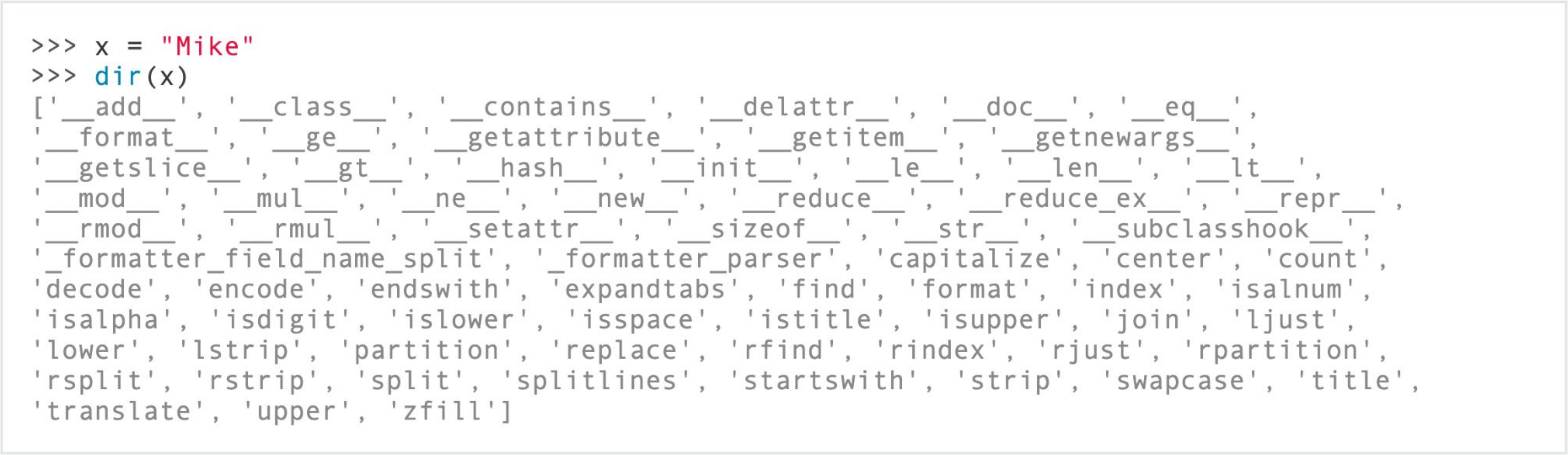
Backtesting 101

Agenda

- Classification tree
- Introduction to Class
- Introduction to backtesting.py
- Application of backtesting.py

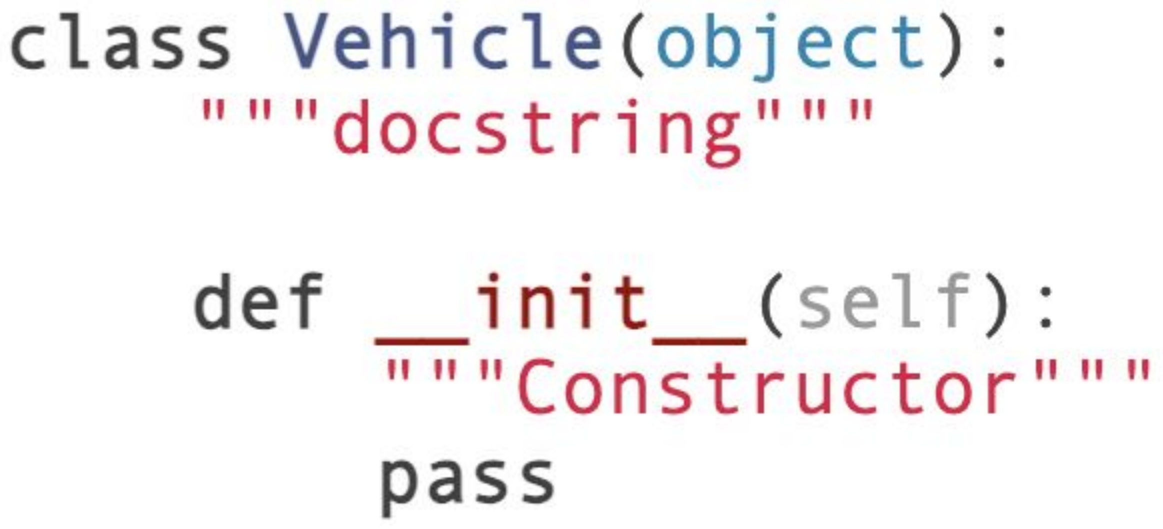
Class

Everything in python is a Class



```
>>> x = "Mike"
>>> dir(x)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'_formatter_field_name_split', '_formatter_parser', 'capitalize', 'center', 'count',
'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum',
'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

Creating a class



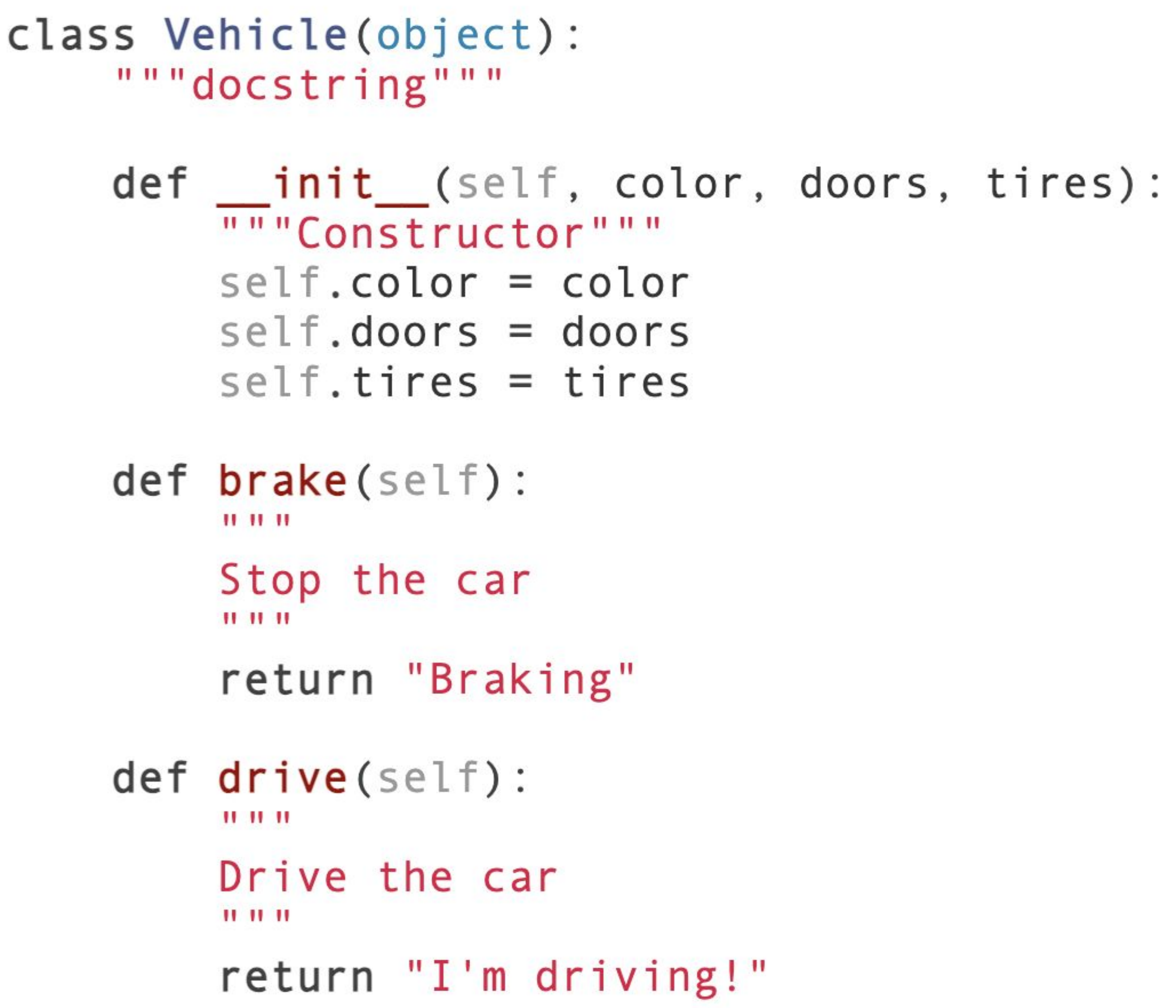
```
class Vehicle(object):
    """docstring"""

