arch (/github/bashtage/arch/tree/main) / examples (/github/bashtage/arch/tree/main/examples)

ARCH Modeling

This setup code is required to run in an IPython notebook

In [1]:

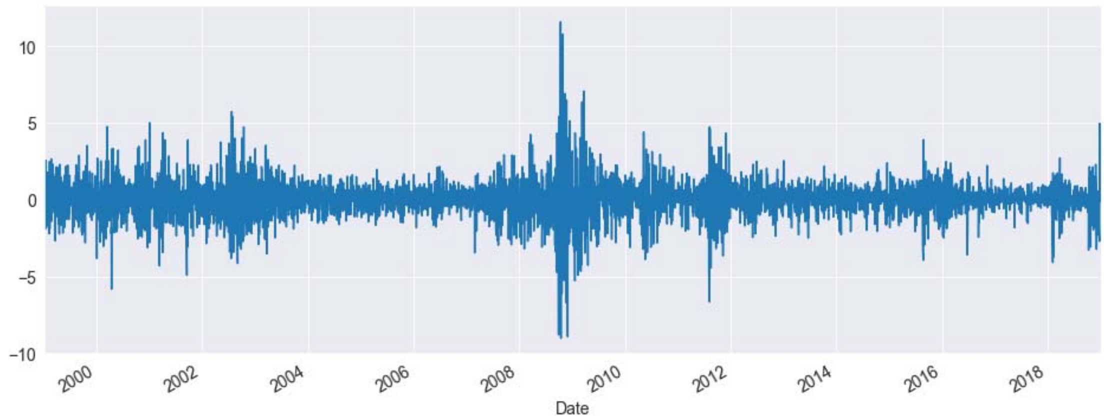
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn

seaborn.set_style("darkgrid")
plt.rc("figure", figsize=(16, 6))
plt.rc("savefig", dpi=90)
plt.rc("font", family="sans-serif")
plt.rc("font", size=14)
```

Setup

These examples will all make use of financial data from Yahoo! Finance. This data set can be loaded from `arch.data.sp500` .

In [2]:

```
import datetime as dt

import arch.data.sp500

st = dt.datetime(1988, 1, 1)
en = dt.datetime(2018, 1, 1)
data = arch.data.sp500.load()
market = data["Adj Close"]
returns = 100 * market.pct_change().dropna()
ax = returns.plot()
xlim = ax.set_xlim(returns.index.min(), returns.index.max())
```



Specifying Common Models

The simplest way to specify a model is to use the model constructor `arch.arch_model` which can specify most common models. The simplest invocation of `arch` will return a model with a constant mean, GARCH(1,1) volatility process and normally distributed errors.

$$
\begin{aligned}
r_t &= \mu + \epsilon_t \\
\sigma_t^2 &= \omega + \alpha\epsilon_{t-1}^2 + \beta\sigma_{t-1}^2 \\
\epsilon_t &= \sigma_t e_t, \;\; e_t \sim N(0,1)
\end{aligned}
$$

The model is estimated by calling `fit`. The optional inputs `iter` controls the frequency of output form the optimizer, and `disp` controls whether convergence information is returned. The results class returned offers direct access to the estimated parameters and related quantities, as well as a `summary` of the estimation results.

GARCH (with a Constant Mean)

The default set of options produces a model with a constant mean, GARCH(1,1) conditional variance and normal errors.

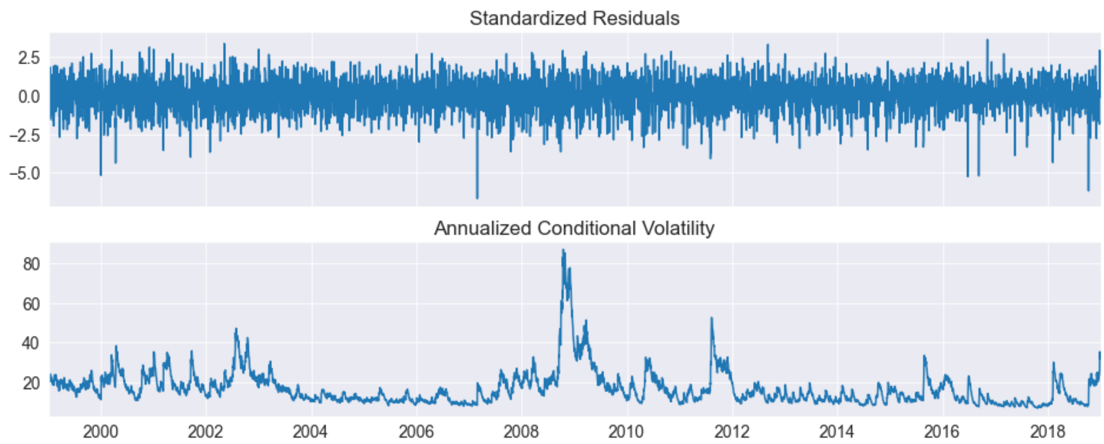
In [3]:

