel='Strike'>
```



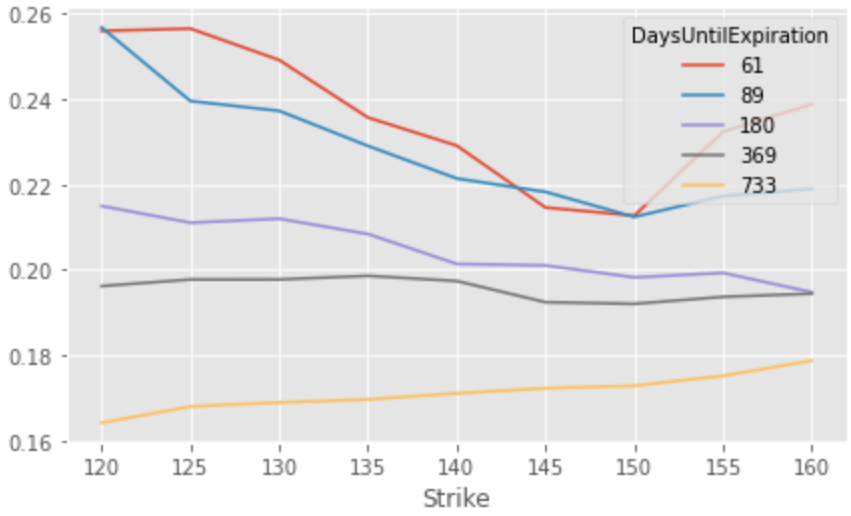
The curve shows that at the money strikes have a higher implied volatiltiy out the money srikes. This is a bit unusual but very likely due to the general state of market volitlity. Traders are pricing at the money call options much more expensive than out the money call options as a result of the demand for call options at the $130 strike.

In a professional setting, traders would use proprietary models (not simple linear interpolation) to rebuild the curves. If the proprietary model resulted in an implied volatility that was higher than the market, a trade would be executed to exploit the anomaly. In our case, we're simply visualizing the prevailing market state.

Another interesting analysis is to look at the multi-expiration skew chart.

```
# get the monthly expirations
expirations = options_frame["Expiration"].unique()[-5:]

# get all the rows where expiration is in our list of expirations
iv_multi = options_frame[options_frame["Expiration"].isin(expirations)]

# get the call options
iv_multi_call = iv_multi[iv_multi["OptionType"] == "call"]

# pivot the data frame to put expiration dates as columns
iv_pivoted = (
    iv_multi_call[["DaysUntilExpiration", "Strike", "ImpliedVolatilityMid"]]
    .pivot(index="Strike", columns="DaysUntilExpiration", values="ImpliedVolatilityMid")
    .dropna()
)

# plot
iv_pivoted.plot(figsize=(7, 4))
```

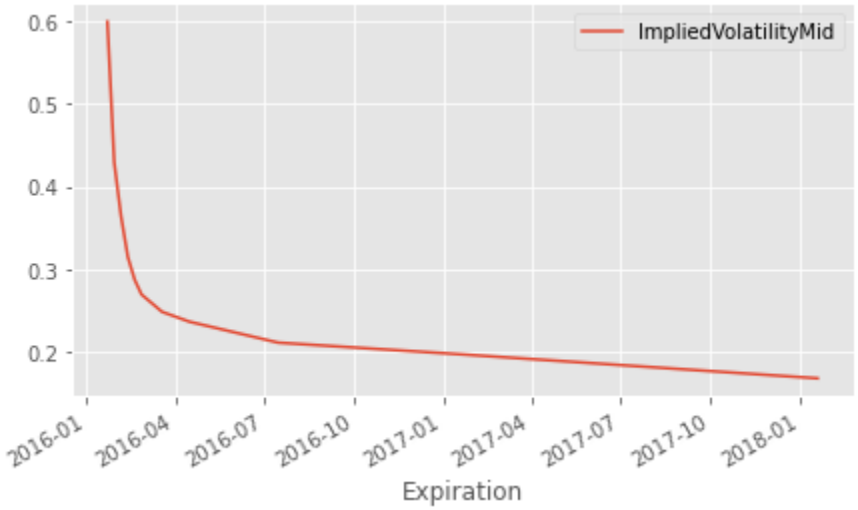
Out[218… <AxesSubplot:xlabel='Strike'>



We'll take a look now at the term structure of volatility. This measures implied volatility at a given strike (usually an at the money strike) but across expirations. Similar to the concept of backwardization and contango in the futures markets, the term structure gives an idea of the demand for options at a particular time in the future.

