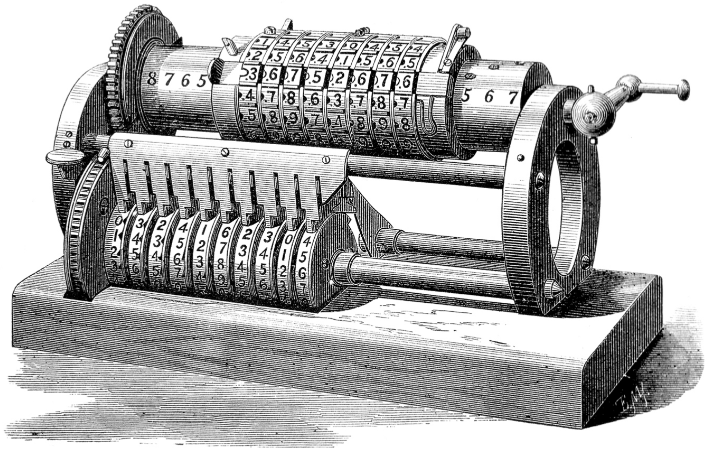
LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

Programming in R



Unit 9: Randomness

Reproducibility

Remember, this course has multiple goals:

- Learn things about the R language: "R"
- Get to know nice tools to use: "Tools"
- Learn things about software development in general: "Dev"

This unit:

- "R" Track: Random Numbers

R Track

Random Numbers

Random Numbers

There are many reasons why your code could want to have **randomness**

- Sample points from a probability distribution for a simulation
- Probabilistic algorithms, e.g. for estimating the expected value of something by randomly evaluating points and taking their average
- Simulate dice or shuffled cards for a game
- Create a password or key for encryption

Random Numbers

This is fundamentally at odds with our concept of computers and how we program them: programs are *deterministic*.

Solutions

- Use hardware-generated randomness, such as electrical noise in an analog circuit -- slow and expensive!
- Use a pseudorandom number generator (PRNG, sometimes just called RNG): produce numbers that are calculated deterministically, but that have a pattern so complicated that they are essentially random for their purpose -- fast, but not "really" "random"
  - Like using the digits of pi, or digits of sqrt(2): deterministic, but not correlated with most things one would be doing with it
  - Actual algorithms are documented / referenced in `?Random`.
- Pseudorandom numbers in R: generated with functions starting with 'r' (runif, rexp, rbinom, etc. -- see `?Distributions`), as well as 'sample' / 'sample.int'; some others.

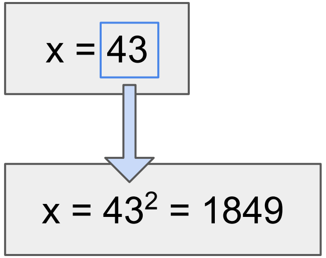
Example PRNG: "Middle-Square Method"

1. Take a number "x" with n digits (e.g. n = 2)

x = 43

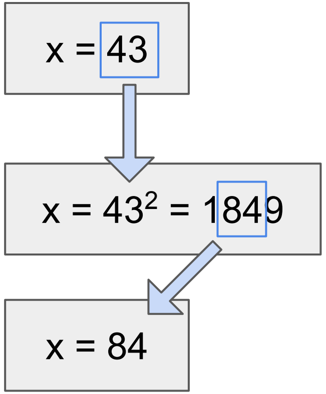
Example PRNG: "Middle-Square Method"

1. Take a number "x" with n digits (e.g. n = 2)
2. set x ➞ x^2



Example PRNG: "Middle-Square Method"

1. Take a number "x" with n digits (e.g. n = 2)
2. set x ➞ x^2
   Restrict x to the middle n digits



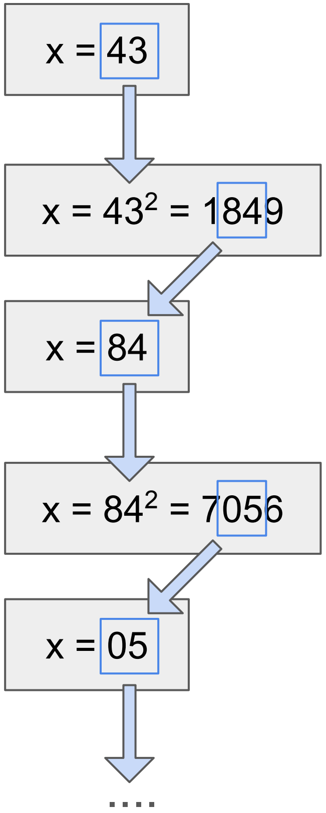
$x = \boxed{43}$

$x = 43^2 = 1\boxed{84}9$

$x = 84$

Example PRNG: "Middle-Square Method"