```
from arch import arch_model

am = arch_model(returns)
res = am.fit(update_freq=5)
print(res.summary())
```

```
Iteration:      5,   Func. Count:     35,   Neg. LLF: 6970.2782862
Iteration:     10,   Func. Count:     63,   Neg. LLF: 6936.7184774
Optimization terminated successfully    (Exit mode 0)
            Current function value: 6936.718476988966
            Iterations: 11
            Function evaluations: 68
```

`plot()` can be used to quickly visualize the standardized residuals and conditional volatility.

In [4]:
```
fig = res.plot(annualize="D")
```



GJR-GARCH

Additional inputs can be used to construct other models. This example sets `o` to 1, which includes one lag of an asymmetric shock which transforms a GARCH model into a GJR-GARCH model with variance dynamics given by

$$\sigma_t^2 = \omega + \alpha\epsilon_{t-1}^2 + \gamma\epsilon_{t-1}^2 I_{[\epsilon_{t-1}<0]} + \beta\sigma_{t-1}^2$$

where I is an indicator function that takes the value 1 when its argument is true.

The log likelihood improves substantially with the introduction of an asymmetric term, and the parameter estimate is highly significant.

In [5]:
```
am = arch_model(returns, p=1, o=1, q=1)
res = am.fit(update_freq=5, disp="off")
print(res.summary())
```

```
                   Constant Mean - GJR-GARCH Model Results
==================================================================
Dep. Variable:              Adj Close   R-squared:
Mean Model:             Constant Mean   Adj. R-squared:
Vol Model:                  GJR-GARCH   Log-Likelihood:
Distribution:                  Normal   AIC:
Method:            Maximum Likelihood   BIC:
                                        No. Observations:
Date:                Tue, Mar 09 2021   Df Residuals:
Time:                        12:03:19   Df Model:
                                  Mean Model
==================================================================
                 coef    std err          t      P>|t|       95.0%
```

TARCH/ZARCH

TARCH (also known as ZARCH) model the *volatility* using absolute values. This model is specified using `power=1.0` since the default power, 2, corresponds to variance processes that evolve in squares.

The volatility process in a TARCH model is given by

$$\sigma_t = \omega + \alpha \left|\epsilon_{t-1}\right| + \gamma \left|\epsilon_{t-1}\right| I_{[\epsilon_{t-1}<0]} + \beta\sigma_{t-1}$$

More general models with other powers (κ) have volatility dynamics given by

$$\sigma_t^\kappa = \omega + \alpha\left|\epsilon_{t-1}\right|^\kappa + \gamma\left|\epsilon_{t-1}\right|^\kappa I_{[\epsilon_{t-1}<0]} + \beta\sigma_{t-1}^\kappa$$

where the conditional variance is $\left(\sigma_t^\kappa\right)^{2/\kappa}$.

The TARCH model also improves the fit, although the change in the log likelihood is less dramatic.

In [6]:

