Scipy’s new infrastructure for probability distributions

Albert Steppi, Quansight PBC

Matt Haberland, Cal Poly, San Luis Obispo

Pamphile Roy, SciPy Library

Why a new infrastructure?

APRIL 6, 2000 *by* JOEL SPOLSKY

Things You Should Never Do, Part I

TOP 10, CEO, NEWS

RFC: stats: univariate distribution infrastructure #15928

https://github.com/scipy/scipy/issues/15928

The Zen of Python

[1]:
```python
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
```

One obvious way to do it?

```
[20]: import scipy.stats as stats
```

```
[21]: stats.norm.cdf(1.3, 0, 2)
```

```
[21]: np.float64(0.7421538891941353)
```

```
[22]: dist = stats.norm(0, 2)
      dist.cdf(1.3)
```

```
[22]: np.float64(0.7421538891941353)
```

Stateless versus frozen distributions

```
[23]: type(stats.norm)

[23]: scipy.stats._continuous_distns.norm_gen

[24]: type(dist)

[24]: scipy.stats._distn_infrastructure.rv_continuous_frozen
```

Limitations of the old infrastructure

- All 125 distributions are instantiated on import, increasing import time
- The documentation generation scheme is inflexible and makes it difficult to write distribution specific documentation.
- Parameters are processed every time a method is called, increasing overhead.
- Distributions cannot be freed by the garbage collector due to self-references.

Families of distributions are classes
Individual distributions are instantiated objects

```
[25]: X = stats.Normal(mu=0, sigma=2)

[26]: X.cdf(1.3)

[26]: np.float64(0.7421538891941353)
```

I need to instantiate an object to compute a cdf?

```
[27]: %timeit X.cdf(1.3)

16.8 µs ± 823 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

[28]: %timeit dist.cdf(1.3)

93.9 µs ± 1.91 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

[29]: %timeit stats.norm.cdf(1.3, 0, 2)

88 µs ± 1.29 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Changes: new method names

- `sf` (survival function) → `ccdf` (complementary cumulative distribution function)
- `logsf` → `logccdf`
- `ppf` (percent point function) → `icdf` (inverse cumulative distribution function)
- `isf` (inverse survive function) → `iccdf` (inverse complementary cumulative distribution function)
- `std` → `standard_deviation`
- `var` → `variance`

Changes: new methods

- `ilogcdf` (inverse of the logarithm of the cumulative distribution function)
- `ilogccdf` (inverse of the logarithm of the complementary cumulative distribution function)
- `logentropy` (logarithm of the entropy)
- `mode` (mode of the distribution)
- `skewness`
- `kurtosis` (*non-excess* kurtosis; see "Standardized Moments" below)

And it has a new `plot` method for convenience

Sampling works differently: controlling random state

Old

```
import numpy as np
np.random.seed(1)
dist = stats.norm
dist.rvs(), dist.rvs(random_state=1)
```

```
(np.float64(1.6243453636632417), np.float64(1.6243453636632417))
```

New

```
X = stats.Normal()
rng = np.random.default_rng(1)  # instantiate a numpy.random.Generator
X.sample(rng=rng), X.sample(rng=1)
```

```
(np.float64(0.345584192064786), np.float64(0.345584192064786))
```

Sampling works differently, shapes

Parameter shapes are baked into the instance. Don't have to be passed to sampling function.

Old:

```
dist.rvs(size=(3, 4, 2), loc=[0, 1]).shape  # `loc` has shape (2,)
```

New:

```
Y = stats.Normal(mu = [0, 1])
Y.sample(shape=(3, 4)).shape  # the sample has shape (3, 4); each element is of
```

Fitting

- No distribution specific fit methods yet.
    - Because the one size fits all approach used was brittle and a source of bug reports.
- Goal: Creating idomatic for fitting using existing SciPy tools.