1. Take a number "x" with n digits (e.g. n = 2)
2. set x ➞ x^2
3. Restrict x to the middle n digits
4. Repeat from 2.



x = 43

x = 43^2 = 1849

x = 84

x = 84^2 = 7056

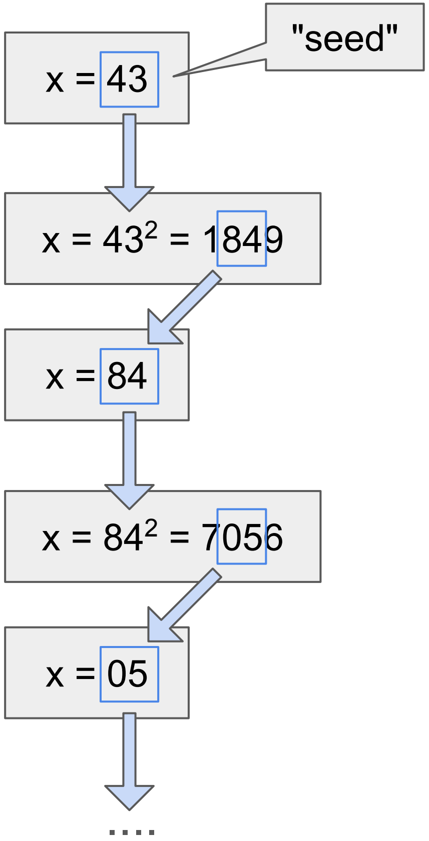
x = 05

....

Example PRNG: "Middle-Square Method"

1. Take a number "x" with n digits (e.g. n = 2)
2. set x ➞ x^2
3. Restrict x to the middle n digits
4. Repeat from 2.

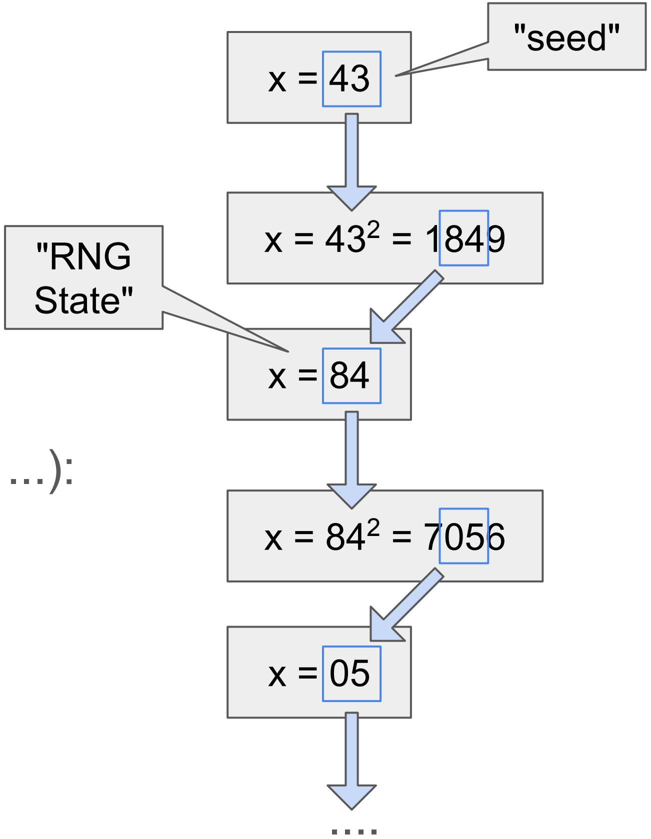
- Initial x is our "**seed**"



Example PRNG: "Middle-Square Method"

1. Take a number "x" with n digits (e.g. n = 2)
2. set x → x^2
3. Restrict x to the middle n digits
4. Repeat from 2.

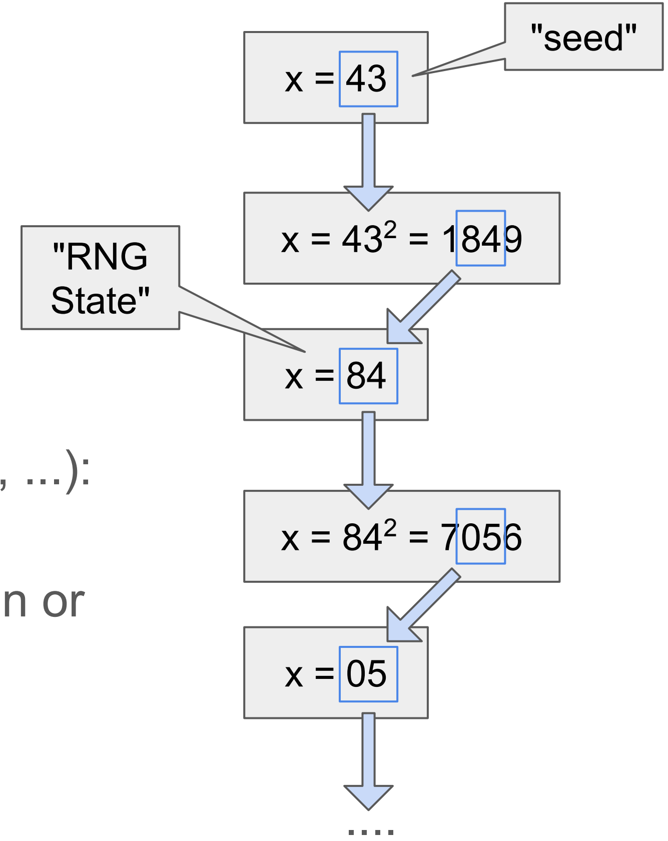
- Initial x is our "**seed**"
- We get a series of x values (43, 84, 05, 25, 62, ...): "RNG **State**"



Example PRNG: "Middle-Square Method"