    def __init__(self):
        """Constructor"""
        pass
```

- we need to use Python’s **class** keyword, followed by the name of the class
- The class name should have the first letter capitalized.
- The object is what the class is based on or inheriting from. This is known as the base class or parent class
- we don’t need to explicitly say we’re inheriting from object.
- Classes have a special method called __init__ (for initialization). This method is called whenever you create (or instantiate) an object based on this class

Method vs Functions

- A function changes its name to “method” when it is within a class.

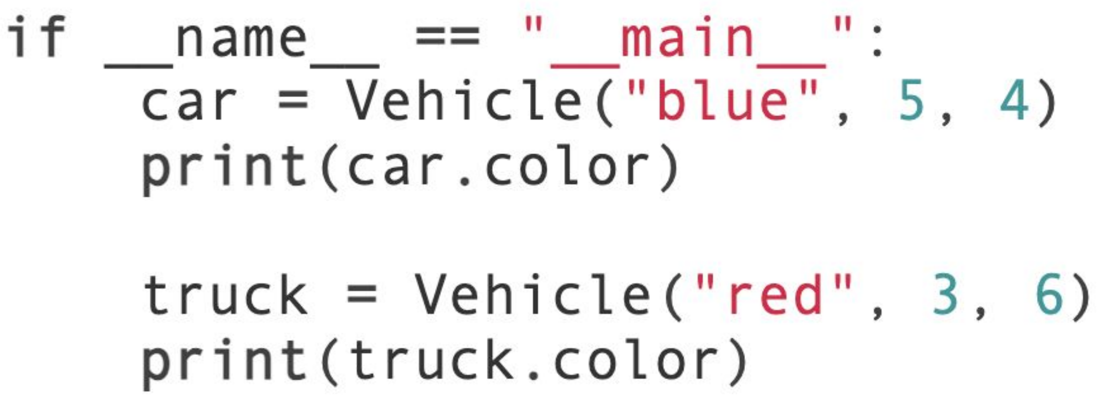
Class attributes & methods



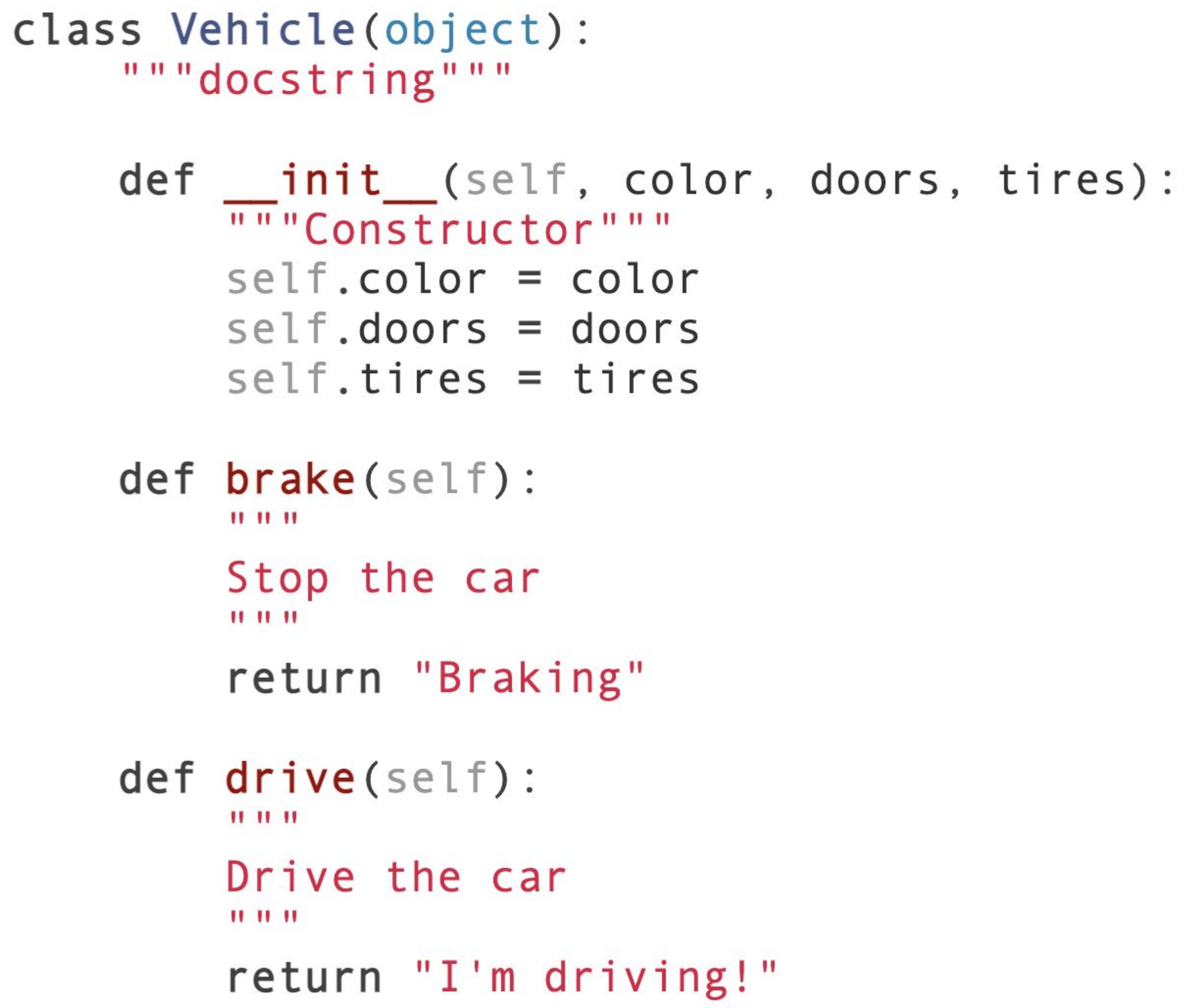
```python
class Vehicle(object):
    """docstring"""

    def __init__(self, color, doors, tires):
        """Constructor"""
        self.color = color
        self.doors = doors
        self.tires = tires

    def brake(self):
        """
        Stop the car
        """
        return "Braking"

    def drive(self):
        """
        Drive the car
        """
        return "I'm driving!"
```

- The code above added three attributes and two methods

What is self?



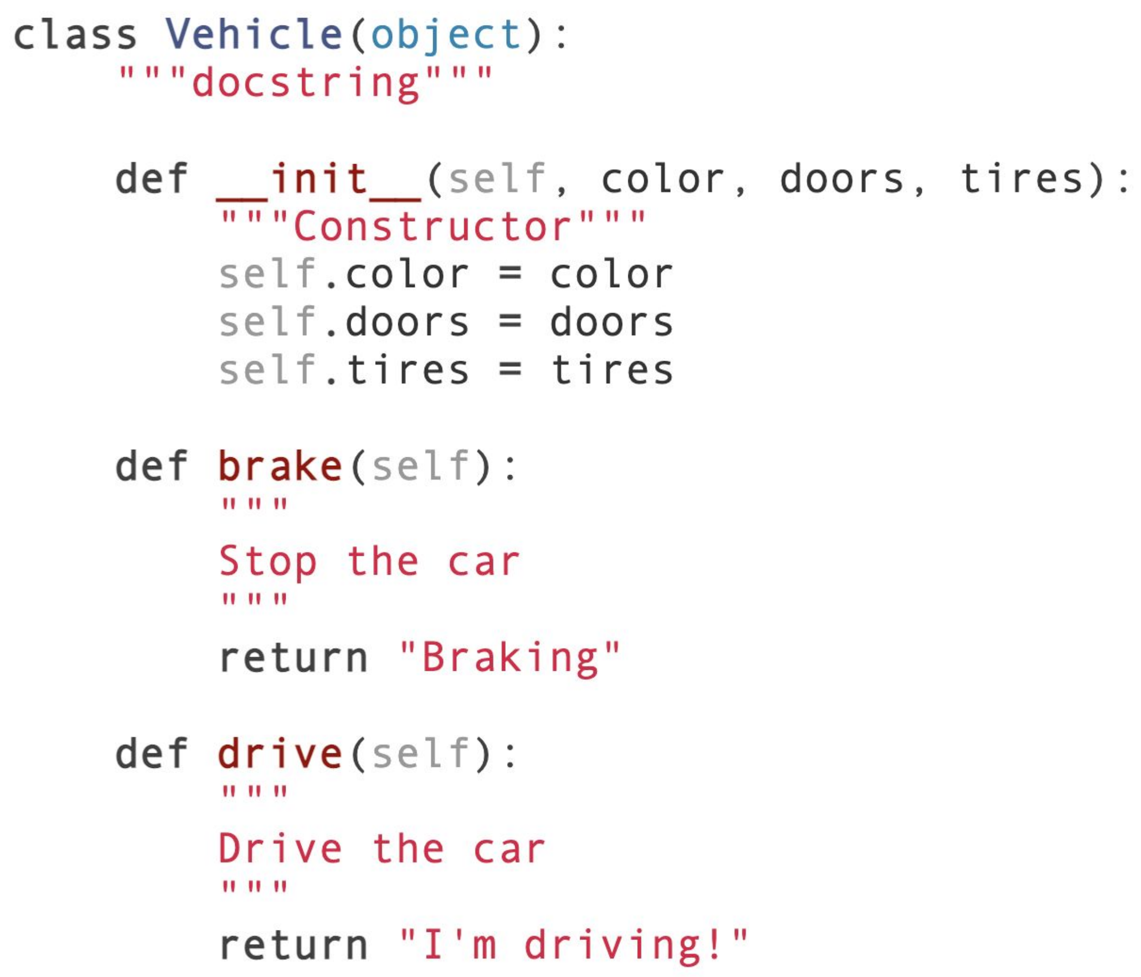
```
if __name__ == "__main__":
    car = Vehicle("blue", 5, 4)
    print(car.color)