```
def nlps(x):
    c, scale = x
    X = Weibull(c=c) * scale
    x = np.sort(np.concatenate((data, X.support())))  # Append the endpoints of
    return -X.logcdf(x[:-1], x[1:]).sum().real  # Minimize the sum of the logs t

res_mps = optimize.minimize(nlps, x0, bounds=bounds)
res_mps.x
```

Cool stuff

Random variable point of view: shifting and scaling

```
[49]: X = stats.Normal(mu=0, sigma=1)
      Y = stats.Normal(mu=1, sigma=2)

[50]: Z = 2*X + 1
      Z

[50]: np.float64(2.0)*Normal(mu=np.float64(0.0), sigma=np.float64(1.0)) + np.float64(1.0)

[51]: Y.pdf(2), Z.pdf(2)

[51]: (np.float64(0.17603266338214976), np.float64(0.17603266338214976))
```

Random variable point of View: transformations

```
[36]: X = stats.Normal(mu=0, sigma=1)
```

```
[37]: Y = stats.exp(X)
      Y
```

```
[37]: exp(Normal(mu=np.float64(0.0), sigma=np.float64(1.0)))
```

```
[43]: Y.pdf(1), stats.lognorm.pdf(1, 1)
```

```
[43]: (np.float64(0.3989422804014327), np.float64(0.3989422804014327))
```

Transformations

```
[52]: X = stats.Normal(mu=0, sigma=1)
      X**2

[52]: (Normal(mu=np.float64(0.0), sigma=np.float64(1.0)))**2

[53]: stats.log(stats.abs(X))

[53]: log(abs(Normal(mu=np.float64(0.0), sigma=np.float64(1.0))))
```

Transformations: Limitations

[54]:
```
stats.log(X)
```

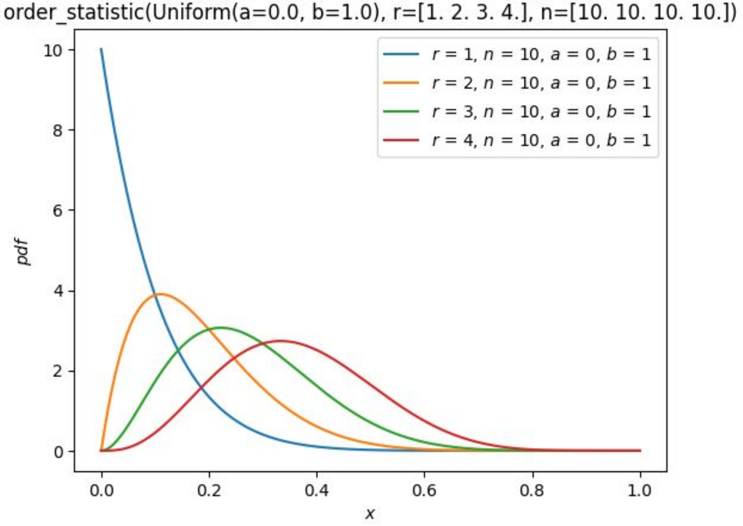
```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
Cell In[54], line 1
----> 1 stats.log(X)

File ~/.venvs/scipy2025/lib/python3.12/site-packages/scipy/stats/_distribution_infrastructure.py:5748, in log(X)
   5745 if np.any(X.support()[0] < 0):
   5746     message = ("The logarithm of a random variable is only implemented when the "
   5747                "support is non-negative.")
-> 5748     raise NotImplementedError(message)
   5749 return MonotonicTransformedDistribution(X, g=np.log, h=np.exp, dh=np.exp,
   5750                                         logdh=lambda u: u)

NotImplementedError: The logarithm of a random variable is only implemented when the support is non-negative.
```

Order Statistics

```
[3]: X = stats.Uniform(a=0, b=1)
     Y = stats.order_statistic(X, n=10, r=[1, 2, 3, 4])
     Y
```

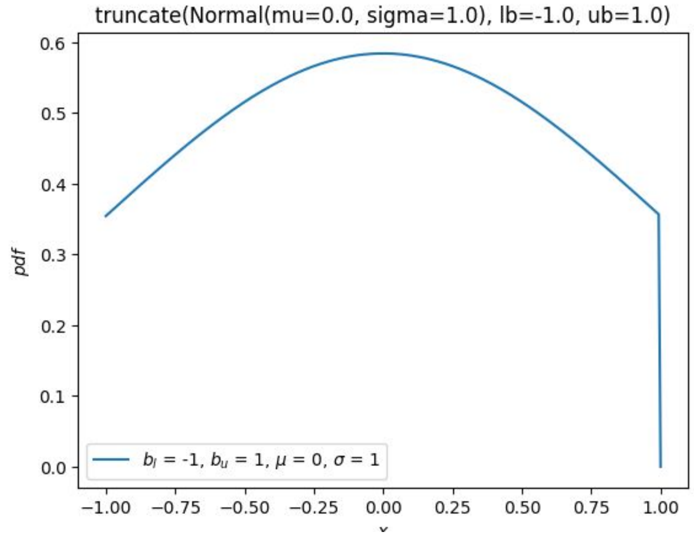
```
[3]: order_statistic(Uniform(a=np.float64(0.0), b=np.float64(1.0)), r=array([1., 2., 3., 4.]), n=array([10., 10., 10., 10.]))
```

