Portfolio Optimization

“Modern Portfolio Theory (MPT), a hypothesis put forth by Harry Markowitz in his paper “Portfolio Selection,” (published in 1952 by the Journal of Finance) is an investment theory based on the idea that risk-averse investors can construct portfolios to optimize or maximize expected return based on a given level of market risk, emphasizing that risk is an inherent part of higher reward. It is one of the most important and influential economic theories dealing with finance and investment.

Monte Carlo Simulation for Optimization Search

We could randomly try to find the optimal portfolio balance using Monte Carlo simulation

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:
```
# Download and get Daily Returns
aapl = pd.read_csv('AAPL_CLOSE',index_col='Date',parse_dates=True)
cisco = pd.read_csv('CISCO_CLOSE',index_col='Date',parse_dates=True)
ibm = pd.read_csv('IBM_CLOSE',index_col='Date',parse_dates=True)
amzn = pd.read_csv('AMZN_CLOSE',index_col='Date',parse_dates=True)
```

In [3]:
```
stocks = pd.concat([aapl,cisco,ibm,amzn],axis=1)
stocks.columns = ['aapl','cisco','ibm','amzn']
```

In [4]:
```
stocks.head()
```

Out[4]:

Date	aapl	cisco	ibm	amzn
2012-01-03	53.063218	15.752778	160.830881	179.03
2012-01-04	53.348386	16.057180	160.174781	177.51
2012-01-05	53.940658	15.997991	159.415086	177.61
2012-01-06	54.504543	15.938801	157.584912	182.61
2012-01-09	54.418089	16.040268	156.764786	178.56

In [5]:
```
mean_daily_ret = stocks.pct_change(1).mean()
mean_daily_ret
```

Out[5]:
```
aapl     0.000750
cisco    0.000599
ibm      0.000081
amzn     0.001328
dtype: float64
```

In [6]:
```
stocks.pct_change(1).corr()
```

Out[6]:

	aapl	cisco	ibm	amzn
aapl	1.000000	0.301990	0.297498	0.235487
cisco	0.301990	1.000000	0.424672	0.284470
ibm	0.297498	0.424672	1.000000	0.258492
amzn	0.235487	0.284470	0.258492	1.000000

Simulating Thousands of Possible Allocations

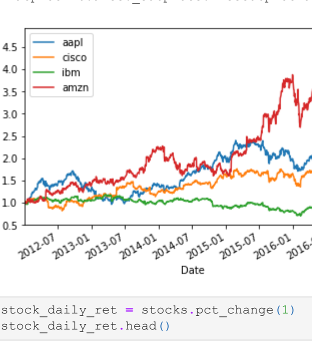
In [7]:
```
stocks.head()
```

Out[7]:

Date	aapl	cisco	ibm	amzn
2012-01-03	53.063218	15.752778	160.830881	179.03
2012-01-04	53.348386	16.057180	160.174781	177.51
2012-01-05	53.940658	15.997991	159.415086	177.61
2012-01-06	54.504543	15.938801	157.584912	182.61
2012-01-09	54.418089	16.040268	156.764786	178.56