1. Take a number "x" with n digits (e.g. n = 2)
2. set x ➞ x^2
3. Restrict x to the middle n digits
4. Repeat from 2.

- Initial x is our "**seed**"
- We get a series of x values (43, 84, 05, 25, 62, ...):
  "RNG **State**"
- We could simulate a coin toss: is the state even or odd?
  ➞ H, T, H, H, T, T,



Example PRNG: "Middle-Square Method"

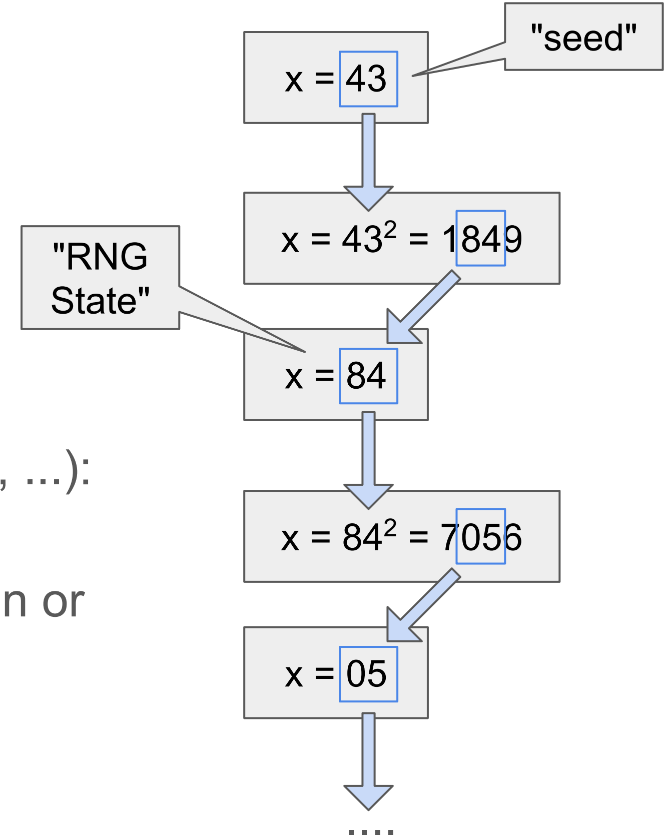
1. Take a number "x" with n digits (e.g. n = 2)
2. set x → x^2
3. Restrict x to the middle n digits
4. Repeat from 2.

- Initial x is our "**seed**"
- We get a series of x values (43, 84, 05, 25, 62, ...):

  "RNG **State**"
- We could simulate a coin toss: is the state even or odd?

  → H, T, H, H, T, T,

  (43, 84, 05, 25, 62, 84, ...)



Example PRNG: "Middle-Square Method"

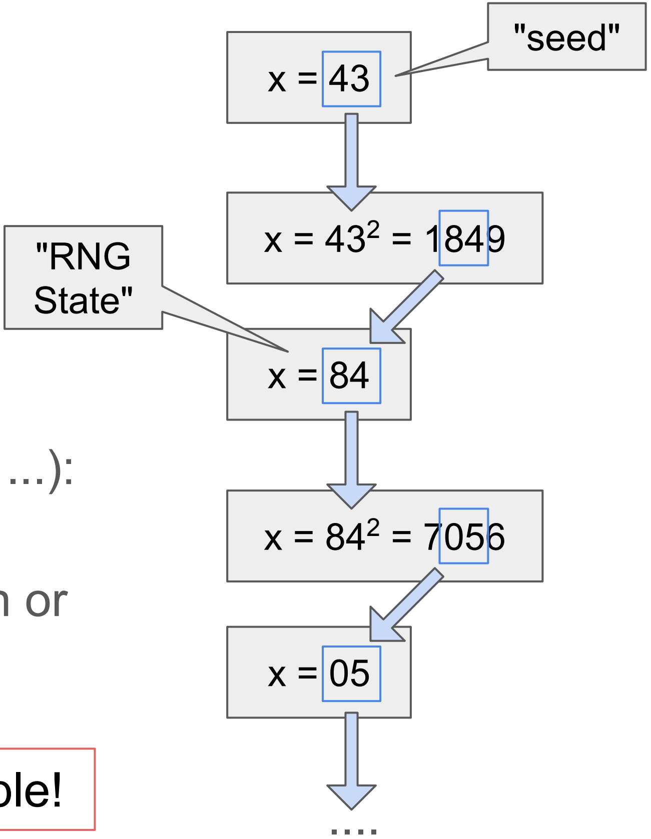
1. Take a number "x" with n digits (e.g. n = 2)
2. set x → x^2
3. Restrict x to the middle n digits
4. Repeat from 2.

- Initial x is our "**seed**"
- We get a series of x values (43, 84, 05, 25, 62, ...): "RNG **State**"
- We could simulate a coin toss: is the state even or odd?

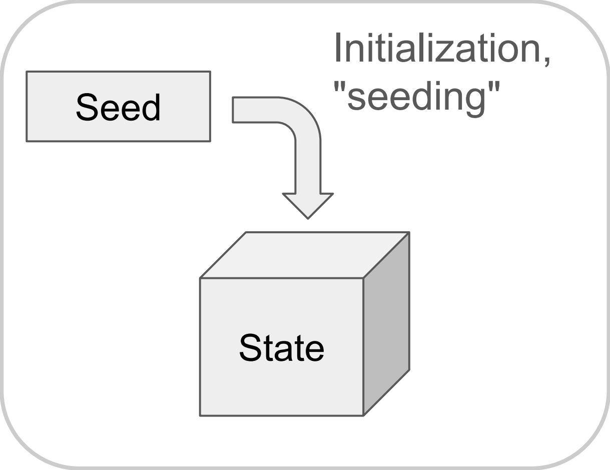
  → H, T, H, H, T, T,

  (43, 84, 05, 25, 62, 84, ...)