```
[65]: Y.plot()
```



Truncation

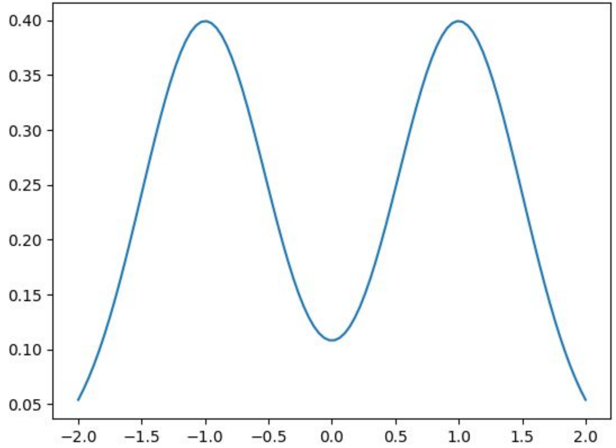
```
[21]: X = stats.Normal(mu=0, sigma=1)
      Y = stats.truncate(X, lb=-1, ub=1)
      Y.plot()
```



Mixture Distributions

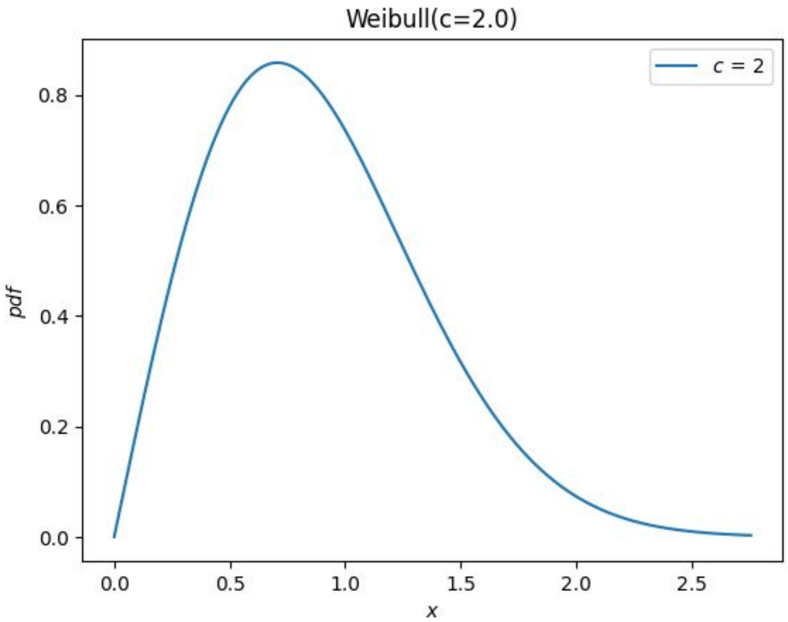
$$Z \sim 0.5 \cdot \mathrm{Normal}\,(1, 0.5) + 0.5 \cdot \mathrm{Normal}\,(-1, 0.5)$$

[85]:
```
Z = stats.Mixture(
    [stats.Normal(mu=-1, sigma=0.5), stats.Normal(mu=1, sigma=0.5)], weights=[0.5, 0.5]
)
Z.plot()  # warning, as of SciPy 1.16 this doesn't actually work yet
```



But the old infra has so many more distributions