    truck = Vehicle("red", 3, 6)
    print(truck.color)
```



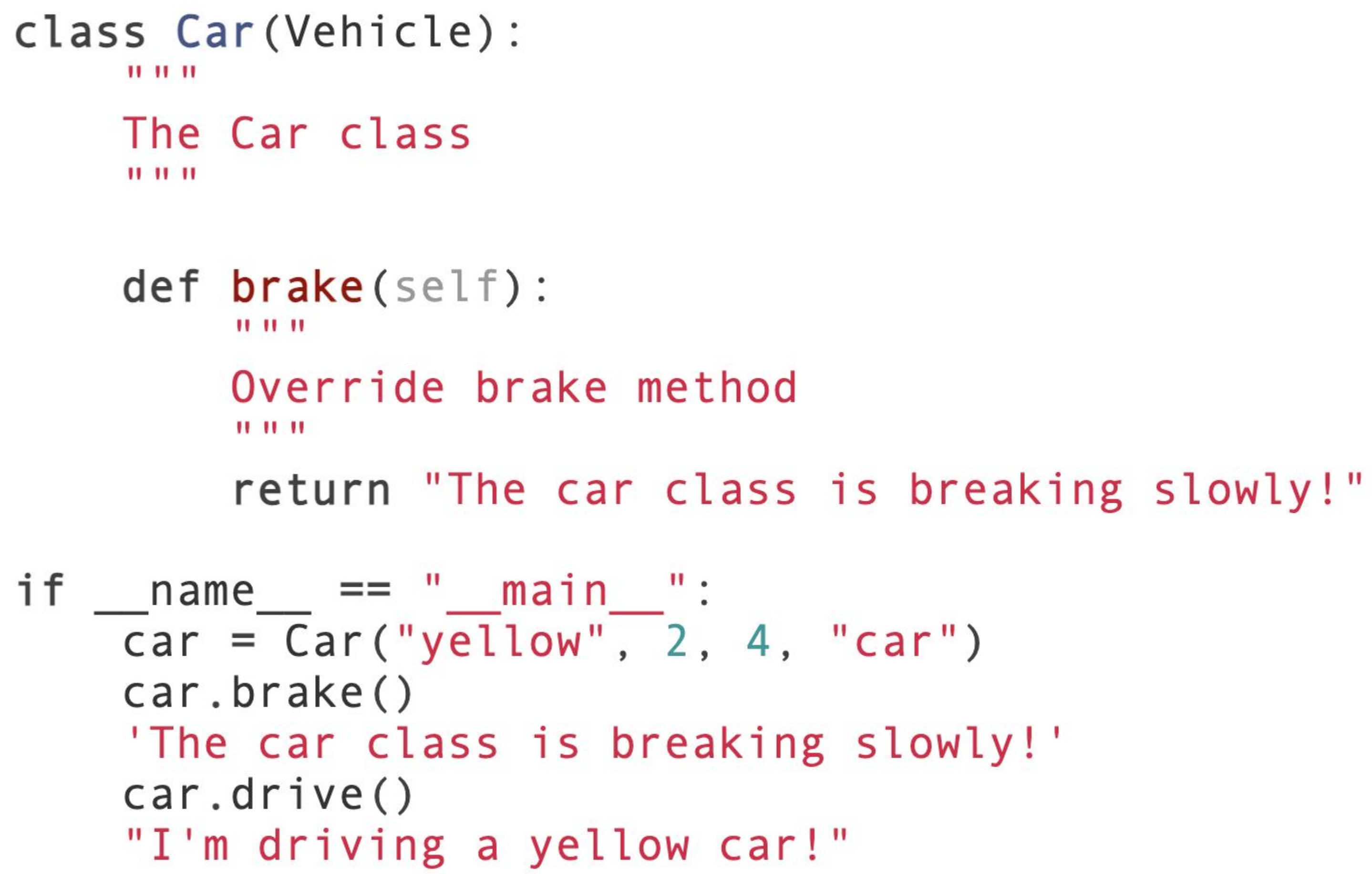
```
class Vehicle(object):
    """docstring"""

    def __init__(self, color, doors, tires):
        """Constructor"""
        self.color = color
        self.doors = doors
        self.tires = tires

    def brake(self):
        """
        Stop the car
        """
        return "Braking"

    def drive(self):
        """
        Drive the car
        """
        return "I'm driving!"
```

- Create two instances of the Vehicle class: a car instance and a truck instance
- Each instance will have its own attributes and methods
- The reason is that the class is using that self argument to tell itself which is which

Subclasses



```
class Vehicle(object):
    """docstring"""

    def __init__(self, color, doors, tires):
        """Constructor"""
        self.color = color
        self.doors = doors
        self.tires = tires

    def brake(self):
        """
        Stop the car
        """
        return "Braking"

    def drive(self):
        """
        Drive the car
        """
        return "I'm driving!"
```



```
class Car(Vehicle):
    """
    The Car class
    """

    def brake(self):
        """
        Override brake method
        """
        return "The car class is breaking slowly!"

