Random Numbers

Random number generation in Julia uses the Mersenne Twister library via `MersenneTwister` objects. Julia has a global RNG, which is used by default. Other RNG types can be plugged in by inheriting the `AbstractRNG` type; they can then be used to have multiple streams of random numbers. Besides `MersenneTwister`, Julia also provides the `RandomDevice` RNG type, which is a wrapper over the OS provided entropy.

Most functions related to random generation accept an optional `AbstractRNG` object as first argument, which defaults to the global one if not provided. Moreover, some of them accept optionally dimension specifications `dims...` (which can be given as a tuple) to generate arrays of random values.

A `MersenneTwister` or `RandomDevice` RNG can generate uniformly random numbers of the following types: `Float16`, `Float32`, `Float64`, `BigFloat`, `Bool`, `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Int128`, `UInt128`, `BigInt` (or complex numbers of those types). Random floating point numbers are generated uniformly in $[0, 1)$. As `BigInt` represents unbounded integers, the interval must be specified (e.g. `rand(big.(1:6))`).

Additionally, normal and exponential distributions are implemented for some `AbstractFloat` and `Complex` types, see `randn` and `randexp` for details.

Warn

Because the precise way in which random numbers are generated is considered an implementation detail, bug fixes and speed improvements may change the stream of numbers that are generated after a version change. Relying on a specific seed or generated stream of numbers during unit testing is thus discouraged - consider testing properties of the methods in question instead.

Random numbers module

`Random.Random` — Module

```
Random
```

Support for generating random numbers. Provides `rand`, `randn`, `AbstractRNG`, `MersenneTwister`, and `RandomDevice`.

Random generation functions

`Base.rand` — Function

```
rand([rng=GLOBAL_RNG], [S], [dims...])
```

Pick a random element or array of random elements from the set of values specified by `S`; `S` can be

- an indexable collection (for example `1:9` or `('x', "y", :z)`),
- an `AbstractDict` or `AbstractSet` object,
- a string (considered as a collection of characters), or
- a type: the set of values to pick from is then equivalent to `typemin(S):typemax(S)` for integers (this is not applicable to `BigInt`), to $[0, 1)$ for floating point numbers and to $[0, 1) + i[0, 1)$ for complex floating point numbers;

`S` defaults to `Float64`. When only one argument is passed besides the optional `rng` and is a `Tuple`, it is interpreted as a collection of values (`S`) and not as `dims`.

Julia 1.1

Support for `S` as a tuple requires at least Julia 1.1.

Examples

```
julia> rand(Int, 2)
2-element Array{Int64,1}:
 1339893410598768192
 1575814717733606317

julia> using Random

julia> rand(MersenneTwister(0), Dict(1=>2, 3=>4))
1=>2

julia> rand((2, 3))
```

```
3

julia> rand(Float64, (2, 3))
2×3 Array{Float64,2}:
 0.999717  0.0143835  0.540787
 0.696556  0.783855   0.938235
```

Note

The complexity of `rand(rng, s::Union{AbstractDict,AbstractSet})` is linear in the length of `s`, unless an optimized method with constant complexity is available, which is the case for `Dict`, `Set` and `BitSet`. For more than a few calls, use `rand(rng, collect(s))` instead, or either `rand(rng, Dict(s))` or `rand(rng, Set(s))` as appropriate.

`Random.rand!` — Function

```
rand!([rng=GLOBAL_RNG], A, [S=eltype(A)])
```

Populate the array `A` with random values. If `S` is specified (`S` can be a type or a collection, cf. `rand` for details), the values are picked randomly from `S`. This is equivalent to `copyto!(A, rand(rng, S, size(A)))` but without allocating a new array.

Examples

```
julia> rng = MersenneTwister(1234);

julia> rand!(rng, zeros(5))
5-element Array{Float64,1}:
 0.5908446386657102
 0.7667970365022592
 0.5662374165061859
 0.4600853424625171
 0.7940257103317943
```

`Random.bitrand` — Function

```
bitrand([rng=GLOBAL_RNG], [dims...])
```

Generate a `BitArray` of random boolean values.