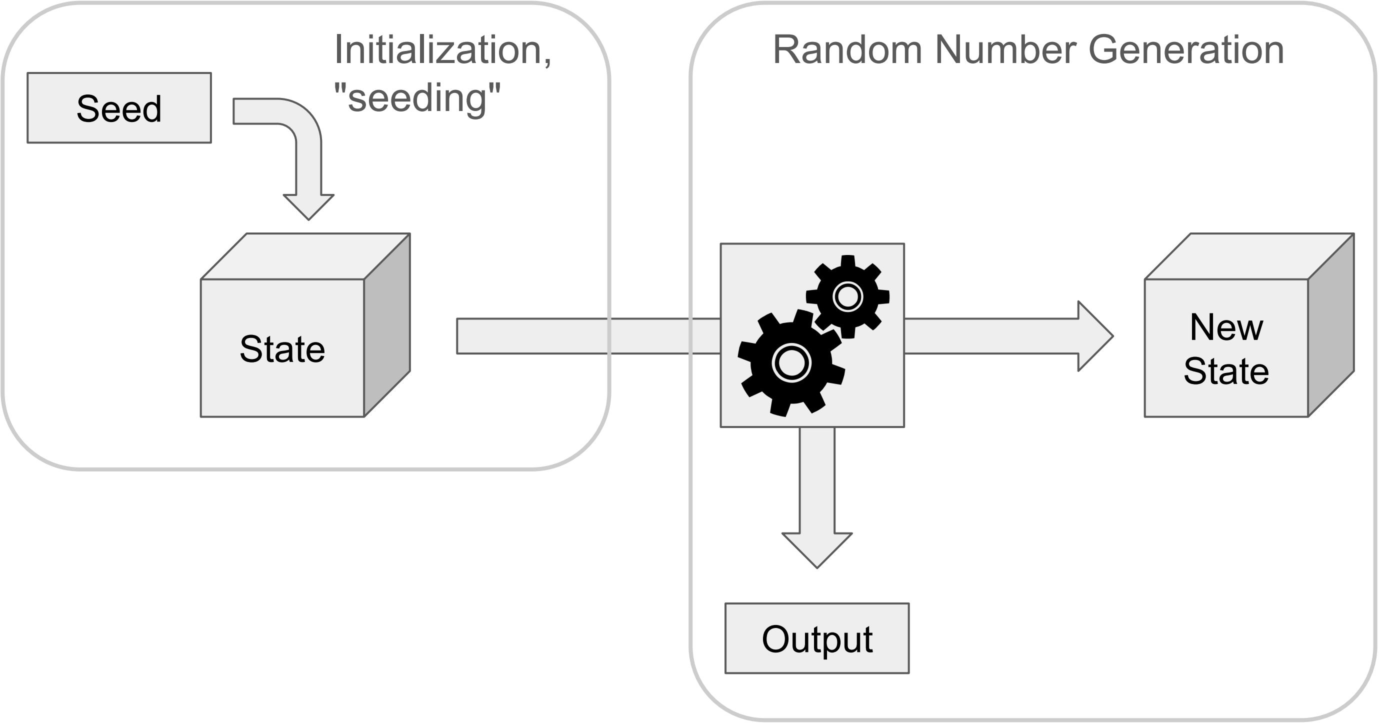
This is a toy-example!



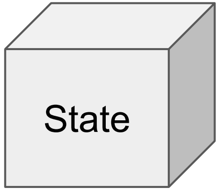
PRNG Generally



PRNG Generally

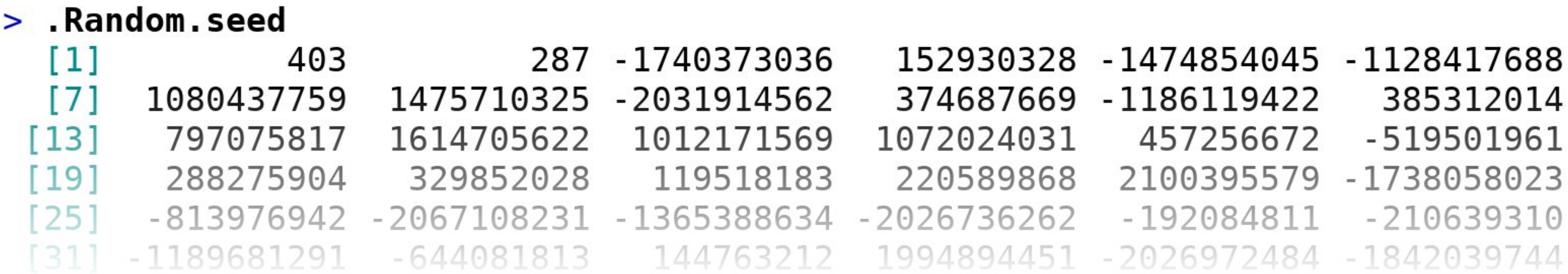


PRNG in R: Seeding



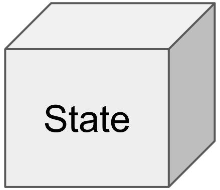
`.Random.seed`

- see it with `ls(all.names = TRUE)`
- only exists once you `set.seed()` or use a random function
- A large vector of numbers!