In [28]:
```
stock_normed = stocks/stocks.iloc[0]
stock_normed.plot()
```

Out[28]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2258d9c4668>
```



In [29]:
```
stock_daily_ret = stocks.pct_change(1)
stock_daily_ret.head()
```

Out[29]:

Date	aapl	cisco	ibm	amzn
2012-01-03	NaN	NaN	NaN	NaN
2012-01-04	0.005374	0.019324	-0.004079	-0.008490
2012-01-05	0.011102	-0.003686	-0.004743	0.000563
2012-01-06	0.010454	-0.003700	-0.011481	0.028152
2012-01-09	-0.001586	0.006366	-0.005204	-0.022178

Log Returns vs Arithmetic Returns

We will now switch over to using log returns instead of arithmetic returns, for many of our use cases they are almost the same,but most technical analyses require detrending/normalizing the time series and using log returns is a nice way to do that. Log returns are convenient to work with in many of the algorithms we will encounter.

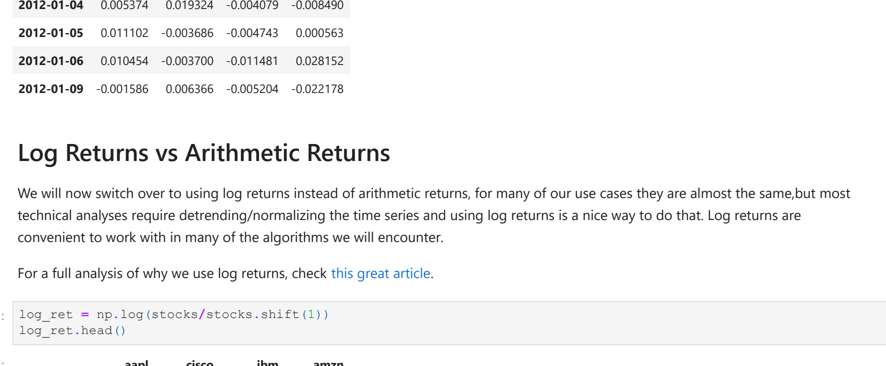
For a full analysis of why we use log returns, check this great article.

In [30]:
```
log_ret = np.log(stocks/stocks.shift(1))
log_ret.head()
```

Out[30]:

Date	aapl	cisco	ibm	amzn
2012-01-03	NaN	NaN	NaN	NaN
2012-01-04	0.005360	0.019139	-0.004088	-0.008526
2012-01-05	0.011041	-0.003693	-0.004754	0.000563
2012-01-06	0.010400	-0.003707	-0.011547	0.027763
2012-01-09	-0.001587	0.006346	-0.005218	-0.022428

In [31]:
```
log_ret.hist(bins=100,figsize=(12,6));
plt.tight_layout()
```



In [32]:
```
log_ret.describe().transpose()
```

Out[32]:

	count	mean	std	min	25%	50%	75%	max
aapl	1257.0	0.000614	0.016466	-0.131875	-0.007358	0.000455	0.009724	0.085022
cisco	1257.0	0.000497	0.014279	-0.116091	-0.006240	0.000213	0.007634	0.118862
ibm	1257.0	0.000011	0.011819	-0.086419	-0.005873	0.000049	0.006477	0.049130
amzn	1257.0	0.001139	0.019362	-0.116503	-0.008534	0.000563	0.011407	0.146225

In [33]:
```
log_ret.mean() * 252
```

Out[33]:
```
aapl     0.154803
cisco    0.125291
ibm      0.002788
amzn     0.287153
dtype: float64
```

In [34]:
```
# Compute pairwise covariance of columns
log_ret.cov()
```

Out[34]:

	aapl	cisco	ibm	amzn
aapl	0.000271	0.000071	0.000057	0.000075
cisco	0.000071	0.000204	0.000072	0.000079
ibm	0.000057	0.000072	0.000140	0.000059
amzn	0.000075	0.000079	0.000059	0.000375

In [35]:
```
log_ret.cov()*252 # multiply by days
```

Out[35]:

	aapl	cisco	ibm	amzn
aapl	0.068326	0.017854	0.014464	0.018986
cisco	0.017854	0.051381	0.018029	0.019956
ibm	0.014464	0.018029	0.035203	0.014939
amzn	0.018986	0.019956	0.014939	0.094470

Single Run for Some Random Allocation

In [36]:
```
# Set seed (optional)
np.random.seed(101)

# Stock Columns
print('Stocks')
print(stocks.columns)
print('\n')

# Create Random Weights
print('Creating Random Weights')
weights = np.array(np.random.random(4))
print(weights)
print('\n')

# Rebalance Weights
print('Rebalance to sum to 1.0')
weights = weights / np.sum(weights)
print(weights)
print('\n')

# Expected Return
print('Expected Portfolio Return')
exp_ret = np.sum(log_ret.mean() * weights) *252
print(exp_ret)
print('\n')