Examples

```
julia> rng = MersenneTwister(1234);

julia> bitrand(rng, 10)
10-element BitArray{1}:
 0
 1
 1
 1
 1
 0
 1
 0
 0
 1
```

`Base.randn` — Function

```
randn([rng=GLOBAL_RNG], [T=Float64], [dims...])
```

Generate a normally-distributed random number of type `T` with mean 0 and standard deviation 1. Optionally generate an array of normally-distributed random numbers. The `Base` module currently provides an implementation for the types `Float16`, `Float32`, and `Float64` (the default), and their `Complex` counterparts. When the type argument is complex, the values are drawn from the circularly symmetric complex normal distribution of variance 1 (corresponding to real and imaginary part having independent normal distribution with mean zero and variance `1/2`).

Examples

```
julia> using Random

julia> rng = MersenneTwister(1234);

julia> randn(rng, ComplexF64)
```

```
0.6133070881429037 - 0.6376291670853887im

julia> randn(rng, ComplexF32, (2, 3))
2×3 Array{Complex{Float32},2}:
 -0.349649-0.638457im  0.376756-0.192146im  -0.396334-0.0136413im
  0.611224+1.56403im   0.355204-0.365563im  0.0905552+1.31012im
```

`Random.randn!` — Function

```
randn!([rng=GLOBAL_RNG], A::AbstractArray) -> A
```

Fill the array `A` with normally-distributed (mean 0, standard deviation 1) random numbers. Also see the `rand` function.

Examples

```
julia> rng = MersenneTwister(1234);

julia> randn!(rng, zeros(5))
5-element Array{Float64,1}:
  0.8673472019512456
 -0.9017438158568171
 -0.4944787535042339
 -0.9029142938652416
  0.8644013132535154
```

`Random.randexp` — Function

```
randexp([rng=GLOBAL_RNG], [T=Float64], [dims...])
```

Generate a random number of type `T` according to the exponential distribution with scale 1. Optionally generate an array of such random numbers. The `Base` module currently provides an implementation for the types `Float16`, `Float32`, and `Float64` (the default).

Examples

```
julia> rng = MersenneTwister(1234);
julia> randexp(rng, Float32)
2.4835055f0

julia> randexp(rng, 3, 3)
3×3 Array{Float64,2}:
 1.5167    1.30652   0.344435
 0.604436  2.78029   0.418516
 0.695867  0.693292  0.643644
```

`Random.randexp!` — Function

```
randexp!([rng=GLOBAL_RNG], A::AbstractArray) -> A
```

Fill the array `A` with random numbers following the exponential distribution (with scale 1).

Examples

```
julia> rng = MersenneTwister(1234);

julia> randexp!(rng, zeros(5))
5-element Array{Float64,1}:
 2.4835053723904896
 1.516703605376473
 0.6044364871025417
 0.6958665886385867
 1.3065196315496677
```

`Random.randstring` — Function

```
randstring([rng=GLOBAL_RNG], [chars], [len=8])
```

Create a random string of length `len`, consisting of characters from `chars`, which defaults to the set of upper- and lower-case letters and the digits 0-9. The optional `rng` argument specifies a random number generator, see Random Numbers.

Examples

```
julia> Random.seed!(3); randstring()
```

```
"4zSHdXlw"

julia> randstring(MersenneTwister(3), 'a':'z', 6)
"bzlhqn"

julia> randstring("ACGT")
"AGGACATT"
```

Note

`chars` can be any collection of characters, of type `Char` or `UInt8` (more efficient), provided `rand` can randomly pick characters from it.

Subsequences, permutations and shuffling

`Random.randsubseq` — Function

```
randsubseq([rng=GLOBAL_RNG,] A, p) -> Vector
```

Return a vector consisting of a random subsequence of the given array `A`, where each element of `A` is included (in order) with independent probability `p`. (Complexity is linear in `p*length(A)`, so this function is efficient even if `p` is small and `A` is large.) Technically, this process is known as "Bernoulli sampling" of `A`.

Examples

```
julia> rng = MersenneTwister(1234);

julia> randsubseq(rng, 1:8, 0.3)
2-element Array{Int64,1}:
 7
 8
```

`Random.randsubseq!` — Function

```
randsubseq!([rng=GLOBAL_RNG,] S, A, p)
```

Like `randsubseq`, but the results are stored in `S` (which is resized as needed).

