

frplib Cheatsheet

C. Genovese

23 Aug 2026 v0.2.14 (frplib v0.2.14)

Interaction

The playground is an enhanced Python read-eval-print loop (REPL). You can use any Python construct, and the environment is preloaded with playground specific functions and values.

You can use arrows (or Control-n and Control-p) to move through your history, optionally modifying earlier commands. Special keys: Arrow keys, Control-a (beginning of line), Control-e (end of line), Control-k delete rest of line, F3 see full history, F2 settings menu, Control-r search history for pattern (type pattern and hit enter to select).

Multi-line input is accepted; a blank line at the end will enter the input.

Syntax errors will show up in a message at the bottom of the screen, usually disallowing entering that input. Many errors raised by code will show special playground-specific error messages, but some errors will show the full Python stack traces.

Kind Factories

All factories with arguments `values...` can take multiple values as individual arguments, or an iterable, or an implied sequence of the form `a, b, ..., c` in a positive or negative direction. (If `a == c` in this form, `b` is ignored, giving a single value `a`.) Values can be numbers or tuples and can contain symbols.

`kind(spec)` – constructs a Kind from a string, an FRP, or another Kind.

`conditional_kind(mapping)` – constructs a conditional Kind from a dictionary or function. (For the latter, use argument `codim=1` if the function wants a scalar argument.) Can be used as a decorator or a function.

`Kind.empty` – the empty Kind

`constant(v)` – Kind of a constant FRP with value `v`

`binary(p)` – the Kind of a 0-1 FRP with weight `p` on 1

`uniform(values...)` – the Kind with specified values and equal weights

`choice(a, b, ratio)` – has values `a` and `b` with weights 1 and `ratio`

`weighted_as(values..., weights=[...])` – arbitrary weights associated with the given values, can also accept a dictionary `{value: weight, ...}`

`weighted_by` – weights on values determined by a general function

`weighted_pairs` – a Kind specified by a sequence of `(value, weight)` pairs

`symmetric(values..., around, weight_by)`

– weights on values determined by a symmetric function `weight_by` around a specified value

`geometric(values..., r)` – weights on values varying geometrically with ratio `r`

`linear(values..., first=a, increment=b)` – weights on values vary linearly from `a` changing by `b` for each value.

`evenly_spaced` – Kind of an FRP whose values consist of evenly spaced numbers

`integers` – Kind with integer sequence as values

`subsets, without_replacement, permutations_of, ordered_samples` – the Kinds of combinatorial operations on sequences: all subsets, samples of a given size without replacement, permutations of a given size, and ordered samples without replacement

`arbitrary` – the Kind with specified values and symbolic (unspecified) weights

FRP Factories

`frp(spec)` – constructs an FRP from a Kind or clones another FRP.

`conditional_frp(mapping)` – constructs a conditional Kind from a dict or function

`shuffle(coll)` – constructs an FRP whose value is a random permutation of the collection `coll`

Kind and FRP Combinators

`kf ^ stat` or `stat(kf)` – apply a statistic `stat` to a Kind (or FRP) `kf`

`kf1 * kf2` – independent joins of `kf1` and `kf2`, either both Kinds or both FRPs

`kf ** n` – independent join power, for Kind or FRP `kf` and natural number `n`

`kf >> ckf` – a general join for a (conditional) Kind (or FRP) `kf` and a conditional Kind (or conditional FRP) `ckf`. Returns a (conditional) Kind (or FRP).

`kf | c` – constrains a Kind/FRP with a condition, `kf` is a Kind or FRP and `c` is a condition.

`m // k` – conditioning on the Kind (or FRP) `k`, where `m` is a conditional Kind (or conditional FRP).

`psi@k | c` – evaluate a statistic with context

`fast_join_pow(stat, kind, n)` – efficiently computes `mstat(kind ** n)`

`bin(scalar_kind, lower, width)` – returns a Kind similar to that given but with values binned in specified intervals

`evolve(start, next_state, steps=1)` – evolves a system through a specified number of steps

`bayes(observed_y, x, y_given_x)` – applies Bayes's rule for Kinds or FRPs

Statistics Factories

`statistic, scalar_statistic` – Creates a statistic from a function. The function is either passed as the first argument or is being defined with `@statistic` used as a decorator. The latter forces dimension 1.

`condition` – Creates a condition from a function. The function is either passed as the first argument or is being defined with `@statistic` used as a decorator.

`Constantly(v)` – a statistic that always returns `v`

`Proj` – constructs projection statistics given an index, list of indices, slice, projection statistic, or integer iterable. Projections are **1-indexed**, unlike tuples.

`Permute` – constructs a permutation statistic given a permutation of 1..n in cycle or ordinary form

`Append, Prepend` – appends or prepends a value to the input tuple

`Get, ElementOf` – former uses an object as a lookup key, latter creates a condition that an element is in the input

Builtin Statistics

`__` – The statistic that reproduces the value passed to it. `Scalar` is similar but forces the result to be a scalar. `Id` is a synonym.