if __name__ == "__main__":
    car = Car("yellow", 2, 4, "car")
    car.brake()
    'The car class is breaking slowly!'
    car.drive()
    "I'm driving a yellow car!"
```

- We didn’t include an __init__ method or a drive method. The reason is that when you subclass Vehicle, you get all its attributes and methods unless you override them
- We did override the brake method and made it say something different from the default

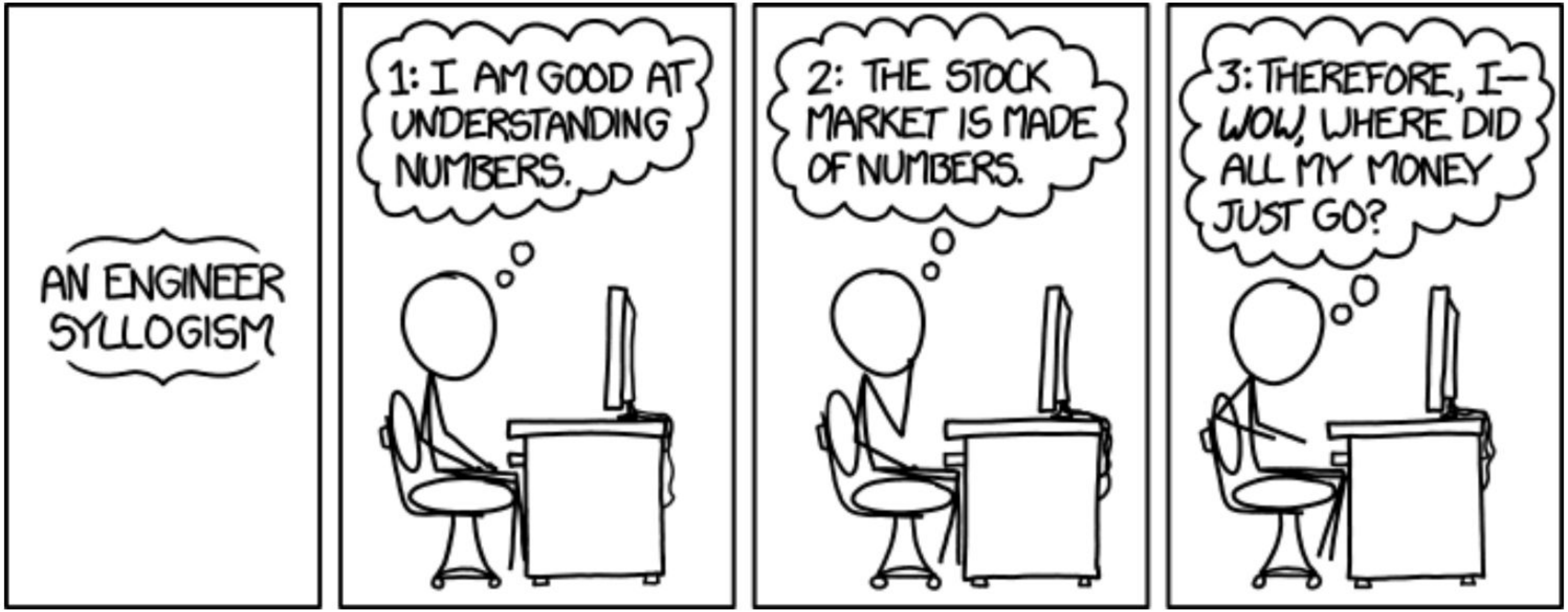
Backtesting.py

What is backtesting.py?

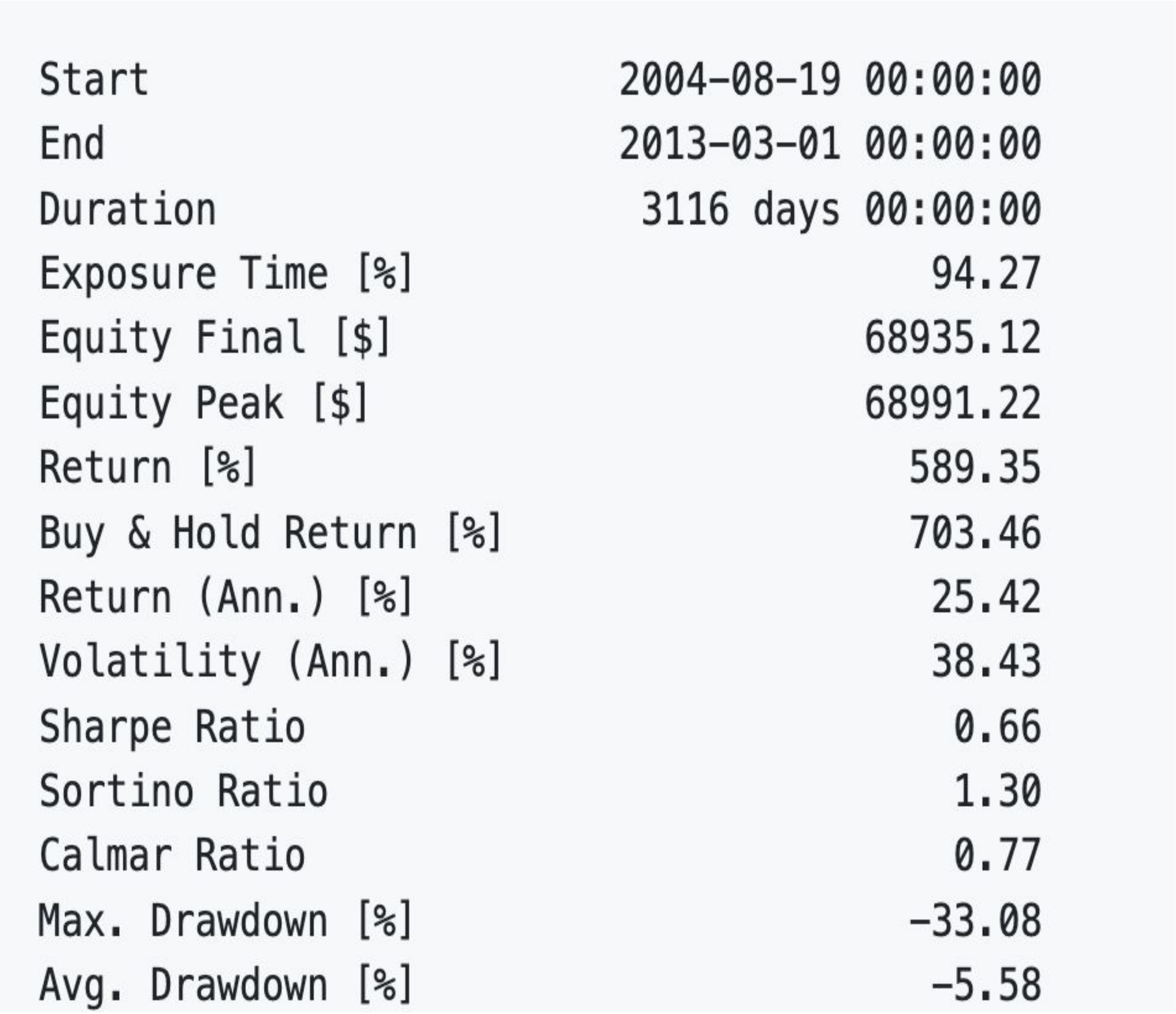
- A python library that allows you to systematically do backtesting with built-in analysis and charts.

- Features
  - Simple, well-documented API
  - Blazing fast execution
  - Built-in optimizer
  - Library of composable base strategies and utilities
  - Supports any financial instrument with candlestick data
  - Detailed results
  - Interactive visualizations

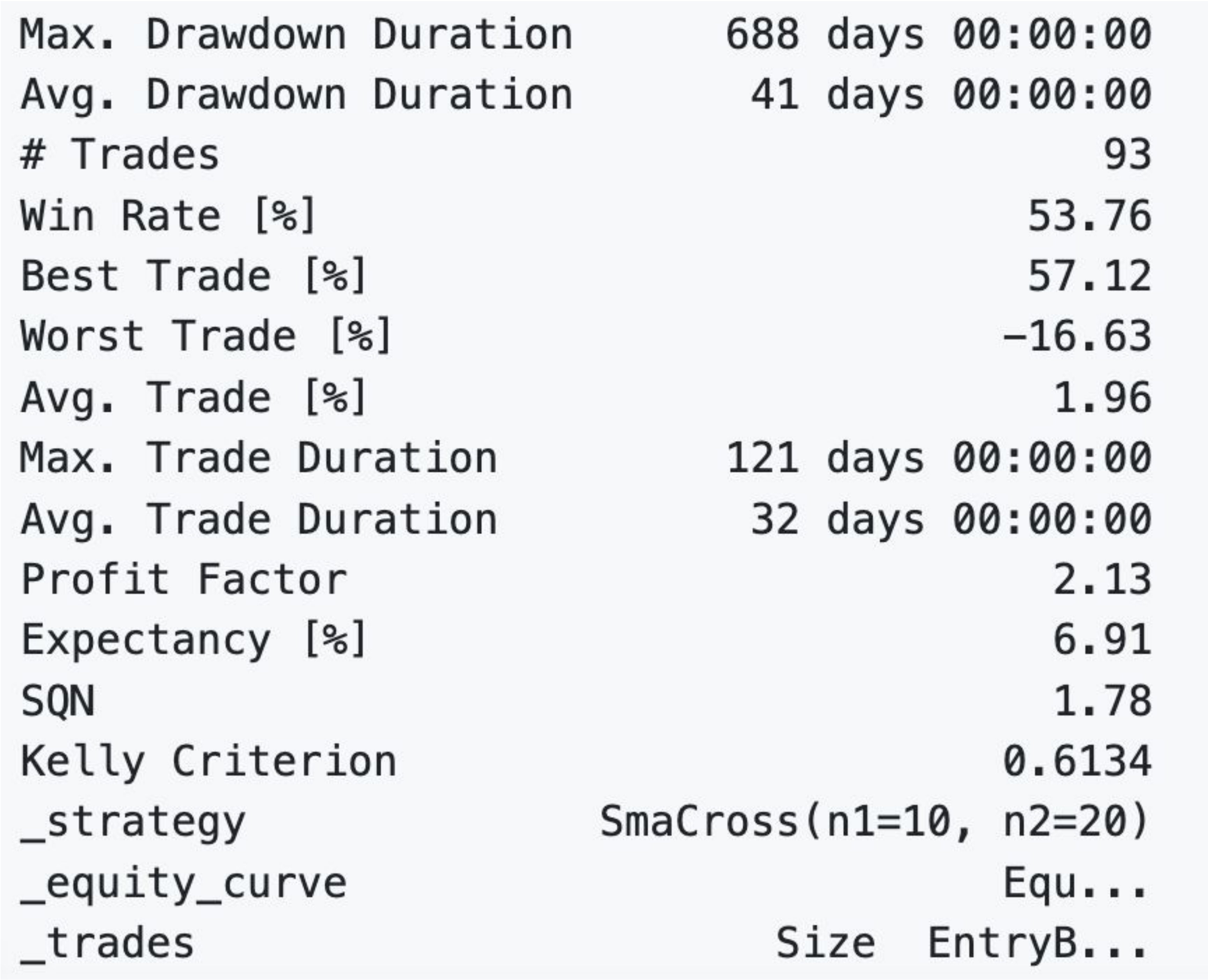
What is backtesting.py?


AN ENGINEER SYLLOGISM
1: I AM GOOD AT UNDERSTANDING NUMBERS.
2: THE STOCK MARKET IS MADE OF NUMBERS.
3: THEREFORE, I— WOW, WHERE DID ALL MY MONEY JUST GO?

Risk Analysis

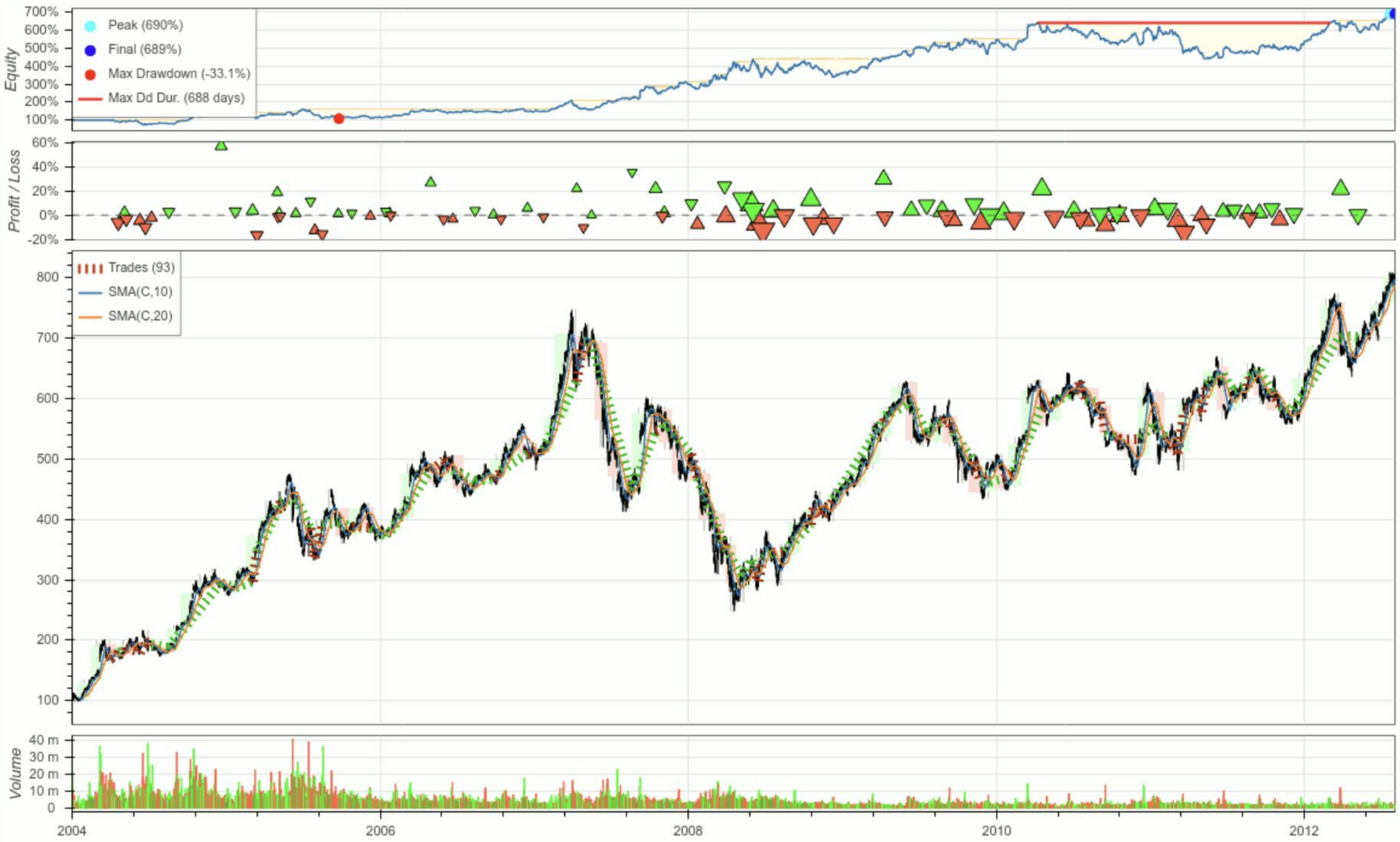