# Expected Variance
print('Expected Volatility')
exp_vol = np.sqrt(np.dot(weights.T, np.dot(log_ret.cov() * 252, weights)))
print(exp_vol)
print('\n')

# Sharpe Ratio
SR = exp_ret/exp_vol
print('Sharpe Ratio')
print(SR)
```

```
Stocks
Index(['aapl', 'cisco', 'ibm', 'amzn'], dtype='object')


Creating Random Weights
[ 0.51639863  0.57066759  0.02847423  0.17152166]


Rebalance to sum to 1.0
[ 0.40122278  0.44338777  0.02212343  0.13326603]


Expected Portfolio Return
0.155992720496 32004


Expected Volatility
0.185026495659


Sharpe Ratio
0.843083148393
```

Great! Now we can just run this many times over!

In [37]:
```
num_ports = 15000

all_weights = np.zeros((num_ports,len(stocks.columns)))
ret_arr = np.zeros(num_ports)
vol_arr = np.zeros(num_ports)
sharpe_arr = np.zeros(num_ports)

for ind in range(num_ports):

    # Create Random Weights
    weights = np.array(np.random.random(4))

    # Rebalance Weights
    weights = weights / np.sum(weights)

    # Save Weights
    all_weights[ind,:] = weights

    # Expected Return
    ret_arr[ind] = np.sum((log_ret.mean() * weights) *252)

    # Expected Variance
    vol_arr[ind] = np.sqrt(np.dot(weights.T, np.dot(log_ret.cov() * 252, weights)))

    # Sharpe Ratio
    sharpe_arr[ind] = ret_arr[ind]/vol_arr[ind]
```

In [38]:
```
sharpe_arr.max()
```

Out[38]:
```
1.0303260551271067
```

In [39]:
```
sharpe_arr.argmax()
```

Out[39]:
```
1419
```

In [40]:
```
all_weights[1419,:]
```

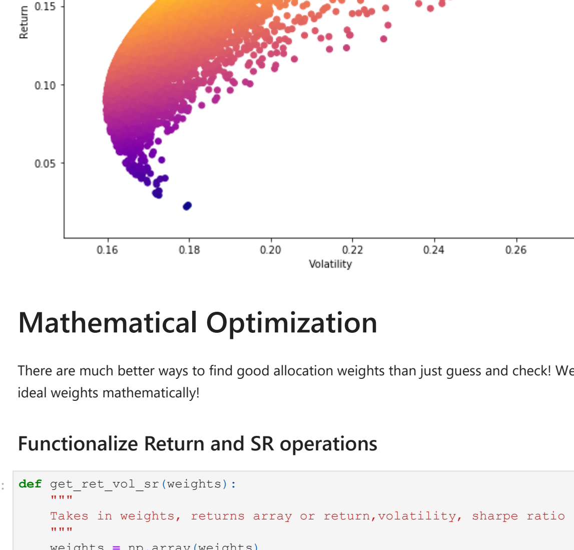
Out[40]:
```
array([ 0.26188068,  0.20759516,  0.00110226,  0.5294219 ])
```

In [41]:
```
max_sr_ret = ret_arr[1419]
max_sr_vol = vol_arr[1419]
```

Plotting the data

In [42]:
```
plt.figure(figsize=(12,8))
plt.scatter(vol_arr,ret_arr,c=sharpe_arr,cmap='plasma')
plt.colorbar(label='Sharpe Ratio')
plt.xlabel('Volatility')
plt.ylabel('Return')