```
am = arch_model(returns, p=1, o=1, q=1, power=1.0)
res = am.fit(update_freq=5)
print(res.summary())
```

```
Iteration:      5,   Func. Count:     45,   Neg. LLF: 6828.9328119
Iteration:     10,   Func. Count:     79,   Neg. LLF: 6799.1786845
Optimization terminated successfully    (Exit mode 0)
            Current function value: 6799.1785211175975
            Iterations: 14
            Function evaluations: 102
            Gradient evaluations: 14
                  Constant Mean - TARCH/ZARCH Model Results
==================================================================
Dep. Variable:              Adj Close   R-squared:
Mean Model:             Constant Mean   Adj. R-squared:
Vol Model:                TARCH/ZARCH   Log-Likelihood:
Distribution:                  Normal   AIC:
```

Student's T Errors

Financial returns are often heavy tailed, and a Student's T distribution is a simple method to capture this feature. The call to `arch` changes the distribution from a Normal to a Students's T.

The standardized residuals appear to be heavy tailed with an estimated degree of freedom near 10. The log-likelihood also shows a large increase.

In [7]:

```
am = arch_model(returns, p=1, o=1, q=1, power=1.0, dist="StudentsT
res = am.fit(update_freq=5)
print(res.summary())
```

```
Iteration:      5,   Func. Count:     50,   Neg. LLF: 6729.0422428
Iteration:     10,   Func. Count:     90,   Neg. LLF: 6722.1511847
Optimization terminated successfully    (Exit mode 0)
            Current function value: 6722.151184733061
            Iterations: 12
            Function evaluations: 103
            Gradient evaluations: 11
                     Constant Mean - TARCH/ZARCH Model Results
```

Fixing Parameters

In some circumstances, fixed rather than estimated parameters might be of interest. A model-result-like class can be generated using the `fix()` method. The class returned is identical to the usual model result class except that information about inference (standard errors, t-stats, etc) is not available.

In the example, I fix the parameters to a symmetric version of the previously estimated model.

In [8]:

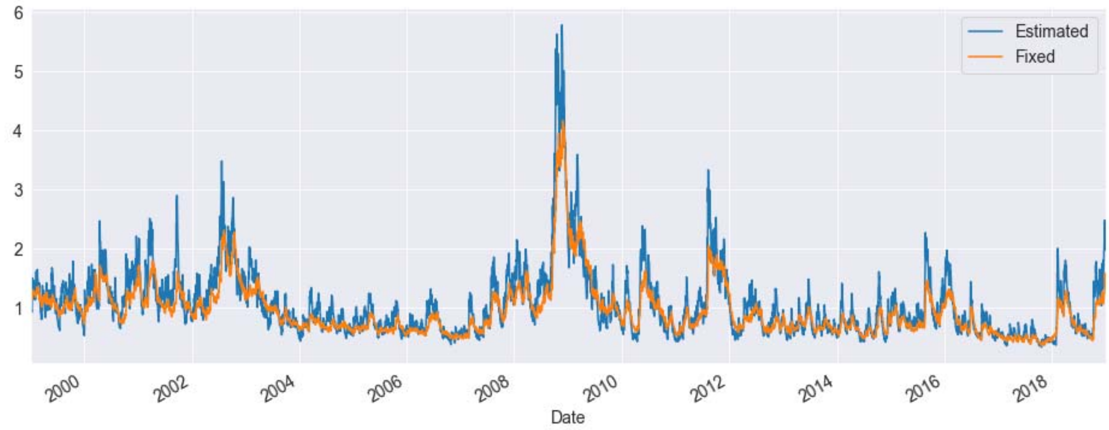
```
fixed_res = am.fix([0.0235, 0.01, 0.06, 0.0, 0.9382, 8.0])
print(fixed_res.summary())
```