```
Start                     2004-08-19 00:00:00
End                       2013-03-01 00:00:00
Duration                   3116 days 00:00:00
Exposure Time [%]                       94.27
Equity Final [$]                     68935.12
Equity Peak [$]                      68991.22
Return [%]                             589.35
Buy & Hold Return [%]                  703.46
Return (Ann.) [%]                       25.42
Volatility (Ann.) [%]                   38.43
Sharpe Ratio                             0.66
Sortino Ratio                            1.30
Calmar Ratio                             0.77
Max. Drawdown [%]                      -33.08
Avg. Drawdown [%]                       -5.58
```



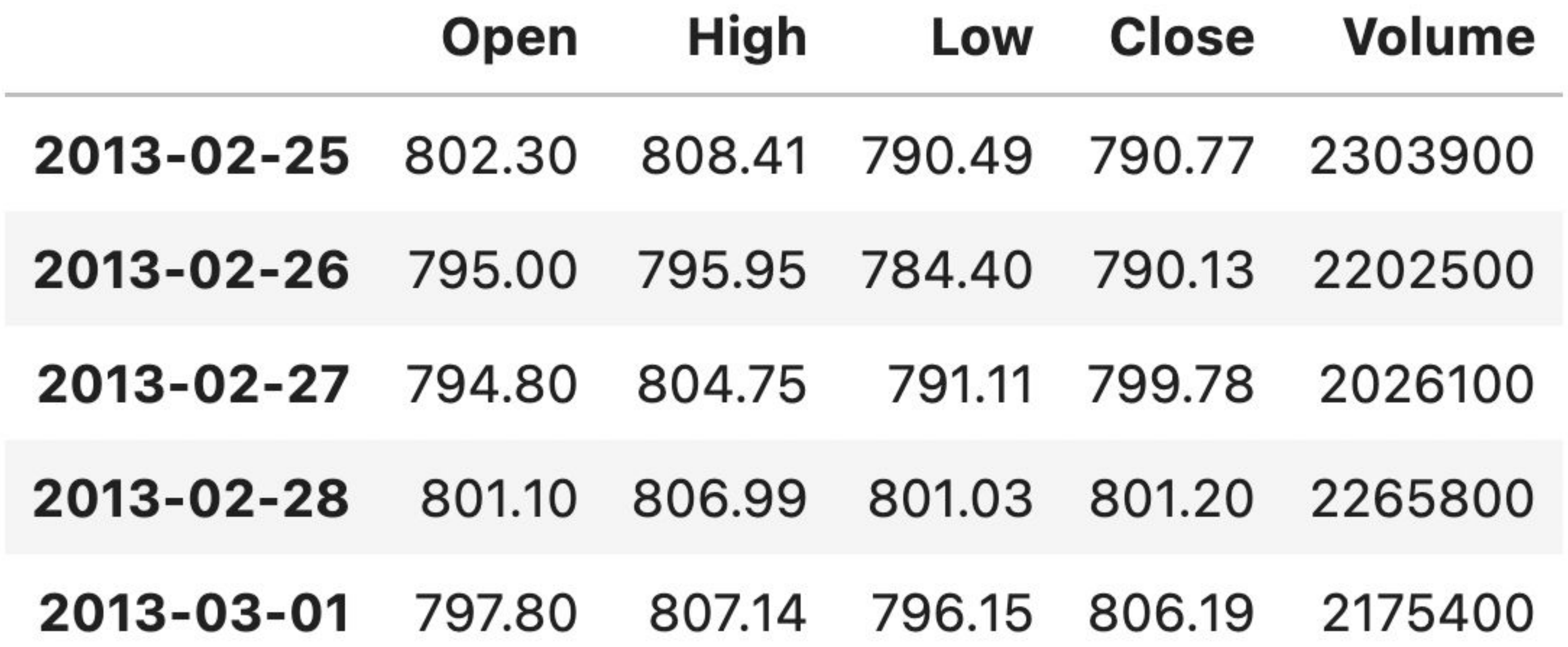
```
Max. Drawdown Duration      688 days 00:00:00
Avg. Drawdown Duration       41 days 00:00:00
# Trades                                   93
Win Rate [%]                            53.76
Best Trade [%]                          57.12
Worst Trade [%]                        -16.63
Avg. Trade [%]                           1.96
Max. Trade Duration         121 days 00:00:00
Avg. Trade Duration          32 days 00:00:00
Profit Factor                            2.13
Expectancy [%]                           6.91
SQN                                      1.78
Kelly Criterion                        0.6134
_strategy                 SmaCross(n1=10, n2=20)
_equity_curve                          Equ...
_trades                       Size  EntryB...
```

Built - in Charts



Data

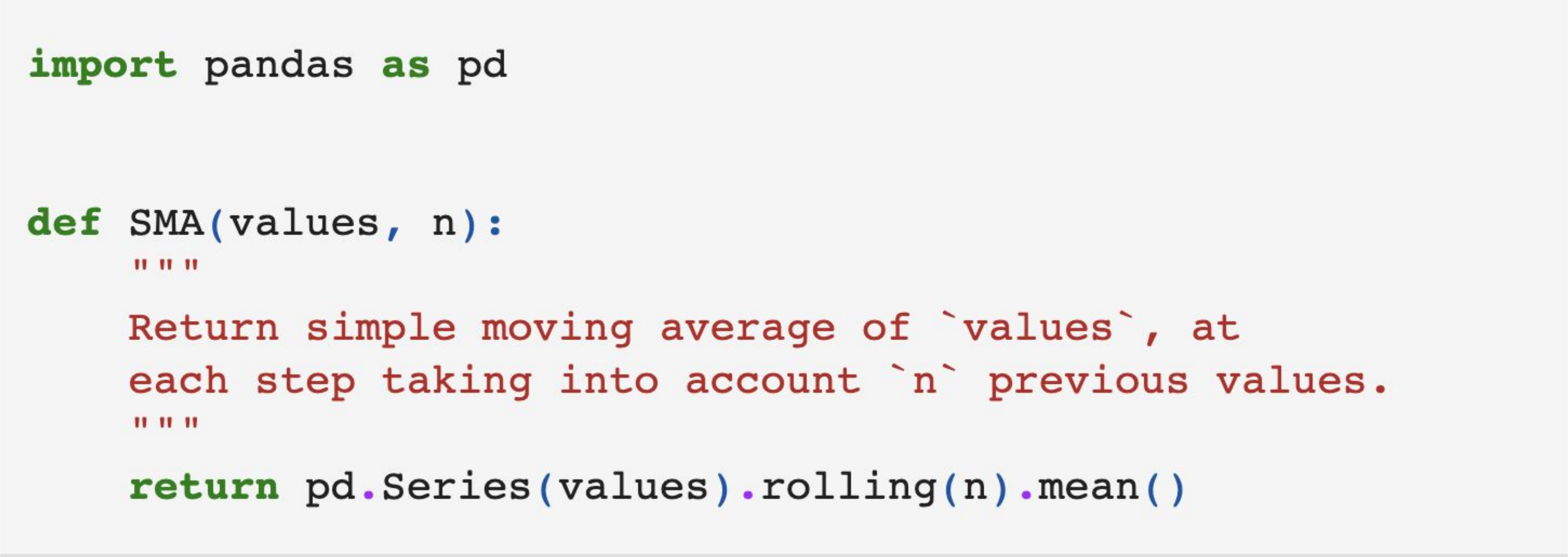
- You bring your own data. Backtesting ingests _all kinds of OHLC data_ (stocks, forex, futures, crypto, ...) as a pandas.DataFrame with columns 'Open', 'High', 'Low', 'Close' and (optionally) 'Volume'.



	Open	High	Low	Close	Volume
2013-02-25	802.30	808.41	790.49	790.77	2303900
2013-02-26	795.00	795.95	784.40	790.13	2202500
2013-02-27	794.80	804.75	791.11	799.78	2026100
2013-02-28	801.10	806.99	801.03	801.20	2265800
2013-03-01	797.80	807.14	796.15	806.19	2175400

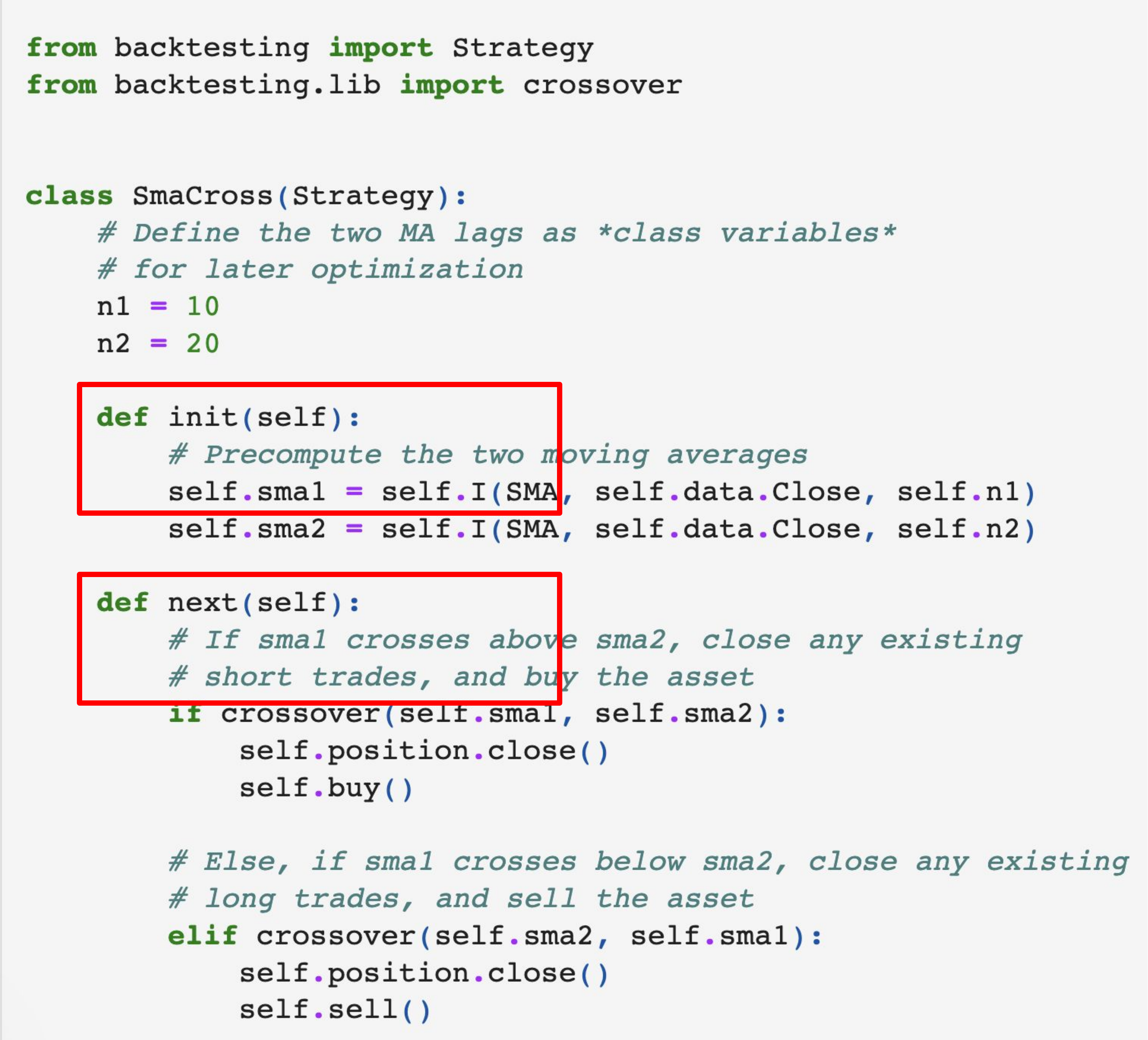
Example: Simple MA Cross-Over Strategy

- You bring your strategy.