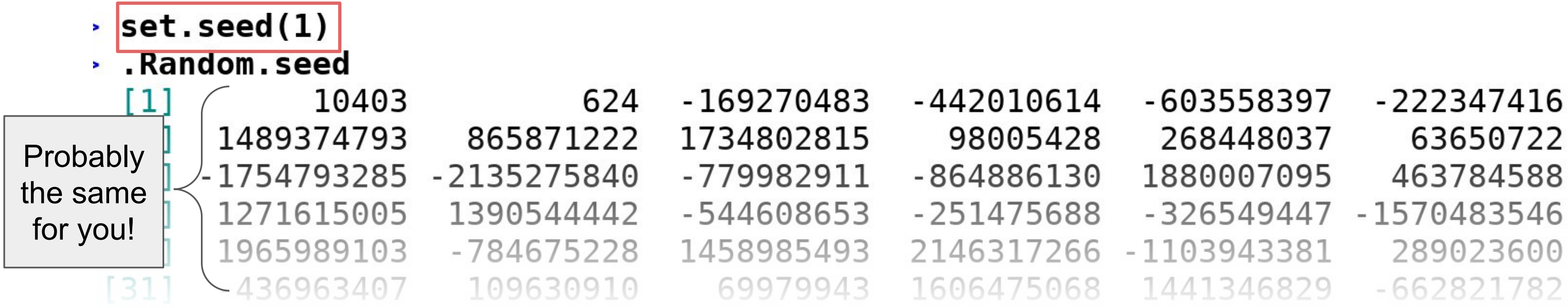
```
> .Random.seed
  [1]        403        287 -1740373036  152930328 -1474854045 -1128417688
  [7] 1080437759 1475710325 -2031914562  374687669 -1186119422  385312014
 [13]  797075817 1614705622  1012171569 1072024031  457256672  -519501961
 [19]  288275904  329852028   119518183  220589868 2100395579 -1738058023
 [25] -813976942 -2067108231 -1365388634 -2026736262 -192084811 -210639310
 [31] -1189681291 -644081813  144763212 1994894451 -2026972484 -1842039744
```

PRNG in R: Seeding



`.Random.seed`

- see it with `ls(all.names = TRUE)`
- only exists once you `set.seed()` or use a random function
- A large vector of numbers!
- When using the same RNG algorithm (`RNGkind()`), **the same seed will lead to the same initial state**

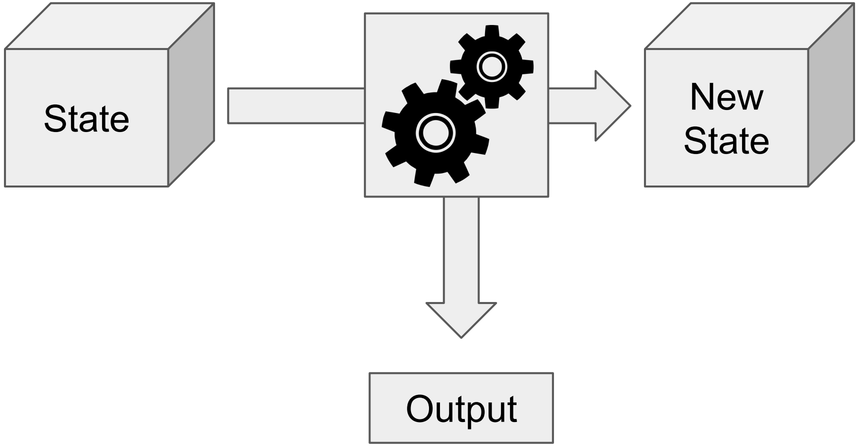


```
> set.seed(1)
> .Random.seed
  [1]       10403         624  -169270483  -442010614  -603558397  -222347416
    ]  1489374793   865871222  1734802815    98005428   268448037    63650722
    ] -1754793285 -2135275840  -779982911  -864886130  1880007095   463784588
    ]  1271615005  1390544442  -544608653  -251475688  -326549447 -1570483546
    ]  1965989103  -784675228  1458985493  2146317266 -1103943381   289023600
 [31]  -436963407   109630910    69979943  1606475068  1441346829  -662821782
```

Probably the same for you!

PRNG in R: Drawing

- Functions starting with `r`:
  - `runif()`: uniform distribution
  - `rexp()`: exponential distribution
  - `rnorm()`: normal distribution
  - ...