```
                      Constant Mean - TARCH/ZARCH Model Results
==================================================================
Dep. Variable:                    Adj Close   R-squared:
Mean Model:                   Constant Mean   Adj. R-squared:
Vol Model:                      TARCH/ZARCH   Log-Likelihood:
Distribution:      Standardized Student's t   AIC:
Method:          User-specified Parameters   BIC:
                                              No. Observations:
Date:                      Tue, Mar 09 2021
Time:                              12:03:20
      Mean Model
=====================
                 coef
---------------------
mu             0.0235
   Volatility Model
=====================
                 coef
---------------------
omega          0.0100
alpha[1]       0.0600
gamma[1]       0.0000
beta[1]        0.9382
     Distribution
=====================
                 coef
---------------------
nu             8.0000
=====================

Results generated with user-specified parameters.
Std. errors not available when the model is not estimated,
```

In [9]:

```
import pandas as pd

df = pd.concat([res.conditional_volatility, fixed_res.conditional_
df.columns = ["Estimated", "Fixed"]
subplot = df.plot()
subplot.set_xlim(xlim)
```

Out[9]: (10596.0, 17896.0)



Building a Model From Components

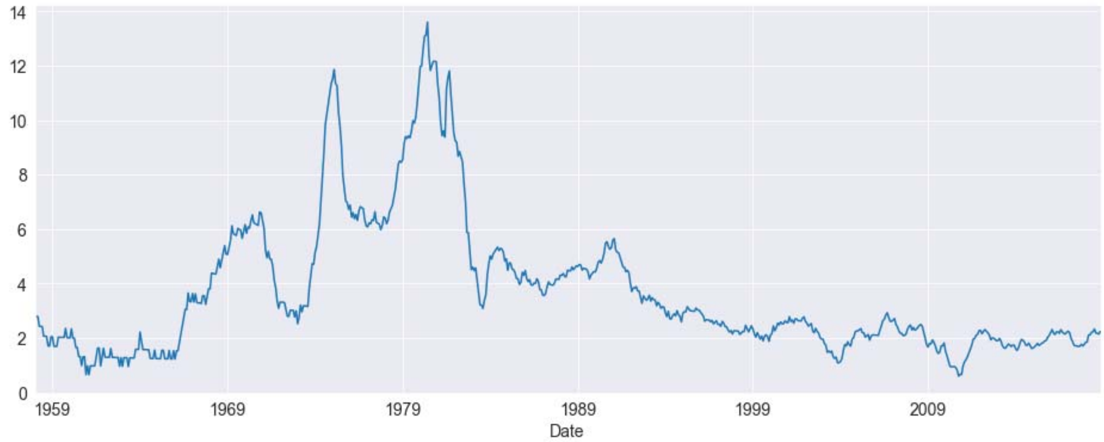
Models can also be systematically assembled from the three model components:

- A mean model (`arch.mean`)
  - Zero mean (`ZeroMean`) - useful if using residuals from a model estimated separately
  - Constant mean (`ConstantMean`) - common for most liquid financial assets
  - Autoregressive (`ARX`) with optional exogenous regressors
  - Heterogeneous (`HARX`) autoregression with optional exogenous regressors
  - Exogenous regressors only (`LS`)
- A volatility process (`arch.volatility`)
  - ARCH (`ARCH`)
  - GARCH (`GARCH`)
  - GJR-GARCH (`GARCH` using `o` argument)
  - TARCH/ZARCH (`GARCH` using `power` argument set to `1`)
  - Power GARCH and Asymmetric Power GARCH (`GARCH` using `power`)
  - Exponentially Weighted Moving Average Variance with estimated coefficient (`EWMAVariance`)
  - Heterogeneous ARCH (`HARCH`)
  - Parameterless Models
    - Exponentially Weighted Moving Average Variance, known as RiskMetrics (`EWMAVariance`)
    - Weighted averages of EWMAs, known as the RiskMetrics 2006 methodology (`RiskMetrics2006`)
- A distribution (`arch.distribution`)
  - Normal (`Normal`)
  - Standardized Students's T (`StudentsT`)

Mean Models

The first choice is the mean model. For many liquid financial assets, a constant mean (or even zero) is adequate. For other series, such as inflation, a more complicated model may be required. These examples make use of Core CPI downloaded from the Federal Reserve Economic Data (https://fred.stlouisfed.org/) site.

In [10]:
```
import arch.data.core_cpi