Examples

```
julia> rng = MersenneTwister(1234);

julia> S = Int64[];

julia> randsubseq!(rng, S, 1:8, 0.3)
2-element Array{Int64,1}:
 7
 8

julia> S
2-element Array{Int64,1}:
 7
 8
```

`Random.randperm` — Function

```
randperm([rng=GLOBAL_RNG,] n::Integer)
```

Construct a random permutation of length `n`. The optional `rng` argument specifies a random number generator (see Random Numbers). The element type of the result is the same as the type of `n`.

To randomly permute an arbitrary vector, see `shuffle` or `shuffle!`.

Julia 1.1

In Julia 1.1 `randperm` returns a vector `v` with `eltype(v) == typeof(n)` while in Julia 1.0 `eltype(v) == Int`.

Examples

```
julia> randperm(MersenneTwister(1234), 4)
```

```
4-element Array{Int64,1}:
 2
 1
 4
 3
```

Random.randperm! — Function

```
randperm!([rng=GLOBAL_RNG,] A::Array{<:Integer})
```

Construct in `A` a random permutation of length `length(A)`. The optional `rng` argument specifies a random number generator (see Random Numbers). To randomly permute an arbitrary vector, see `shuffle` or `shuffle!`.

Examples

```
julia> randperm!(MersenneTwister(1234), Vector{Int}(undef, 4))
4-element Array{Int64,1}:
 2
 1
 4
 3
```

Random.randcycle — Function

```
randcycle([rng=GLOBAL_RNG,] n::Integer)
```

Construct a random cyclic permutation of length `n`. The optional `rng` argument specifies a random number generator, see Random Numbers. The element type of the result is the same as the type of `n`.

Julia 1.1

In Julia 1.1 `randcycle` returns a vector `v` with `eltype(v) == typeof(n)` while in Julia 1.0 `eltype(v) == Int`.

Examples

```
julia> randcycle(MersenneTwister(1234), 6)
6-element Array{Int64,1}:
 3
 5
 4
 6
 1
 2
```

Random.randcycle! — Function

```
randcycle!([rng=GLOBAL_RNG,] A::Array{<:Integer})
```

Construct in `A` a random cyclic permutation of length `length(A)`. The optional `rng` argument specifies a random number generator, see Random Numbers.

Examples

```
julia> randcycle!(MersenneTwister(1234), Vector{Int}(undef, 6))
6-element Array{Int64,1}:
 3
 5
 4
 6
 1
 2
```

Random.shuffle — Function

```
shuffle([rng=GLOBAL_RNG,] v::AbstractArray)
```

Return a randomly permuted copy of `v`. The optional `rng` argument specifies a random number generator (see Random Numbers). To permute `v` in-place, see `shuffle!`. To obtain randomly permuted indices, see `randperm`.

Examples

```
julia> rng = MersenneTwister(1234);

julia> shuffle(rng, Vector(1:10))
10-element Array{Int64,1}:
  6
  1
 10
  2
  3
  9
  5
  7
  4
  8
```

`Random.shuffle!` — Function

```
shuffle!([rng=GLOBAL_RNG,] v::AbstractArray)
```

In-place version of `shuffle`: randomly permute `v` in-place, optionally supplying the random-number generator `rng`.

Examples

```
julia> rng = MersenneTwister(1234);

julia> shuffle!(rng, Vector(1:16))
16-element Array{Int64,1}:
  2
 15
  5
 14
  1
  9
 10
  6
 11
  3
 16
  7
  4
 12
  8
 13
```

Generators (creation and seeding)

`Random.seed!` — Function

```
seed!([rng=GLOBAL_RNG], seed) -> rng
seed!([rng=GLOBAL_RNG]) -> rng
```

Reseed the random number generator: `rng` will give a reproducible sequence of numbers if and only if a `seed` is provided. Some RNGs don't accept a seed, like `RandomDevice`. After the call to `seed!`, `rng` is equivalent to a newly created object initialized with the same seed.

If `rng` is not specified, it defaults to seeding the state of the shared thread-local generator.