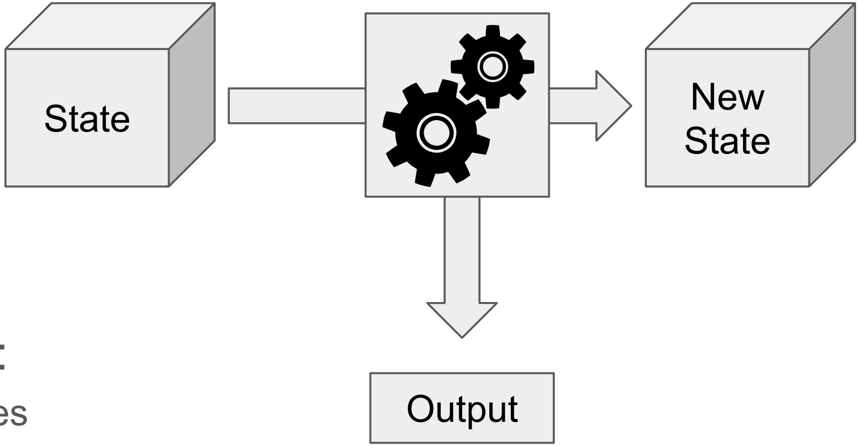
PRNG in R: Drawing

- Functions starting with `r`:
  - `runif()`: uniform distribution
  - `rexp()`: exponential distribution
  - `rnorm()`: normal distribution
  - ...
- Internally, when drawing random value:
  - draw one or more uniformly distributed values
  - build other random variables from this.

    One could, e.g., do:

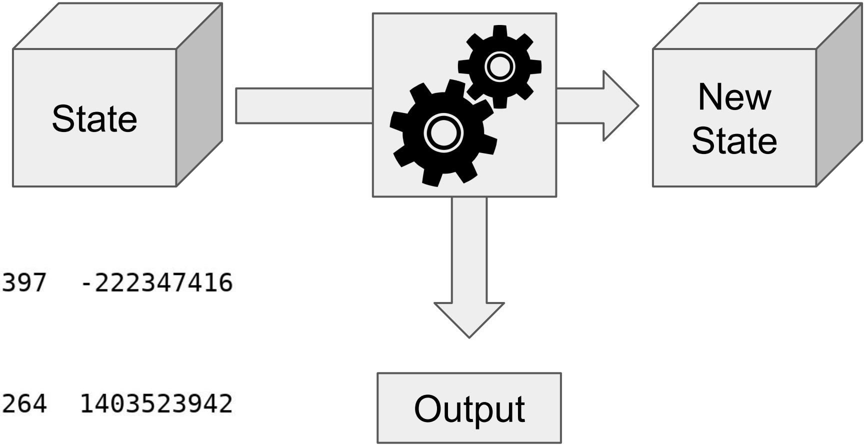
    `rexp() = -log(runif())`

    (R does something like this but more complicated)



PRNG in R: Drawing

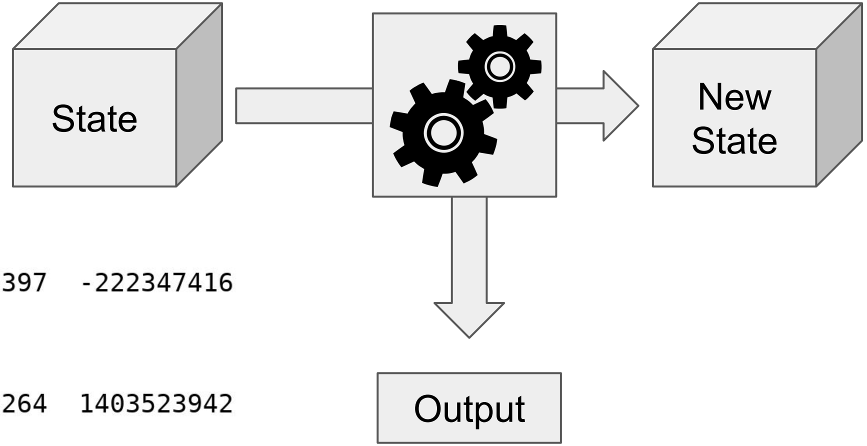
Drawing RVs advances the state:



```
> set.seed(1)
> head(.Random.seed)
[1]       10403         624  -169270483  -442010614  -603558397  -222347416
> runif(1)
[1] 0.2655087
> head(.Random.seed)
[1]       10403           1  1654269195 -1877109783  -961256264  1403523942
```

PRNG in R: Drawing

Drawing RVs advances the state:



```
> set.seed(1)
> head(.Random.seed)
[1]       10403         624  -169270483  -442010614  -603558397  -222347416
> runif(1)
[1] 0.2655087
> head(.Random.seed)
[1]       10403           1  1654269195 -1877109783  -961256264  1403523942
```

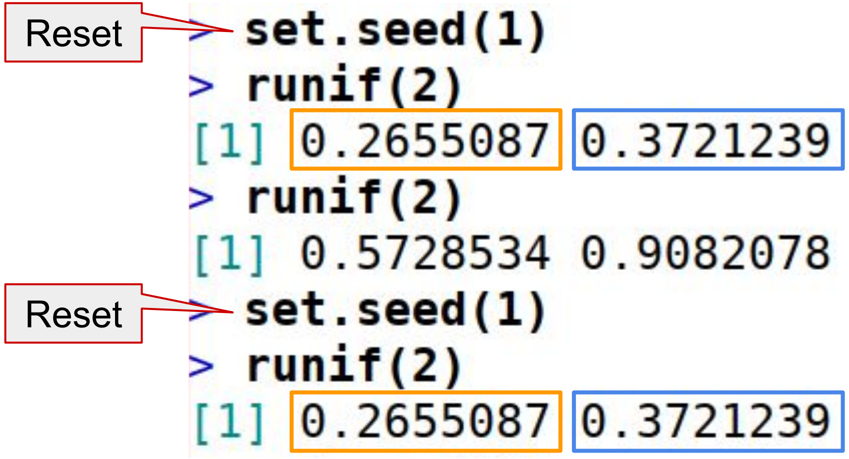
This is deterministic! Do it again, it advances the same amount.

```
> set.seed(1)
> head(.Random.seed)
[1]       10403         624  -169270483  -442010614  -603558397  -222347416
> runif(1)
[1] 0.2655087
> head(.Random.seed)
[1]       10403           1  1654269195 -1877109783  -961256264  1403523942
```