core_cpi = arch.data.core_cpi.load()
ann_inflation = 100 * core_cpi.CPILFESL.pct_change(12).dropna()
fig = ann_inflation.plot()
```



All mean models are initialized with constant variance and normal errors. For `ARX` models, the `lags` argument specifies the lags to include in the model.

In [11]:
```
from arch.univariate import ARX

ar = ARX(100 * ann_inflation, lags=[1, 3, 12])
print(ar.fit().summary())
```

```
                    AR - Constant Variance Model Results
==================================================================
Dep. Variable:                CPILFESL   R-squared:
Mean Model:                         AR   Adj. R-squared:
```

Volatility Processes

Volatility processes can be added a a mean model using the `volatility` property. This example adds an ARCH(5) process to model volatility. The arguments `iter` and `disp` are used in `fit()` to suppress estimation output.

In [12]:

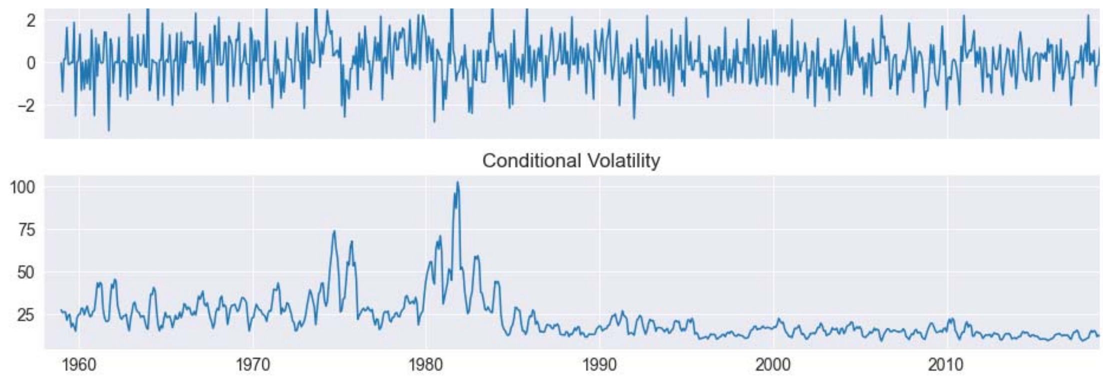
```
from arch.univariate import ARCH, GARCH

ar.volatility = ARCH(p=5)
res = ar.fit(update_freq=0, disp="off")
print(res.summary())
```

```
                          AR - ARCH Model Results
==================================================================
Dep. Variable:                CPILFESL   R-squared:
Mean Model:                         AR   Adj. R-squared:
Vol Model:                        ARCH   Log-Likelihood:
Distribution:                   Normal   AIC:
Method:             Maximum Likelihood   BIC:
                                         No. Observations:
Date:                 Tue, Mar 09 2021   Df Residuals:
Time:                         12:03:20   Df Model:
                                    Mean Model
==================================================================
                  coef    std err          t      P>|t|       95.
------------------------------------------------------------------
Const           2.8500      1.883      1.513      0.130      [ -0
CPILFESL[1]     1.0859  3.534e-02     30.726 2.590e-207      [  1
CPILFESL[3]    -0.0788  3.855e-02     -2.045  4.084e-02   [ -0.15
CPILFESL[12]   -0.0189  1.157e-02     -1.630      0.103 [-4.154e-
                              Volatility Model
==================================================================
                coef    std err          t      P>|t|    95.0% Co
------------------------------------------------------------------
omega        76.8602     16.015      4.799  1.592e-06 [ 45.472,1.
alpha[1]      0.1345  4.003e-02      3.359  7.824e-04 [5.600e-02,
alpha[2]      0.2280  6.284e-02      3.628  2.860e-04   [  0.105,
alpha[3]      0.1838  6.802e-02      2.702  6.891e-03 [5.047e-02,
alpha[4]      0.2538  7.826e-02      3.242  1.185e-03   [  0.100,
alpha[5]      0.1954  7.091e-02      2.756  5.853e-03 [5.644e-02,
==================================================================