# Add red dot for max SR
plt.scatter(max_sr_vol,max_sr_ret,c='red',s=50,edgecolors='black')
```

Out[42]:
```
<matplotlib.collections.PathCollection at 0x2258e24ac50>
```



Mathematical Optimization

There are much better ways to find good allocation weights than just guess and check! We can use optimization functions to find the ideal weights mathematically!

Functionalize Return and SR operations

In [43]:
```
def get_ret_vol_sr(weights):
    """
    Takes in weights, returns array or return,volatility, sharpe ratio
    """
    weights = np.array(weights)
    ret = np.sum(log_ret.mean() * weights) * 252
    vol = np.sqrt(np.dot(weights.T, np.dot(log_ret.cov() * 252, weights)))
    sr = ret/vol
    return np.array([ret,vol,sr])
```

In [44]:
```
from scipy.optimize import minimize
```

To fully understand all the parameters, check out: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html

In [45]:
```
help(minimize)
```

```
Help on function minimize in module scipy.optimize._minimize:

minimize(fun, x0, args=(), method=None, jac=None, hess=None, hessp=None, bounds=None, constraints=(), tol=None,
callback=None, options=None)
    Minimization of scalar function of one or more variables.

    In general, the optimization problems are of the form::

        minimize f(x) subject to

        g_i(x) >= 0,  i = 1,...,m
        h_j(x)  = 0,  j = 1,...,p

    where x is a vector of one or more variables.
    ``g_i(x)`` are the inequality constraints.
    ``h_j(x)`` are the equality constrains.

    Optionally, the lower and upper bounds for each element in x can also be
    specified using the `bounds` argument.

    Parameters
    ----------
    fun : callable
        Objective function.
    x0 : ndarray
        Initial guess.
    args : tuple, optional
        Extra arguments passed to the objective function and its
        derivatives (Jacobian, Hessian).
    method : str or callable, optional
        Type of solver.  Should be one of

            - 'Nelder-Mead' :ref:`(see here) <optimize.minimize-neldermead>`
            - 'Powell'      :ref:`(see here) <optimize.minimize-powell>`
            - 'CG'          :ref:`(see here) <optimize.minimize-cg>`
            - 'BFGS'        :ref:`(see here) <optimize.minimize-bfgs>`
            - 'Newton-CG'   :ref:`(see here) <optimize.minimize-newtoncg>`
            - 'L-BFGS-B'    :ref:`(see here) <optimize.minimize-lbfgsb>`
            - 'TNC'         :ref:`(see here) <optimize.minimize-tnc>`
            - 'COBYLA'      :ref:`(see here) <optimize.minimize-cobyla>`
            - 'SLSQP'       :ref:`(see here) <optimize.minimize-slsqp>`
            - 'dogleg'      :ref:`(see here) <optimize.minimize-dogleg>`
            - 'trust-ncg'   :ref:`(see here) <optimize.minimize-trustncg>`
            - custom - a callable object (added in version 0.14.0),
              see below for description.

        If not given, chosen to be one of ``BFGS``, ``L-BFGS-B``, ``SLSQP``,
        depending if the problem has constraints or bounds.
    jac : bool or callable, optional
        Jacobian (gradient) of objective function. Only for CG, BFGS,
        Newton-CG, L-BFGS-B, TNC, SLSQP, dogleg, trust-ncg.
        If `jac` is a Boolean and is True, `fun` is assumed to return the
        gradient along with the objective function. If False, the
        gradient will be estimated numerically.
        `jac` can also be a callable returning the gradient of the
        objective. In this case, it must accept the same arguments as `fun`.
    hess, hessp : callable, optional
        Hessian (matrix of second-order derivatives) of objective function or
        Hessian of objective function times an arbitrary vector p.  Only for
        Newton-CG, dogleg, trust-ncg.
        Only one of `hessp` or `hess` needs to be given.  If `hess` is
        provided, then `hessp` will be ignored.  If neither `hess` nor
        `hessp` is provided, then the Hessian product will be approximated
        using finite differences on `jac`. `hessp` must compute the Hessian
        times an arbitrary vector.
    bounds : sequence, optional
        Bounds for variables (only for L-BFGS-B, TNC and SLSQP).
        ``(min, max)`` pairs for each element in ``x``, defining
        the bounds on that parameter. Use None for one of ``min`` or
        ``max`` when there is no bound in that direction.
    constraints : dict or sequence of dict, optional
        Constraints definition (only for COBYLA and SLSQP).
        Each constraint is defined in a dictionary with fields:

            type : str
                Constraint type: 'eq' for equality, 'ineq' for inequality.
            fun : callable
                The function defining the constraint.
            jac : callable, optional
                The Jacobian of `fun` (only for SLSQP).
            args : sequence, optional
                Extra arguments to be passed to the function and Jacobian.