Examples

```
julia> Random.seed!(1234);

julia> x1 = rand(2)
2-element Array{Float64,1}:
 0.590845
 0.766797

julia> Random.seed!(1234);

julia> x2 = rand(2)
2-element Array{Float64,1}:
 0.590845
 0.766797

julia> x1 == x2
true

julia> rng = MersenneTwister(1234); rand(rng, 2) == x1
true

julia> MersenneTwister(1) == Random.seed!(rng, 1)
true
```

```
julia> rand(Random.seed!(rng), Bool) # not reproducible
true

julia> rand(Random.seed!(rng), Bool)
false

julia> rand(MersenneTwister(), Bool) # not reproducible either
true
```

`Random.AbstractRNG` — Type

```
AbstractRNG
```

Supertype for random number generators such as `MersenneTwister` and `RandomDevice`.

`Random.MersenneTwister` — Type

```
MersenneTwister(seed)
MersenneTwister()
```

Create a `MersenneTwister` RNG object. Different RNG objects can have their own seeds, which may be useful for generating different streams of random numbers. The `seed` may be a non-negative integer or a vector of `UInt32` integers. If no seed is provided, a randomly generated one is created (using entropy from the system). See the `seed!` function for reseeding an already existing `MersenneTwister` object.

Examples

```
julia> rng = MersenneTwister(1234);

julia> x1 = rand(rng, 2)
2-element Array{Float64,1}:
 0.5908446386657102
 0.7667970365022592

julia> rng = MersenneTwister(1234);

julia> x2 = rand(rng, 2)
```

```
2-element Array{Float64,1}:
 0.5908446386657102
 0.7667970365022592

julia> x1 == x2
true
```

`Random.RandomDevice` — Type

```
RandomDevice()
```

Create a `RandomDevice` RNG object. Two such objects will always generate different streams of random numbers. The entropy is obtained from the operating system.

Hooking into the `Random` API

There are two mostly orthogonal ways to extend `Random` functionalities:

1. generating random values of custom types
2. creating new generators

The API for 1) is quite functional, but is relatively recent so it may still have to evolve in subsequent releases of the `Random` module. For example, it's typically sufficient to implement one `rand` method in order to have all other usual methods work automatically.

The API for 2) is still rudimentary, and may require more work than strictly necessary from the implementor, in order to support usual types of generated values.

Generating random values of custom types

Generating random values for some distributions may involve various trade-offs. *Pre-computed* values, such as an alias table for discrete distributions, or “squeezing” functions for univariate distributions, can speed up sampling considerably. How much information should be pre-computed can depend on the number of values we plan to draw from a distribution. Also, some random number generators can have certain properties that various algorithms may want to exploit.

The `Random` module defines a customizable framework for obtaining random values that can address these issues. Each invocation of `rand` generates a *sampler* which can be customized with the above trade-offs in mind, by adding methods to `Sampler`, which in turn can dispatch on the random number generator, the object that characterizes the distribution, and a suggestion for the number of repetitions. Currently, for the latter, `Val{1}` (for a single sample) and `Val{Inf}` (for an arbitrary number) are used, with `Random.Repetition` an alias for both.

The object returned by `Sampler` is then used to generate the random values. When implementing the random generation interface for a value `X` that can be sampled from, the implementor should define the method

```
rand(rng, sampler)
```

for the particular `sampler` returned by `Sampler(rng, X, repetition)`.

Samplers can be arbitrary values that implement `rand(rng, sampler)`, but for most applications the following predefined samplers may be sufficient:

1. `SamplerType{T}()` can be used for implementing samplers that draw from type `T` (e.g. `rand(Int)`). This is the default returned by `Sampler` for *types*.
2. `SamplerTrivial(self)` is a simple wrapper for `self`, which can be accessed with `[]`. This is the recommended sampler when no pre-computed information is needed (e.g. `rand(1:3)`), and is the default returned by `Sampler` for *values*.
3. `SamplerSimple(self, data)` also contains the additional `data` field, which can be used to store arbitrary pre-computed values, which should be computed in a *custom method* of `Sampler`.

We provide examples for each of these. We assume here that the choice of algorithm is independent of the RNG, so we use `AbstractRNG` in our signatures.