Covariance estimator: robust
```

Plotting the standardized residuals and the conditional volatility shows some large (in magnitude) errors, even when standardized.

In [13]:

```
fig = res.plot()
```



Distributions

Finally the distribution can be changed from the default normal to a standardized Student's T using the `distribution` property of a mean model.

The Student's t distribution improves the model, and the degree of freedom is estimated to be near 8.

In [14]:
```
from arch.univariate import StudentsT

ar.distribution = StudentsT()
res = ar.fit(update_freq=0, disp="off")
print(res.summary())
```

```
                              AR - ARCH Model Results
==================================================================
Dep. Variable:                   CPILFESL   R-squared:
Mean Model:                            AR   Adj. R-squared:
Vol Model:                           ARCH   Log-Likelihood:
```

WTI Crude

The next example uses West Texas Intermediate Crude data from FRED. Three models are fit using alternative distributional assumptions. The results are printed, where we can see that the normal has a much lower log-likelihood than either the Standard Student's T or the Standardized Skew Student's T -- however, these two are fairly close. The closeness of the T and the Skew T indicate that returns are not heavily skewed.

In [15]:

```
from collections import OrderedDict

import arch.data.wti

crude = arch.data.wti.load()
crude_ret = 100 * crude.DCOILWTICO.dropna().pct_change().dropna()
res_normal = arch_model(crude_ret).fit(disp="off")
res_t = arch_model(crude_ret, dist="t").fit(disp="off")
res_skewt = arch_model(crude_ret, dist="skewt").fit(disp="off")
lls = pd.Series(
    OrderedDict(
        (
            ("normal", res_normal.loglikelihood),
            ("t", res_t.loglikelihood),
            ("skewt", res_skewt.loglikelihood),
        )
    )
)
print(lls)
params = pd.DataFrame(
    OrderedDict(
        (
            ("normal", res_normal.params),
            ("t", res_t.params),
            ("skewt", res_skewt.params),
        )
    )
)
params
```