```python
import pandas as pd


def SMA(values, n):
    """
    Return simple moving average of `values`, at
    each step taking into account `n` previous values.
    """
    return pd.Series(values).rolling(n).mean()
```

Need to overwrite init and next

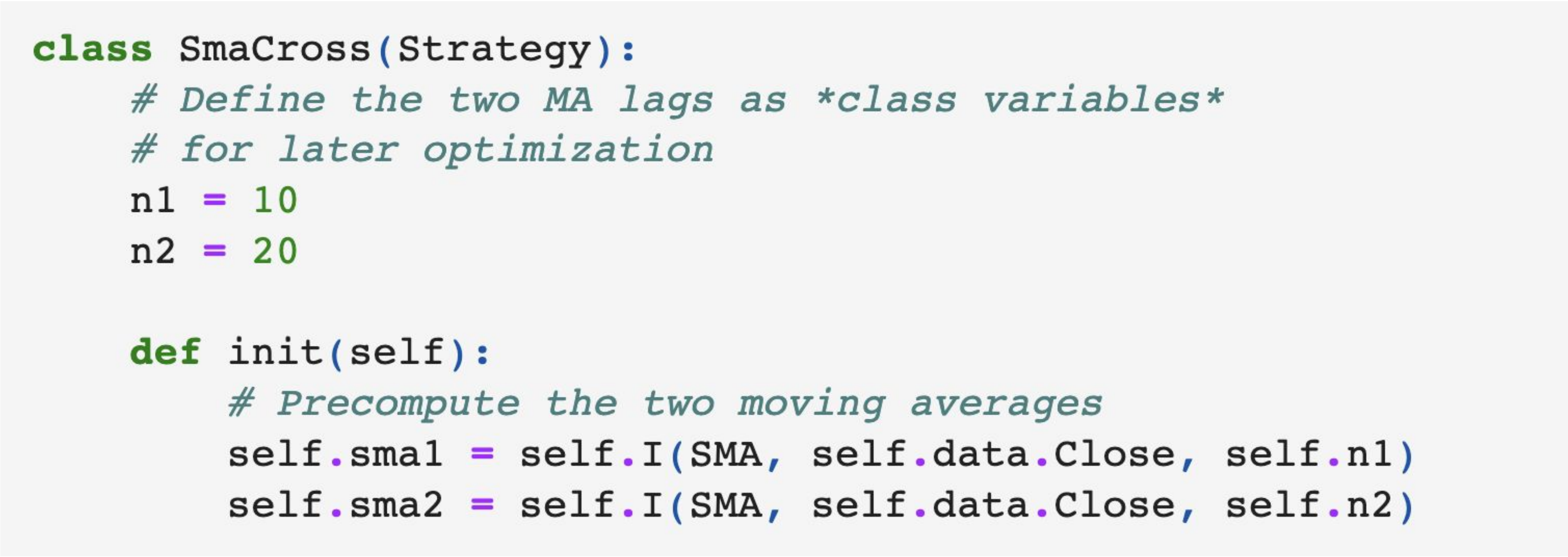


```
from backtesting import Strategy
from backtesting.lib import crossover


class SmaCross(Strategy):
    # Define the two MA lags as *class variables*
    # for later optimization
    n1 = 10
    n2 = 20

    def init(self):
        # Precompute the two moving averages
        self.sma1 = self.I(SMA, self.data.Close, self.n1)
        self.sma2 = self.I(SMA, self.data.Close, self.n2)

    def next(self):
        # If sma1 crosses above sma2, close any existing
        # short trades, and buy the asset
        if crossover(self.sma1, self.sma2):
            self.position.close()
            self.buy()