`Random.Sampler` — Type

```
Sampler(rng, x, repetition = Val(Inf))
```

Return a sampler object that can be used to generate random values from `rng` for `x`.

When `sp = Sampler(rng, x, repetition)`, `rand(rng, sp)` will be used to draw random values, and should be defined accordingly.

`repetition` can be `Val(1)` or `Val(Inf)`, and should be used as a suggestion for deciding the amount of precomputation, if applicable.

`Random.SamplerType` and `Random.SamplerTrivial` are default fallbacks for *types* and *values*, respectively. `Random.SamplerSimple` can be used to store pre-computed values without defining extra types for only this purpose.

`Random.SamplerType` — Type

```
SamplerType{T}()
```

A sampler for types, containing no other information. The default fallback for `Sampler` when called with types.

`Random.SamplerTrivial` — Type

```
SamplerTrivial(x)
```

Create a sampler that just wraps the given value `x`. This is the default fall-back for values. The `eltype` of this sampler is equal to `eltype(x)`.

The recommended use case is sampling from values without precomputed data.

`Random.SamplerSimple` — Type

```
SamplerSimple(x, data)
```

Create a sampler that wraps the given value `x` and the `data`. The `eltype` of this sampler is equal to `eltype(x)`.

The recommended use case is sampling from values with precomputed data.

Decoupling pre-computation from actually generating the values is part of the API, and is also available to the user. As an example, assume that `rand(rng, 1:20)` has to be called repeatedly in a loop: the way to take advantage of this decoupling is as follows:

```
rng = MersenneTwister()
sp = Random.Sampler(rng, 1:20) # or Random.Sampler(MersenneTwister, 1:20)
for x in X
    n = rand(rng, sp) # similar to n = rand(rng, 1:20)
    # use n
```

```
end
```

This is the mechanism that is also used in the standard library, e.g. by the default implementation of random array generation (like in `rand(1:20, 10)`).

Generating values from a type

Given a type `T`, it's currently assumed that if `rand(T)` is defined, an object of type `T` will be produced. `SamplerType` is the *default sampler for types*. In order to define random generation of values of type `T`, the `rand(rng::AbstractRNG, ::Random.SamplerType{T})` method should be defined, and should return values what `rand(rng, T)` is expected to return.

Let's take the following example: we implement a `Die` type, with a variable number `n` of sides, numbered from `1` to `n`. We want `rand(Die)` to produce a `Die` with a random number of up to 20 sides (and at least 4):

```
struct Die
    nsides::Int # number of sides
end

Random.rand(rng::AbstractRNG, ::Random.SamplerType{Die}) = Die(rand(rng, 4:20))

# output
```

Scalar and array methods for `Die` now work as expected:

```
julia> rand(Die)
Die(10)

julia> rand(MersenneTwister(0), Die)
Die(16)

julia> rand(Die, 3)
3-element Array{Die,1}:
 Die(5)
 Die(20)
 Die(9)

julia> a = Vector{Die}(undef, 3); rand!(a)
3-element Array{Die,1}:
 Die(11)
 Die(20)
 Die(10)
```

A simple sampler without pre-computed data

Here we define a sampler for a collection. If no pre-computed data is required, it can be implemented with a `SamplerTrivial` sampler, which is in fact the *default fallback for values*.

In order to define random generation out of objects of type `S`, the following method should be defined: `rand(rng::AbstractRNG, sp::Random.SamplerTrivial{S})`. Here, `sp` simply wraps an object of type `S`, which can be accessed via `sp[]`. Continuing the `Die` example, we want now to define `rand(d::Die)` to produce an `Int` corresponding to one of `d`'s sides:

```
julia> Random.rand(rng::AbstractRNG, d::Random.SamplerTrivial{Die}) = rand(rng, 1:d[

julia> rand(Die(4))
2

julia> rand(Die(4), 3)
3-element Array{Any,1}:
 1
 4
 2
```

Given a collection type `S`, it's currently assumed that if `rand(::S)` is defined, an object of type `eltype(S)` will be produced. In the last example, a `Vector{Any}` is produced; the reason is that `eltype(Die) == Any`. The remedy is to define `Base.eltype(::Type{Die}) = Int`.