        Equality constraint means that the constraint function result is to
        be zero whereas inequality means that it is to be non-negative.
        Note that COBYLA only supports inequality constraints.
    tol : float, optional
        Tolerance for termination. For detailed control, use solver-specific
        options.
    options : dict, optional
        A dictionary of solver options. All methods accept the following
        generic options:

            maxiter : int
                Maximum number of iterations to perform.
            disp : bool
                Set to True to print convergence messages.

        For method-specific options, see :func:`show_options()`.
    callback : callable, optional
        Called after each iteration, as ``callback(xk)``, where ``xk`` is the
        current parameter vector.

    Returns
    -------
    res : OptimizeResult
        The optimization result represented as a ``OptimizeResult`` object.
        Important attributes are: ``x`` the solution array, ``success`` a
        Boolean flag indicating if the optimizer exited successfully and
        ``message`` which describes the cause of the termination. See
        `OptimizeResult` for a description of other attributes.


    See also
    --------
    minimize_scalar : Interface to minimization algorithms for scalar
        univariate functions
    show_options : Additional options accepted by the solvers

    Notes
    -----
    This section describes the available solvers that can be selected by the
    'method' parameter. The default method is *BFGS*.

    **Unconstrained minimization**

    Method :ref:`Nelder-Mead <optimize.minimize-neldermead>` uses the
    Simplex algorithm [1]_, [2]_. This algorithm is robust in many
    applications. However, if numerical computation of derivative can be
    trusted, other algorithms using the first and/or second derivatives
    information might be preferred for their better performance in
    general.

    Method :ref:`Powell <optimize.minimize-powell>` is a modification
    of Powell's method [3]_, [4]_ which is a conjugate direction
    method. It performs sequential one-dimensional minimizations along
    each vector of the directions set (`direc` field in `options` and
    `info`), which is updated at each iteration of the main
    minimization loop. The function need not be differentiable, and no
    derivatives are taken.

    Method :ref:`CG <optimize.minimize-cg>` uses a nonlinear conjugate
    gradient algorithm by Polak and Ribiere, a variant of the
    Fletcher-Reeves method described in [5]_ pp.  120-122. Only the
    first derivatives are used.

    Method :ref:`BFGS <optimize.minimize-bfgs>` uses the quasi-Newton
    method of Broyden, Fletcher, Goldfarb, and Shanno (BFGS) [5]_
    pp. 136. It uses the first derivatives only. BFGS has proven good
    performance even for non-smooth optimizations. This method also
    returns an approximation of the Hessian inverse, stored as
    `hess_inv` in the OptimizeResult object.

    Method :ref:`Newton-CG <optimize.minimize-newtoncg>` uses a
    Newton-CG algorithm [5]_ pp. 168 (also known as the truncated
    Newton method). It uses a CG method to the compute the search
    direction. See also *TNC* method for a box-constrained
    minimization with a similar algorithm.

    Method :ref:`dogleg <optimize.minimize-dogleg>` uses the dog-leg
    trust-region algorithm [5]_ for unconstrained minimization. This
    algorithm requires the gradient and Hessian; furthermore the
    Hessian is required to be positive definite.

    Method :ref:`trust-ncg <optimize.minimize-trustncg>` uses the
    Newton conjugate gradient trust-region algorithm [5]_ for
    unconstrained minimization. This algorithm requires the gradient
    and either the Hessian or a function that computes the product of
    the Hessian with a given vector.

    **Constrained minimization**

    Method :ref:`L-BFGS-B <optimize.minimize-lbfgsb>` uses the L-BFGS-B
    algorithm [6]_, [7]_ for bound constrained minimization.