```
[88]: Weibull = stats.make_distribution(stats.weibull_min)
      W = Weibull(c=2.0)
      W.plot()
```



Custom Distributions

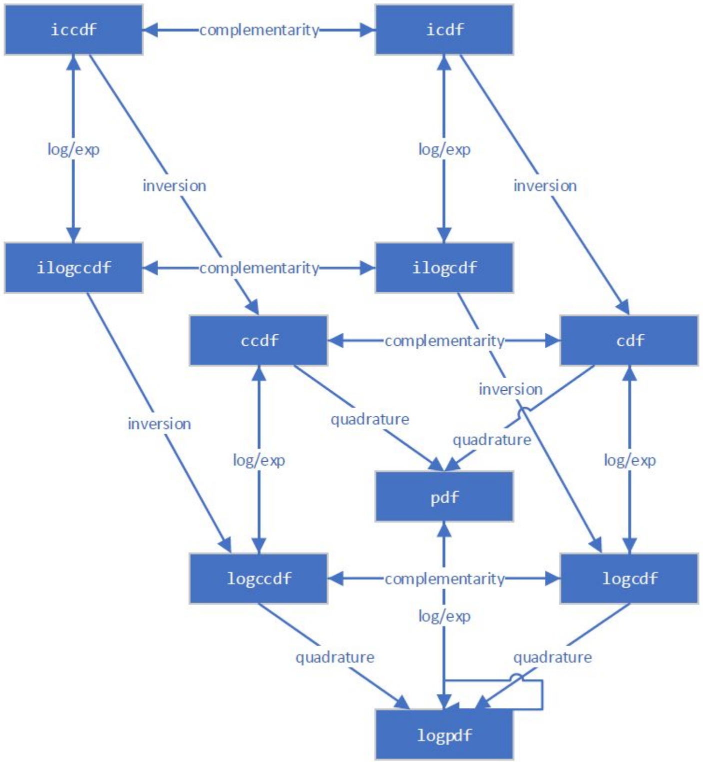
```
[102…
class LogUniform:
    @property
    def __make_distribution_version__(self):
        return "1.16.0"

    @property
    def parameters(self):
        return {'a': {'endpoints': (0, np.inf),
                      'inclusive': (False, False)},
                'b': {'endpoints': ('a', np.inf),
                      'inclusive': (False, False)}}


    @property
    def support(self):
        return {'endpoints': ('a', 'b'), 'inclusive': (True, True)}

    def pdf(self, x, a, b):
        return 1 / (x * (np.log(b)- np.log(a)))
```

```
[103]: LogUniform = stats.make_distribution(LogUniform())
```

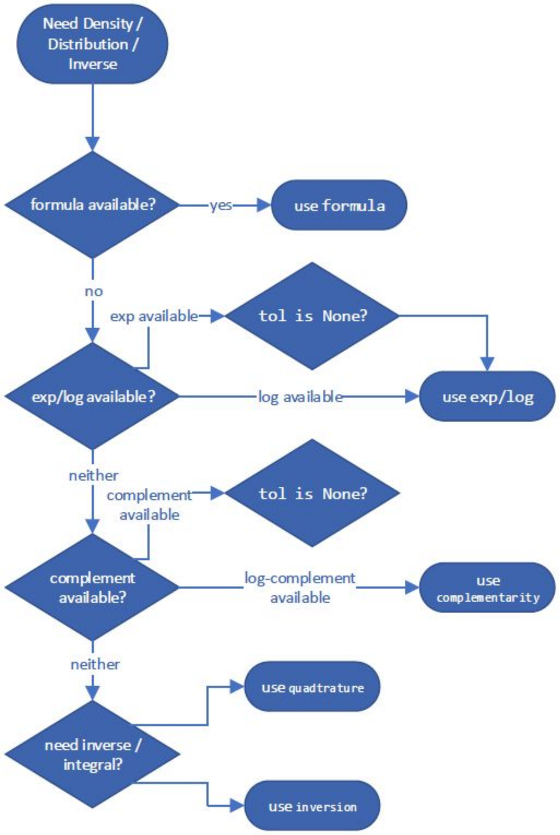
Methods are vectorized by default

```
[103]: LogUniform = stats.make_distribution(LogUniform())

[105]: A = LogUniform(a=1, b=10)

[109]: %timeit A.cdf(np.linspace(1, 10, 10000))

75 ms ± 2.63 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

[110]: %timeit stats.loguniform.cdf(np.linspace(1, 10, 10000), 1, 10)

451 µs ± 9.19 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Relations between methods



Method selection example for CDF



Upcoming work

- Infrastructure for circular distributions
- More sophisticated arithmetic on random variables, e.g. (X + Y)
- Support for alternative array API backends
- Consistent “one-way” recipe for distribution fitting

Tutorial available

Random Variable Transition Guide — SciPy v1.15.3 Manual

Acknowledgements

Others who contributed to the design discussion

- Christoph Baumgarten (@chrisb83)
- Evgeni Burovski (@ev-br)
- Neil Girdhar (@NeilGirdhar)
- Robert Kern (@rkern)
- Tirth Patel (@tirthashepatel)
- Josef Perktold (@josef-pkt)
- Pamphile Roy (@tupui)
- Daniel Schmitz (@dschmitz89)

and more, sorry if I’ve missed anyone.

Work funded by Chan Zuckerberg Initiative (CZI) grant EOSS Cycle 5 grant

SciPy: Fundamental tools for Biomedicine

Thank you