Generating values for an `AbstractFloat` type

`AbstractFloat` types are special-cased, because by default random values are not produced in the whole type domain, but rather in `[0,1)`. The following method should be implemented for `T <: AbstractFloat`: `Random.rand(::AbstractRNG, ::Random.SamplerTrivial{Random.CloseOpen01{T}})`

An optimized sampler with pre-computed data

Consider a discrete distribution, where numbers `1:n` are drawn with given probabilities that sum to one. When many values are needed from this distribution, the fastest method is using an alias table. We don't provide the algorithm for building such a table here, but suppose it is available in `make_alias_table(probabilities)` instead, and `draw_number(rng, alias_table)` can be used to draw a random number from it.

Suppose that the distribution is described by

```
struct DiscreteDistribution{V <: AbstractVector}
```

```
    probabilities::V
end
```

and that we *always* want to build an alias table, regardless of the number of values needed (we learn how to customize this below). The methods

```
Random.eltype(::Type{<:DiscreteDistribution}) = Int

function Random.Sampler(::Type{<:AbstractRNG}, distribution::DiscreteDistribution,
    SamplerSimple(disribution, make_alias_table(distribution.probabilities))
end
```

should be defined to return a sampler with pre-computed data, then

```
function rand(rng::AbstractRNG, sp::SamplerSimple{<:DiscreteDistribution})
    draw_number(rng, sp.data)
end
```

will be used to draw the values.

Custom sampler types

The `SamplerSimple` type is sufficient for most use cases with precomputed data. However, in order to demonstrate how to use custom sampler types, here we implement something similar to `SamplerSimple`.

Going back to our `Die` example: `rand(::Die)` uses random generation from a range, so there is an opportunity for this optimization. We call our custom sampler `SamplerDie`.

```
import Random: Sampler, rand

struct SamplerDie <: Sampler{Int} # generates values of type Int
    die::Die
    sp::Sampler{Int} # this is an abstract type, so this could be improved
end

Sampler(RNG::Type{<:AbstractRNG}, die::Die, r::Random.Repetition) =
    SamplerDie(die, Sampler(RNG, 1:die.nsides, r))
# the `r` parameter will be explained later on

rand(rng::AbstractRNG, sp::SamplerDie) = rand(rng, sp.sp)
```

It's now possible to get a sampler with `sp = Sampler(rng, die)`, and use `sp` instead of `die` in any `rand` call involving `rng`. In the simplistic example above, `die` doesn't need to be stored in `SamplerDie` but this is often the case in practice.

Of course, this pattern is so frequent that the helper type used above, namely `Random.SamplerSimple`, is available, saving us the definition of `SamplerDie`: we could have implemented our decoupling with:

```
Sampler(RNG::Type{<:AbstractRNG}, die::Die, r::Random.Repetition) =
    SamplerSimple(die, Sampler(RNG, 1:die.nsides, r))

rand(rng::AbstractRNG, sp::SamplerSimple{Die}) = rand(rng, sp.data)
```

Here, `sp.data` refers to the second parameter in the call to the `SamplerSimple` constructor (in this case equal to `Sampler(rng, 1:die.nsides, r)`), while the `Die` object can be accessed via `sp[]`.

Like `SamplerDie`, any custom sampler must be a subtype of `Sampler{T}` where `T` is the type of the generated values. Note that `SamplerSimple(x, data) isa Sampler{eltype(x)}`, so this constrains what the first argument to `SamplerSimple` can be (it's recommended to use `SamplerSimple` like in the `Die` example, where `x` is simply forwarded while defining a `Sampler` method). Similarly, `SamplerTrivial(x) isa Sampler{eltype(x)}`.

Another helper type is currently available for other cases, `Random.SamplerTag`, but is considered as internal API, and can break at any time without proper deprecations.