`Sum`, `Product`, `Min`, `Max`, `Mean` – operations on the value tuple; for example, `Sum` gives the component sum
`Exp`, `Log`, `Log2`, `Log10`, `Sqrt`, `Abs`, `Floor`, `Ceil`
`Sin`, `Cos`, `Tan`, `ACos`, `ASin`, `ATan2`, `Sinh`, `Cosh`, `Tanh`, . . . – arithmetic and special functions
`Diff`, `Diffs` – first-order and higher-order differences
`NormalCDF` – standard Normal CDF

`Cases` – creates a statistic from a dictionary with optional default

`top` and `bottom` – statistics that always return true and false, respectively.

Statistics Combinators

`Fork(stat1, stat2, ..., statn)` – creates a new statistic that combines the results of `stat1` ... `statn` (with the same argument) into a tuple. (`MFork` is identical but is intended for monoidal statistics.)

`ForEach(stat)` – apply statistic to each component of the input tuple, combining results into a tuple
`IfThenElse(cond, statt, statf)` – applies condition `cond` evaluates to true, apply `statt`, else `statf`.

`And`, `Or`, `Not`, `Xor` – logical operations on the results of statistics, returning a condition. For example, `And(stat1, stat2)` gives a condition that returns true if both `stat1` and `stat2` do.

`All`, `Any` – condition true on every/some components

Actions

`E` – expectation operator, computes expectation of a Kind, FRP, conditional Kind, or conditional frp. (The latter two return functions.) See also `Var`.

`D_` – distribution operator `D_ (X) (psi)` returns `E(psi(X))`

`unfold(k)` – shows the unfolded Kind tree for a given Kind `k`

`clean(k)` – given Kind `k` removes any branches that are numerically zero according to a specified tolerance (default 1e-16). It also rounds numeric values to avoid round-off error in comparing values

`FRP.sample(n, obj, summary=True)` – generate `n` samples from the given Kind or FRP `obj`.

Default produces a summary table, but if `summary=False`, give all the values.

`Kind.equal`, `Kind.compare` – compare Kinds for structural equality, testing weights and values within a specified numerical tolerance (default: 1e-12)

`Kind.divergence(k1, k2)` – computes relative entropy of Kind `k1` relative to this `k2`.

`Market` – Market commands ('demo', 'buy', 'compare', 'show') in the playground

Utilities

`show(x)` – displays an object, list, or dictionary in a more friendly manner.

`clone(X)` – produces a copy of its argument 'X' if possible; primarily useful with FRPs and conditional FRPs, where it produces fresh copies with their own values.

Property Accessors

`X.value` – for an FRP `X`, returns `X`'s value, activating it if necessary.

`dim(x)` – returns the dimension of 'x', if available. Note that taking the dimension of an FRP may force the Kind computation.

`codim(x)` – returns the codimension of 'x', if available

`size(x)` – returns the size of 'x', if available

`typeof(x)` – returns the type of 'x'

`values(x)` – returns the *set* of 'x''s values, if available; applies to Kinds

Symbolic Manipulation

`is_symbolic(x)` – returns true if 'x' is symbolic

`symbol(name)` – takes a string and creates a symbolic term with that name

`symbols(names)` – takes a string with space-separated names and returns symbols

`gen_symbol()` – returns a unique symbol name every time it is called

`substitute(quantity, mapping)` – substitutes values from mapping for the symbols in 'quantity'; mapping is a dictionary associating symbol names with values. Not all symbols need to be substituted; if all are substituted with a numeric value then the result is numeric.

`substitute_with(mapping)` – returns a function that takes a quantity and substitutes with mapping in that quantity.

`substitution(quantity, **kw)` – like 'substitute' but takes names and values as keyword arguments rather than through a dictionary.

Tuples and Quantities

`tup(x...)` – converts arguments to a quantitative vector tuple, whose values are numeric or symbolic quantities and can be added or scaled like vectors.

`as_scalar(value)` :: converts a 1-tuple to a scalar

`as_quantity(spec)` – converts to a quantity, takes symbols, strings, or numbers, e.g.,

`as_quantity('1/2')`, `as_quantity(1.2)`,

`as_quantity('a')`.

`numeric_exp(x)`, `numeric_ln(x)`,
`numeric_log2(x)`, `numeric_log10(x)`,
`numeric_abs(x)`, `numeric_sqrt(x)`,
`numeric_floor(x)`, `numeric_ceil(x)` – numeric special functions that act on quantities

Function Helpers

`identity(x)` – a function that returns its argument

`const(a)` – returns a function that itself always returns the value 'a'

`compose(f, g)` – returns the function 'f' after 'g'

Sequence Helpers

`irange` – creates an inclusive integer ranges with optional gaps

`index_of`, `index_where` – searches sequence with control over what to return if not found

`every(f, iterable)` – returns true if 'f(x)' is truthy for every 'x' in 'iterable'

`some(f, iterable)` – returns true if 'f(x)' is truthy for some 'x' in 'iterable'

`lmap(f, iterable)` – returns a **list** containing 'f(x)' for every 'x' in 'iterable'

`frequencies(iterable, counts_only=False)` – computes counts of unique values in iterable; returns a dictionary, but if `counts_only` is True, return just the counts without labels.

Help

`info()` – interactive help system. Also accepts a topic string or most playground objects (e.g., uniform).

`help(obj)` – built-in python help, you can call this on any playground function or object to get guidance on its use.