```
normal   -18165.858870
t        -17919.643916
skewt    -17916.669052
dtype: float64
```

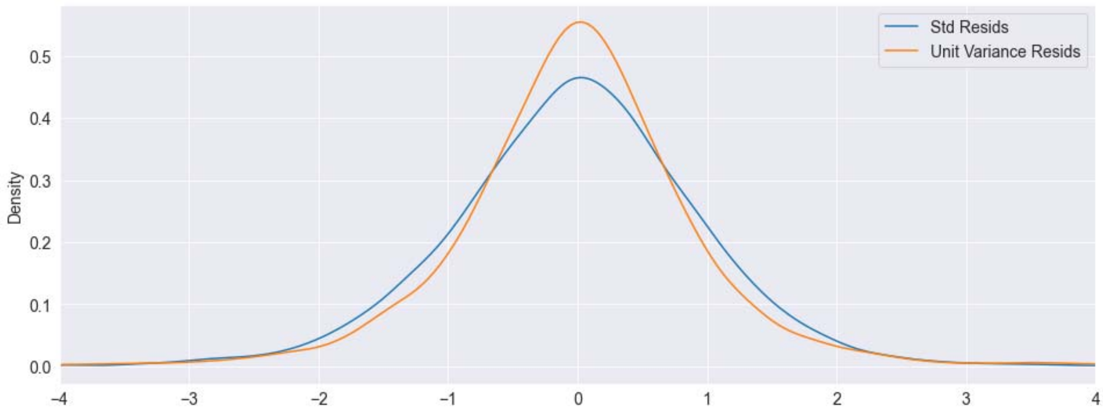
Out[15]:

	normal	t	skewt
alpha[1]	0.085627	0.064980	0.064889
beta[1]	0.909098	0.927950	0.928215
lambda	NaN	NaN	-0.036986

	normal	t	skewt
mu	0.046682	0.056438	0.040928

The standardized residuals can be computed by dividing the residuals by the conditional volatility. These are plotted along with the (unstandardized, but scaled) residuals. The non-standardized residuals are more peaked in the center indicating that the distribution is somewhat more heavy tailed than that of the standardized residuals.

In [16]:
```
std_resid = res_normal.resid / res_normal.conditional_volatility
unit_var_resid = res_normal.resid / res_normal.resid.std()
df = pd.concat([std_resid, unit_var_resid], 1)
df.columns = ["Std Resids", "Unit Variance Resids"]
subplot = df.plot(kind="kde", xlim=(-4, 4))
```



Simulation

All mean models expose a method to simulate returns from assuming the model is correctly specified. There are two required parameters, `params` which are the model parameters, and `nobs`, the number of observations to produce.

Below we simulate from a GJR-GARCH(1,1) with Skew-t errors using parameters estimated on the WTI series. The simulation returns a `DataFrame` with 3 columns:

- `data` : The simulated data, which includes any mean dynamics.
- `volatility` : The conditional volatility series
- `errors` : The simulated errors generated to produce the model. The errors are the difference between the data and its conditional mean, and can be transformed into the standardized errors by dividing by the volatility.

In [17]:
```
res = arch_model(crude_ret, p=1, o=1, q=1, dist="skewt").fit(disp=
pd.DataFrame(res.params)
```

Out[17]:

	params
mu	0.029365
omega	0.044375
alpha[1]	0.044344

	params
gamma[1]	0.036104
beta[1]	0.931280
nu	6.211290

In [18]:

```
sim_mod = arch_model(None, p=1, o=1, q=1, dist="skewt")

sim_data = sim_mod.simulate(res.params, 1000)
sim_data.head()
```

Out[18]:

	data	volatility	errors
0	-2.939211	1.464904	-2.968576
1	-2.041332	1.658853	-2.070698
2	-0.645152	1.718141	-0.674517
3	-1.812483	1.682297	-1.841848
4	1.530242	1.718407	1.500877

Simulations can be reproduced using a NumPy `RandomState` . This requires constructing a model from components where the `RandomState` instance is passed into to the distribution when the model is created.

The cell below contains code that builds a model with a constant mean, GJR-GARCH volatility and Skew t errors initialized with a user-provided `RandomState` . Saving the initial state allows it to be restored later so that the simulation can be run with the same random values.

In [19]:

```
import numpy as np
from arch.univariate import GARCH, ConstantMean, SkewStudent

rs = np.random.RandomState([892380934, 189201902, 129129894, 98904
# Save the initial state to reset later
state = rs.get_state()

dist = SkewStudent(random_state=rs)
vol = GARCH(p=1, o=1, q=1)
repro_mod = ConstantMean(None, volatility=vol, distribution=dist)

repro_mod.simulate(res.params, 1000).head()
```

Out[19]:

	data	volatility	errors
0	1.616836	4.787697	1.587470
1	4.106780	4.637128	4.077415
2	4.530200	4.561456	4.500834
3	2.284833	4.507738	2.255467
4	3.378518	4.381014	3.349153

Resetting the state using `set_state` shows that calling `simulate` using the same underlying state in the `RandomState` produces the same objects.

In [20]:

```
# Reset the state to the initial state
rs.set_state(state)
repro_mod.simulate(res.params, 1000).head()
```

Out[20]:

	data	volatility	errors
0	1.616836	4.787697	1.587470
1	4.106780	4.637128	4.077415
2	4.530200	4.561456	4.500834
3	2.284833	4.507738	2.255467
4	3.378518	4.381014	3.349153