Using distinct algorithms for scalar or array generation

In some cases, whether one wants to generate only a handful of values or a large number of values will have an impact on the choice of algorithm. This is handled with the third parameter of the `Sampler` constructor. Let's assume we defined two helper types for `Die`, say `SamplerDie1` which should be used to generate only few random values, and `SamplerDieMany` for many values. We can use those types as follows:

```
Sampler(RNG::Type{<:AbstractRNG}, die::Die, ::Val{1}) = SamplerDie1(...)
Sampler(RNG::Type{<:AbstractRNG}, die::Die, ::Val{Inf}) = SamplerDieMany(...)
```

Of course, `rand` must also be defined on those types (i.e. `rand(::AbstractRNG, ::SamplerDie1)` and `rand(::AbstractRNG, ::SamplerDieMany)`). Note that, as usual, `SamplerTrivial` and `SamplerSimple` can be used if custom types are not necessary.

Note: `Sampler(rng, x)` is simply a shorthand for `Sampler(rng, x, Val(Inf))`, and `Random.Repetition` is an alias for `Union{Val{1}, Val{Inf}}`.

Creating new generators

The API is not clearly defined yet, but as a rule of thumb:

1. any `rand` method producing "basic" types (`isbitstype` integer and floating types in `Base`) should be defined for this specific RNG, if they are needed;
2. other documented `rand` methods accepting an `AbstractRNG` should work out of the box, (provided the methods from 1) what are relied on are implemented), but can of course be specialized for this RNG if there is room for optimization;
3. `copy` for pseudo-RNGs should return an independent copy that generates the exact same random sequence as the original from that point when called in the same way. When this is not feasible (e.g. hardware-based RNGs), `copy` must not be implemented.

Concerning 1), a `rand` method may happen to work automatically, but it's not officially supported and may break without warnings in a subsequent release.

To define a new `rand` method for an hypothetical `MyRNG` generator, and a value specification `s` (e.g. `s == Int`, or `s == 1:10`) of type `S==typeof(s)` or `S==Type{s}` if `s` is a type, the same two methods as we saw before must be defined:

1. `Sampler(::Type{MyRNG}, ::S, ::Repetition)`, which returns an object of type say `SamplerS`
2. `rand(rng::MyRNG, sp::SamplerS)`

It can happen that `Sampler(rng::AbstractRNG, ::S, ::Repetition)` is already defined in the `Random` module. It would then be possible to skip step 1) in practice (if one wants to specialize generation for this particular RNG type), but the corresponding `SamplerS` type is considered as internal detail, and may be changed without warning.

Specializing array generation

In some cases, for a given RNG type, generating an array of random values can be more efficient with a specialized method than by merely using the decoupling technique explained before. This is for example the case for `MersenneTwister`, which natively writes random values in an array.

To implement this specialization for `MyRNG` and for a specification `s`, producing elements of type `S`, the following method can be defined: `rand!(rng::MyRNG, a::AbstractArray{S}, ::SamplerS)`, where `SamplerS` is the type of the sampler returned by `Sampler(MyRNG, s, Val(Inf))`. Instead of `AbstractArray`, it's possible to implement the functionality only for a subtype, e.g. `Array{S}`. The non-mutating array method of `rand` will automatically call this specialization internally.

Reproducibility

By using an RNG parameter initialized with a given seed, you can reproduce the same pseudorandom number sequence when running your program multiple times. However, a minor release of Julia (e.g. 1.3 to 1.4) *may change* the sequence of pseudorandom numbers generated from a specific seed. (Even if the sequence produced by a low-level function like `rand` does not change, the output of higher-level functions like `randsubseq` may change due to algorithm updates.) Rationale: guaranteeing that pseudorandom streams never change prohibits many algorithmic improvements.

If you need to guarantee exact reproducibility of random data, it is advisable to simply *save the data* (e.g. as a supplementary attachment in a scientific publication). (You can also, of course, specify a particular Julia version and package manifest, especially if you require bit reproducibility.)

Software tests that rely on *specific* "random" data should also generally save the data or embed it into the test code. On the other hand, tests that should pass for *most* random data (e.g. testing `A \ (A*x) ≈ x` for a random matrix `A = randn(n,n)`) can use an RNG with a fixed seed to ensure that simply running the test many times does not encounter a failure due to very improbable data (e.g. an extremely ill-conditioned matrix).

The statistical *distribution* from which random samples are drawn *is* guaranteed to be the same across any minor Julia releases.

« The Julia REPL SHA »