        # Else, if sma1 crosses below sma2, close any existing
        # long trades, and sell the asset
        elif crossover(self.sma2, self.sma1):
            self.position.close()
            self.sell()
```

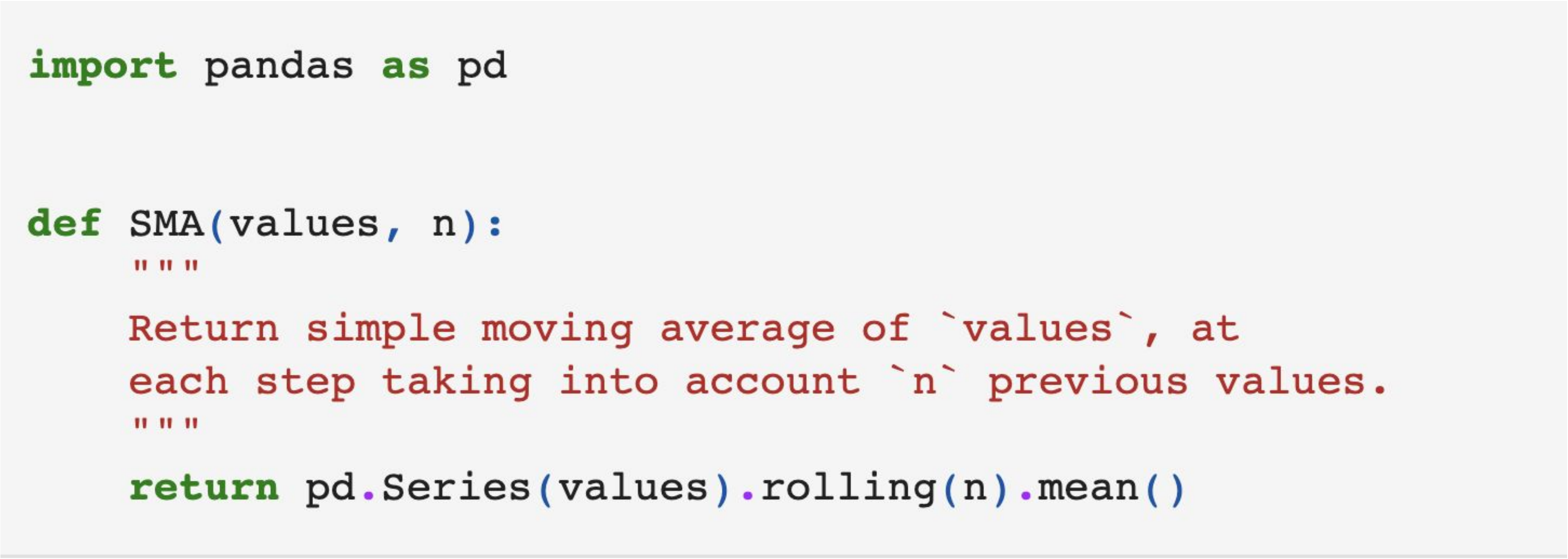
- Method init() is invoked before the strategy is run. Within it, one ideally precomputes in efficient, vectorized manner whatever indicators and signals the strategy depends on.

- Method next() is then iteratively called by the Backtest instance, once for each data point (data frame row), simulating the incremental availability of each new full candlestick bar.

Init



```python
class SmaCross(Strategy):
    # Define the two MA lags as *class variables*
    # for later optimization
    n1 = 10
    n2 = 20

    def init(self):
        # Precompute the two moving averages
        self.sma1 = self.I(SMA, self.data.Close, self.n1)
        self.sma2 = self.I(SMA, self.data.Close, self.n2)
```

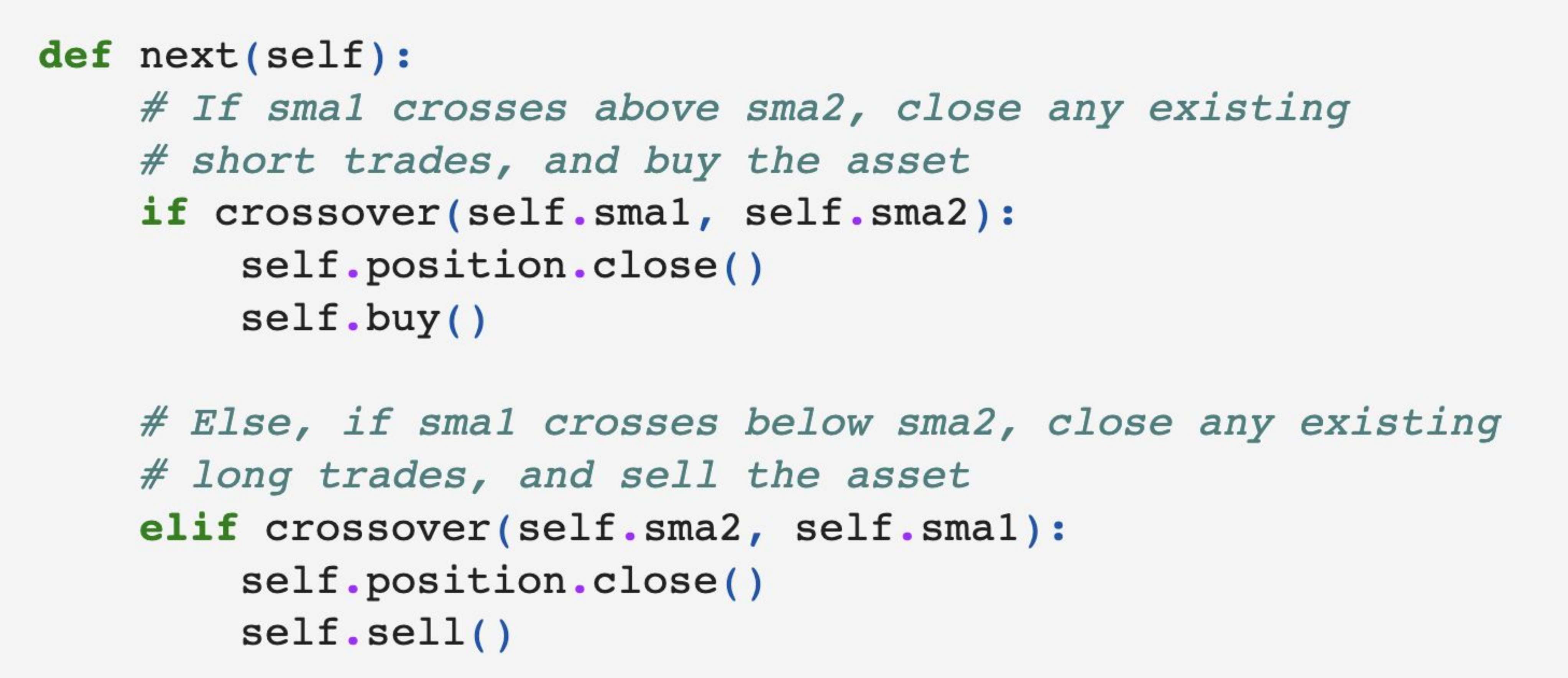


```python
import pandas as pd


def SMA(values, n):
    """
    Return simple moving average of `values`, at
    each step taking into account `n` previous values.
    """
    return pd.Series(values).rolling(n).mean()
```

- we declare and compute indicators indirectly by wrapping them in self.I()

- The wrapper is passed a function (our SMA function) along with any arguments to call it with (our close values and the MA lag)

Next



```python
def next(self):
    # If sma1 crosses above sma2, close any existing
    # short trades, and buy the asset
    if crossover(self.sma1, self.sma2):
        self.position.close()
        self.buy()