    Method :ref:`TNC <optimize.minimize-tnc>` uses a truncated Newton
    algorithm [5]_, [8]_ to minimize a function with variables subject
    to bounds. This algorithm uses gradient information; it is also
    called Newton Conjugate-Gradient. It differs from the *Newton-CG*
    method described above as it wraps a C implementation and allows
    each variable to be given upper and lower bounds.

    Method :ref:`COBYLA <optimize.minimize-cobyla>` uses the
    Constrained Optimization BY Linear Approximation (COBYLA) method
    [9]_, [10]_, [11]_. The algorithm is based on linear
    approximations to the objective function and each constraint. The
    method wraps a FORTRAN implementation of the algorithm. The
    constraints functions 'fun' may return either a single number
    or an array or list of numbers.

    Method :ref:`SLSQP <optimize.minimize-slsqp>` uses Sequential
    Least SQuares Programming to minimize a function of several
    variables with any combination of bounds, equality and inequality
    constraints. The method wraps the SLSQP Optimization subroutine
    originally implemented by Dieter Kraft [12]_. Note that the
    wrapper handles infinite values in bounds by converting them into
    large floating values.

    **Custom minimizers**

    It may be useful to pass a custom minimization method, for example
    when using a frontend to this method such as `scipy.optimize.basinhopping`
    or a different library.  You can simply pass a callable as the ``method``
    parameter.

    The callable is called as ``method(fun, x0, args, **kwargs, **options)``
    where ``kwargs`` corresponds to any other parameters passed to `minimize`
    (such as `callback`, `hess`, etc.), except the `options` dict, which has
    its contents also passed as `method` parameters pair by pair.  Also, if
    `jac` has been passed as a bool type, `jac` and `fun` are mangled so that
    `fun` returns just the function values and `jac` is converted to a function
    returning the Jacobian.  The method shall return an ``OptimizeResult``
    object.

    The provided `method` callable must be able to accept (and possibly ignore)
    arbitrary parameters; the set of parameters accepted by `minimize` may
    expand in future versions and then these parameters will be passed to
    the method.  You can find an example in the scipy.optimize tutorial.

    .. versionadded:: 0.11.0

    References
    ----------
    .. [1] Nelder, J A, and R Mead. 1965. A Simplex Method for Function
        Minimization. The Computer Journal 7: 308-13.
    .. [2] Wright M H. 1996. Direct search methods: Once scorned, now
        respectable, in Numerical Analysis 1995: Proceedings of the 1995
        Dundee Biennial Conference in Numerical Analysis (Eds. D F
        Griffiths and G A Watson). Addison Wesley Longman, Harlow, UK.
        191-208.
    .. [3] Powell, M J D. 1964. An efficient method for finding the minimum of
       a function of several variables without calculating derivatives. The
       Computer Journal 7: 155-162.
    .. [4] Press W, S A Teukolsky, W T Vetterling and B P Flannery.
       Numerical Recipes (any edition), Cambridge University Press.
    .. [5] Nocedal, J, and S J Wright. 2006. Numerical Optimization.
       Springer New York.
    .. [6] Byrd, R H and P Lu and J. Nocedal. 1995. A Limited Memory
       Algorithm for Bound Constrained Optimization. SIAM Journal on
       Scientific and Statistical Computing 16 (5): 1190-1208.
    .. [7] Zhu, C and R H Byrd and J Nocedal. 1997. L-BFGS-B: Algorithm
       778: L-BFGS-B, FORTRAN routines for large scale bound constrained
       optimization. ACM Transactions on Mathematical Software 23 (4):
       550-560.
    .. [8] Nash, S G. Newton-Type Minimization Via the Lanczos Method.
       1984. SIAM Journal of Numerical Analysis 21: 770-778.
    .. [9] Powell, M J D. A direct search optimization method that models
       the objective and constraint functions by linear interpolation.
       1994. Advances in Optimization and Numerical Analysis, eds. S. Gomez
       and J-P Hennart, Kluwer Academic (Dordrecht), 51-67.
    .. [10] Powell M J D. Direct search algorithms for optimization
       calculations. 1998. Acta Numerica 7: 287-336.
    .. [11] Powell M J D. A view of algorithms for optimization without
       derivatives. 2007.Cambridge University Technical Report DAMTP
       2007/NA03
    .. [12] Kraft, D. A software package for sequential quadratic
       programming. 1988. Tech. Rep. DFVLR-FB 88-28, DLR German Aerospace
       Center -- Institute for Flight Mechanics, Koln, Germany.