PRNG in R: Drawing

Important for reproducibility:

- Setting the same seed gives the same random values.
- Calling the same sequence of random functions in the same order(!) yields the same sequence of results
- The RNG state advances in discrete steps



```
> set.seed(1)
> runif(2)
[1] 0.2655087 0.3721239
> runif(2)
[1] 0.5728534 0.9082078
> set.seed(1)
> runif(2)
[1] 0.2655087 0.3721239
```

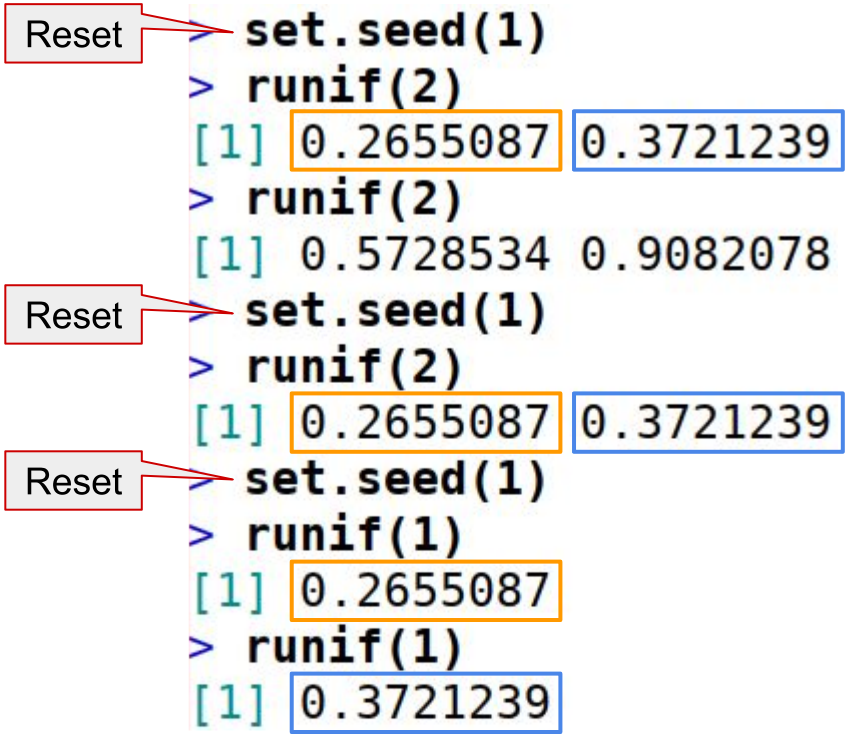
Reset

Reset

PRNG in R: Drawing

Important for reproducibility:

- Setting the same seed gives the same random values.
- Calling the same sequence of random functions in the same order(!) yields the same sequence of results
- The RNG state advances in discrete steps



Reset

Reset

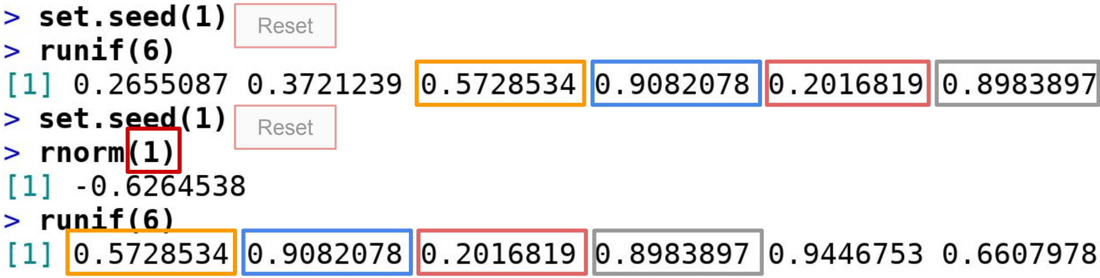
Reset

```
> set.seed(1)
> runif(2)
[1] 0.2655087 0.3721239
> runif(2)
[1] 0.5728534 0.9082078
> set.seed(1)
> runif(2)
[1] 0.2655087 0.3721239
> set.seed(1)
> runif(1)
[1] 0.2655087
> runif(1)
[1] 0.3721239
```

PRNG in R: Drawing

Important for reproducibility:

- Setting the same seed gives the same random values.
- Calling the same sequence of random functions in the same order(!) yields the same sequence of results
- The RNG state advances in discrete steps
- Some calls advance the state more than others