    # Else, if sma1 crosses below sma2, close any existing
    # long trades, and sell the asset
    elif crossover(self.sma2, self.sma1):
        self.position.close()
        self.sell()
```

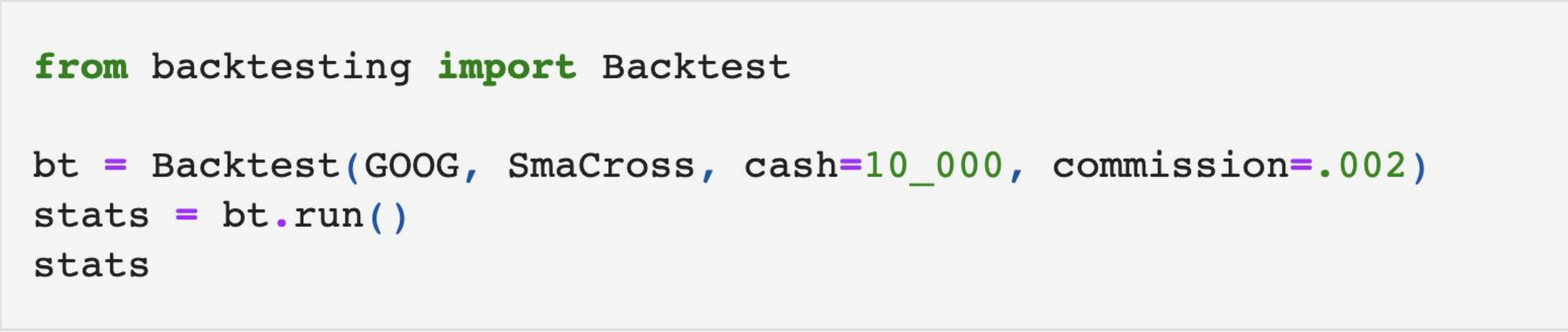
- Check if the faster moving average just crossed over the slower one
- If it did and upwards, we close the possible short position and go long
- if it did and downwards, we close the open long position and go short
- We use backtesting.lib.crossover() function

Init vs Next

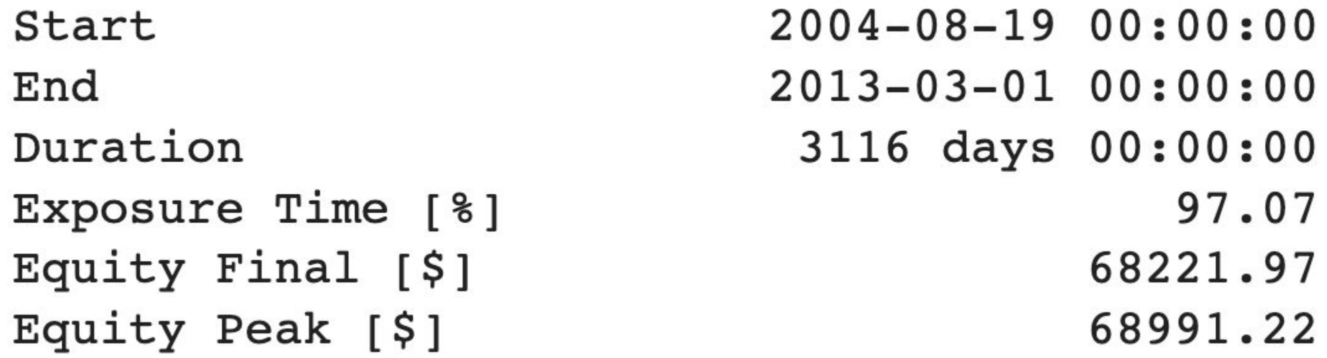
- In init(), the whole series of points was available,

- whereas in next(), the length of self.data and all declared indicators is adjusted on each next() call so that array[-1] (e.g. self.data.Close[-1] or self.sma1[-1]) always contains the most recent value

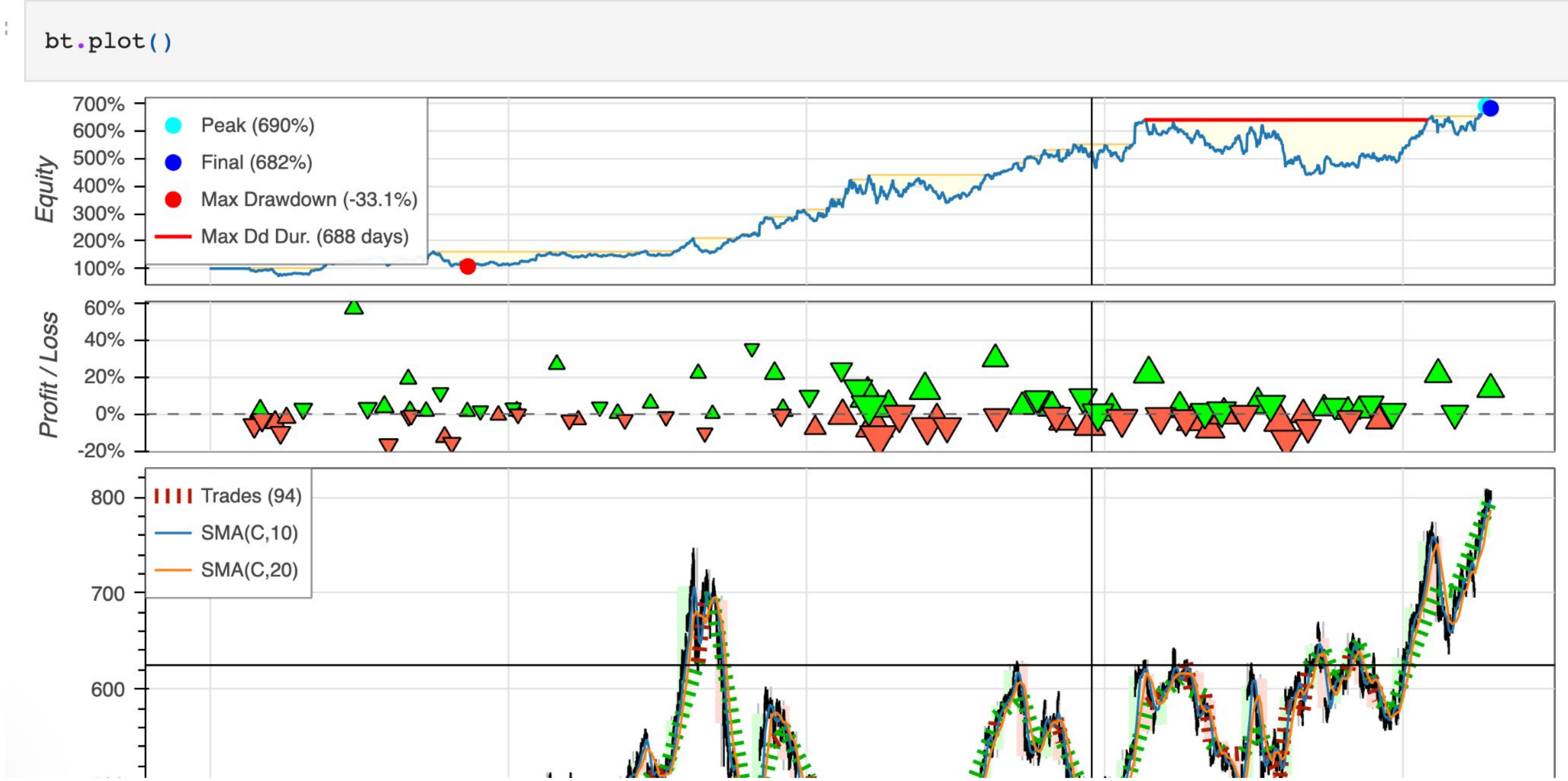
Backtesting



```
from backtesting import Backtest

bt = Backtest(GOOG, SmaCross, cash=10_000, commission=.002)
stats = bt.run()
stats
```

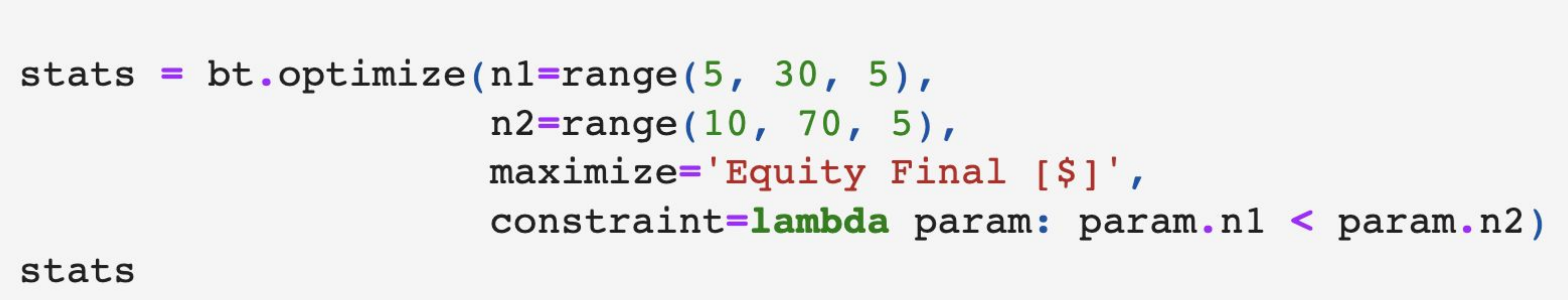


```
Start                     2004-08-19 00:00:00
End                       2013-03-01 00:00:00
Duration                   3116 days 00:00:00
Exposure Time [%]                       97.07
Equity Final [$]                     68221.97
Equity Peak [$]                      68991.22
```



```
bt.plot()
```

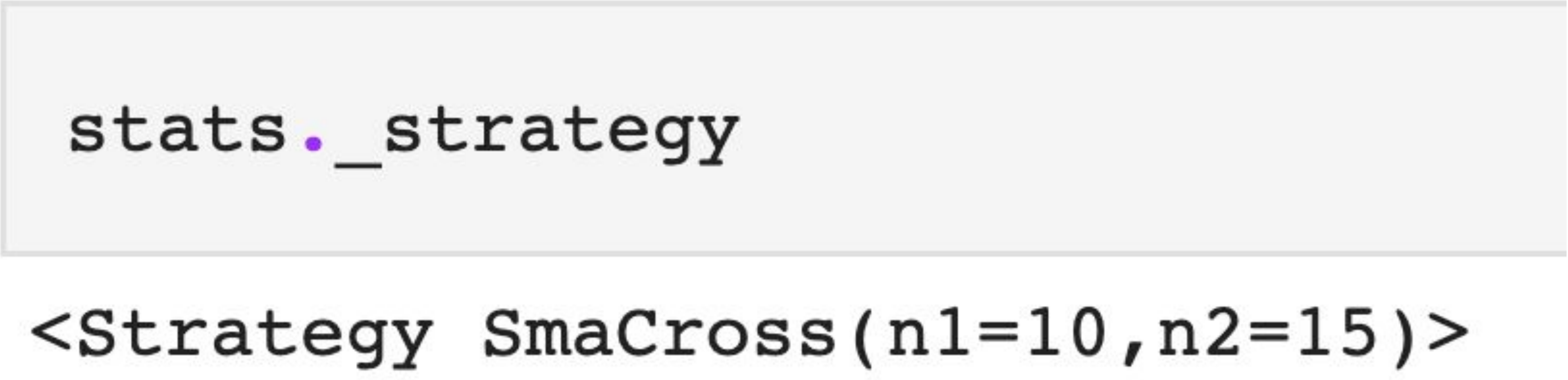
Optimization



```
stats = bt.optimize(n1=range(5, 30, 5),
                    n2=range(10, 70, 5),
                    maximize='Equity Final [$]',
                    constraint=lambda param: param.n1 < param.n2)
stats
```

- Parameter n1 is tested for values in range between 5 and 30 and parameter n2 for values between 10 and 70, respectively

- We limit admissible parameter combinations with an ad hoc constraint function



```
stats._strategy
```

```
<Strategy SmaCross(n1=10,n2=15)>
```