```
# select a strike to plot
iv = options_frame[options_frame["Strike"] == 130.0]

# get the call options
iv_call = iv[iv["OptionType"] == "call"]

# set the strike as the index so pandas plots nicely
iv_call[["Expiration", "ImpliedVolatilityMid"]].set_index("Expiration").plot(
    figsize=(7, 4)
)
```

Out[219… <AxesSubplot:xlabel='Expiration'>



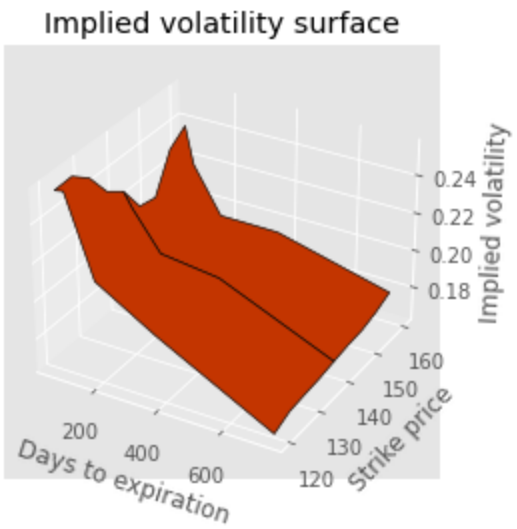
This is a clear demonstration of a violation of the Black-Scholes model assuptions. A flat volatility term structure does not exist in the markets and we have the empirical evidence to prove it.

At the $130 strike, implied is very high at very near expirations. This is a result of the recent market volatility which has increased demand for this option.

One last thing to do is plot the famous volatility surface. This gives a three dimentional representation of the implied volatility against strike price and expiration. While it is sometimes hard to glean any tradeable insight from the surface directly, building surfaces is important to value other derivatives that may not have an exact strike and expiration. This usually happens in the institutional setting however so professional traders can devise their own volatility models and use the implied volatility surface to understand if options are cheap or expensive.

```
# pivot the dataframe
iv_pivoted_surface = iv_multi_call[['DaysUntilExpiration', 'Strike', 'ImpliedVolatilityMid']].pivot(index='

# create the figure object
fig = plt.figure(figsize=(7, 4))

# add the subplot with projection argument
ax = fig.add_subplot(111, projection='3d')

# get the 1d values from the pivoted dataframe
x, y, z = iv_pivoted_surface.columns.values, iv_pivoted_surface.index.values, iv_pivoted_surface.values

# return coordinate matrices from coordinate vectors
X, Y = np.meshgrid(x, y)

# set labels
ax.set_xlabel('Days to expiration')
ax.set_ylabel('Strike price')
ax.set_zlabel('Implied volatility')
ax.set_title('Implied volatility surface')

# plot
ax.plot_surface(X, Y, z, rstride=4, cstride=4, color='orangered', edgecolors='k', lw=0.6)
```

```
Out[220…  <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f94a0925c70>
```



Conclusion

This ebook introduced the concept of implied volatility which is an unobserved, but important, input into pricing models. Before we were able to discuss implied volatility, we needed an introduction to the Black-Scholes model and associated pricing formula. Along the way we reviewed the greeks and understood how the Black-Scholes formula is used practically in the markets.

In addition to the theory and math behind the models, we learned how to code it all in Python relying heavily on `Pandas` to gather and manipulate data and `NumPy` and `SciPy` to implement some of the models.

References

- http://pandas.pydata.org/pandas-docs/stable/index.html
- http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.optimize.brentq.html
- http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.interpolate.interp1d.html
- http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.meshgrid.html
- http://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html
- http://nbviewer.ipython.org/github/quantopian/qgrid/blob/master/qgrid_demo.ipynb
- http://daringfireball.net/projects/markdown/syntax
- http://www.codecogs.com/latex/eqneditor.php
- http://www.bespokeoptions.com/blog/2015/10/06/implied-volatility-with-c-and-python-pt-1
- http://blog.nag.com/2013/10/implied-volatility-using-pythons-pandas.html
- http://www.888options.com
- https://en.wikipedia.org/wiki/Black%E2%80%93Scholes_model
- https://en.wikipedia.org/wiki/Greeks_(finance)
- https://en.wikipedia.org/wiki/Volatility_smile
- http://pythonpodcast.com/scott-sanderson-algorithmic-trading.html
- http://pythonpodcast.com/yves-hilpisch-quant-finance.html

About Me

Hi, I'm Jason.

I studied finance, economics and computer science during my undergraduate coursework in the US. In graduate school I studied quantitative finance and trading, graduating with a Master of Science in Finance. I traded interest rate derivatives for a hedge fund in Chicago, worked for JPMorgan in Chicago, BP in London, a global agricultural trading firm in Singaore, a technology venture capital firm, a commodities trading firm and now Amazon Web Services.

I'm online:

- LinkedIn
- Personal Twitter
- PyQuant News Twitter
- PyQuant News
- Github
- Finance Tools

PyQuant News

Using Python just got easy

The web's best resources for developers using Python for quantitative and scientific analysis.