    Examples
    --------
    Let us consider the problem of minimizing the Rosenbrock function. This
    function (and its respective derivatives) is implemented in `rosen`
    (resp. `rosen_der`, `rosen_hess`) in the `scipy.optimize`.

    >>> from scipy.optimize import minimize, rosen, rosen_der

    A simple application of the *Nelder-Mead* method is:

    >>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
    >>> res = minimize(rosen, x0, method='Nelder-Mead', tol=1e-6)
    >>> res.x
    array([ 1.,  1.,  1.,  1.,  1.])

    Now using the *BFGS* algorithm, using the first derivative and a few
    options:

    >>> res = minimize(rosen, x0, method='BFGS', jac=rosen_der,
    ...                options={'gtol': 1e-6, 'disp': True})
    Optimization terminated successfully.
             Current function value: 0.000000
             Iterations: 26
             Function evaluations: 31
             Gradient evaluations: 31
    >>> res.x
    array([ 1.,  1.,  1.,  1.,  1.])
    >>> print(res.message)
    Optimization terminated successfully.
    >>> res.hess_inv
    array([[ 0.00749589,  0.01255155,  0.02396251,  0.04750988,  0.09495377],  # may vary
           [ 0.01255155,  0.02510441,  0.04794055,  0.09502834,  0.18996269],
           [ 0.02396251,  0.04794055,  0.09631614,  0.19092151,  0.38165151],
           [ 0.04750988,  0.09502834,  0.19092151,  0.38341252,  0.7664427 ],
           [ 0.09495377,  0.18996269,  0.38165151,  0.7664427,   1.53713523]])


    Next, consider a minimization problem with several constraints (namely
    Example 16.4 from [5]_). The objective function is:

    >>> fun = lambda x: (x[0] - 1)**2 + (x[1] - 2.5)**2

    There are three constraints defined as:

    >>> cons = ({'type': 'ineq', 'fun': lambda x:  x[0] - 2 * x[1] + 2},
    ...         {'type': 'ineq', 'fun': lambda x: -x[0] - 2 * x[1] + 6},
    ...         {'type': 'ineq', 'fun': lambda x: -x[0] + 2 * x[1] + 2})

    And variables must be positive, hence the following bounds:

    >>> bnds = ((0, None), (0, None))

    The optimization problem is solved using the SLSQP method as:

    >>> res = minimize(fun, (2, 0), method='SLSQP', bounds=bnds,
    ...                constraints=cons)

    It should converge to the theoretical solution (1.4 ,1.7).
```

Optimization works as a minimization function, since we actually want to maximize the Sharpe Ratio, we will need to turn it negative so we can minimize the negative sharpe (same as maximizing the postive sharpe)

In [46]:
```
def neg_sharpe(weights):
    return  get_ret_vol_sr(weights)[2] * -1
```

In [47]:
```
# Contraints
def check_sum(weights):
    '''
    Returns 0 if sum of weights is 1.0
    '''
    return np.sum(weights) - 1
```

In [48]:
```
# By convention of minimize function it should be a function that returns zero for conditions
cons = ({'type':'eq','fun': check_sum})
```

In [49]:
```
# 0-1 bounds for each weight
bounds = ((0, 1), (0, 1), (0, 1), (0, 1))
```

In [50]:
```
# Initial Guess (equal distribution)
init_guess = [0.25,0.25,0.25,0.25]
```

In [51]:
```
# Sequential Least SQuares Programming (SLSQP).
opt_results = minimize(neg_sharpe,init_guess,method='SLSQP',bounds=bounds,constraints=cons)
```

In [52]:
```
opt_results
```

Out[52]:
```
     fun: -1.030716870334955
     jac: array([  5.64455986e-05,   4.18275595e-05,   3.39921728e-01,
        -4.45097685e-05])
 message: 'Optimization terminated successfully.'
    nfev: 42
     nit: 7
    njev: 7
  status: 0
 success: True
       x: array([  2.66289778e-01,   2.04189819e-01,   9.24621165e-17,
         5.29520404e-01])
```

In [53]:
```
opt_results.x
```

Out[53]:
```
array([  2.66289778e-01,   2.04189819e-01,   9.24621165e-17,
         5.29520404e-01])
```

In [54]:
```
get_ret_vol_sr(opt_results.x)
```

Out[54]:
```
array([ 0.21885915,  0.21233683,  1.03071687])
```

All Optimal Portfolios (Efficient Frontier)

The efficient frontier is the set of optimal portfolios that offers the highest expected return for a defined level of risk or the lowest risk for a given level of expected return. Portfolios that lie below the efficient frontier are sub-optimal, because they do not provide enough return for the level of risk. Portfolios that cluster to the right of the efficient frontier are also sub-optimal, because they have a higher level of risk for the defined rate of return.

Efficient Frontier http://www.investopedia.com/terms/e/efficientfrontier

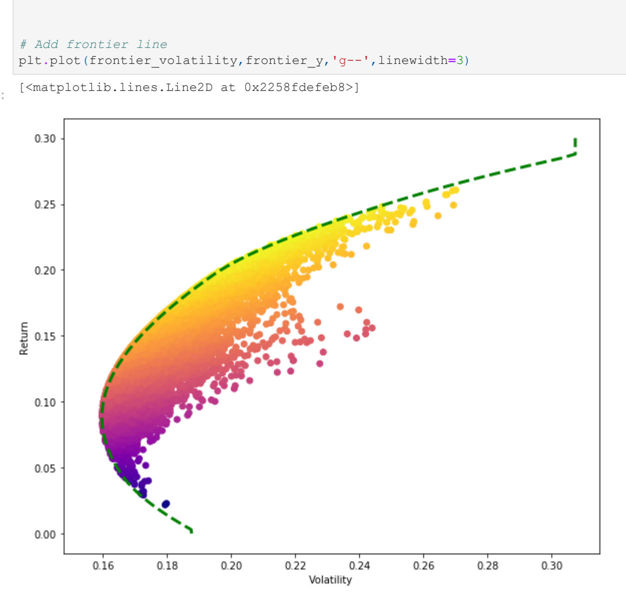
In [55]:
```
# Our returns go from 0 to somewhere along 0.3
# Create a linspace number of points to calculate x on
frontier_y = np.linspace(0,0.3,100) # Change 100 to a lower number for slower computers!
```

In [56]:
```
def minimize_volatility(weights):
    return  get_ret_vol_sr(weights)[1]
```

In [57]:
```
frontier_volatility = []

for possible_return in frontier_y:
    # function for return
    cons = ({'type':'eq','fun': check_sum},
            {'type':'eq','fun': lambda w: get_ret_vol_sr(w)[0] - possible_return})

    result = minimize(minimize_volatility,init_guess,method='SLSQP',bounds=bounds,constraints=cons)

    frontier_volatility.append(result['fun'])
```

In [58]:
```
plt.figure(figsize=(12,8))
plt.scatter(vol_arr,ret_arr,c=sharpe_arr,cmap='plasma')
plt.colorbar(label='Sharpe Ratio')
plt.xlabel('Volatility')
plt.ylabel('Return')



# Add frontier line
plt.plot(frontier_volatility,frontier_y,'g--',linewidth=3)
```

Out[58]:
```
[<matplotlib.lines.Line2D at 0x2258fdefeb8>]
```



Great Job!