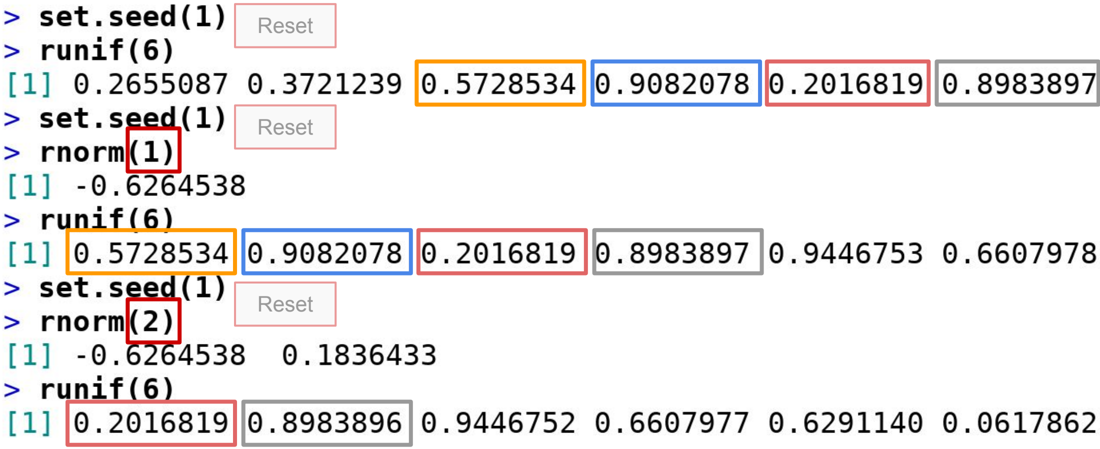


```
> set.seed(1)
> runif(6)
[1] 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819 0.8983897
> set.seed(1)
> rnorm(1)
[1] -0.6264538
> runif(6)
[1] 0.5728534 0.9082078 0.2016819 0.8983897 0.9446753 0.6607978
```

PRNG in R: Drawing

Important for reproducibility:

- Setting the same seed gives the same random values.
- Calling the same sequence of random functions in the same order(!) yields the same sequence of results
- The RNG state advances in discrete steps
- Some calls advance the state more than others

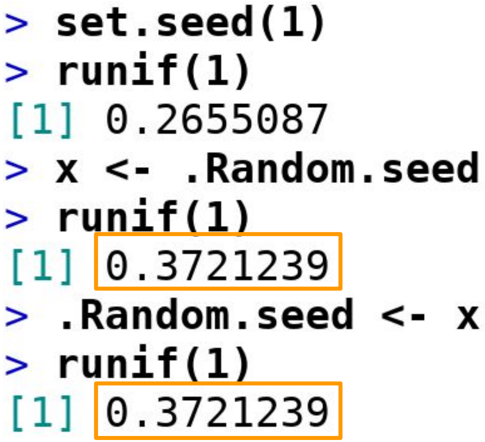


```
> set.seed(1)
> runif(6)
[1] 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819 0.8983897
> set.seed(1)
> rnorm(1)
[1] -0.6264538
> runif(6)
[1] 0.5728534 0.9082078 0.2016819 0.8983897 0.9446753 0.6607978
> set.seed(1)
> rnorm(2)
[1] -0.6264538  0.1836433
> runif(6)
[1] 0.2016819 0.8983896 0.9446752 0.6607977 0.6291140 0.0617862
```

PRNG in R: Drawing

Important for reproducibility:

- Setting the same seed gives the same random values.
- Calling the same sequence of random functions in the same order(!) yields the same sequence of results
- The RNG state advances in discrete steps
- Some calls advance the state more than others
- `.Random.seed` saves the state
- but beware that some random functions may "cache" their values & have addnl. state -- see `?Random` --> use `set.seed()`!



```
> set.seed(1)
> runif(1)
[1] 0.2655087
> x <- .Random.seed
> runif(1)
[1] 0.3721239
> .Random.seed <- x
> runif(1)
[1] 0.3721239
```

Using PRNGs

Usecase: Hypothesis testing & p-value estimation

- You measure a statistic that is transformed in a complicated way?
  - e.g. rounding, cutoffs, ...
  - -> Simulate the data under H0 to calculate p-value!
- *Bootstrapping*: draw samples with replacement and fit models
  - `sample(values, replace = TRUE)`
- Permutation tests
  - Is data column X relevant for a given result?
  - Shuffle this column and run the analysis again (x1000) and see what changes
  - (Does not always work: E.g. if X and Y are almost copies of each other, shuffling only one will sometimes have no visible effect)

Usecase: Random Search

- You have something with "hyperparameters" -- settings that influence performance

  E.g. `f(x, y, z)`
- Often some of them are important, others have no influence

  E.g. `x`, `y` have an influence, `z` does not

  In general, you don't know which!
- If you try a grid of 10 values each, you need to evaluate `f()` 10 x 10 x 10 times (in general: 10^N)
- If you randomly sample values, you get a good resolution in ~ 10 x 10 evals!

[Bergstra & Bengio 2012]

Usecase: Visualization & Data Exploration

- If you have a very large dataset
- You want to explore it in some way:
  - plot it along various axes
  - try out various methods
- You often get a good approximation of what it looks like / what works well if you consider a small subset
- -> Randomly subsample your data

Usecase: More Specific Algorithms

Many algorithms make use of randomness more specifically:

- Markov-Chain Monte-Carlo -- Sample from & integrate complex probability distributions, relevant in Bayesian modelling
- Evolutionary algorithms: Quite robust methods for optimization
- Data augmentation techniques

These are often implemented in packages, you don't need to write these yourself.

Distributions in R

	Random Value	Probability (density)	Cumulative d.f., $P(X \le x)$	Quantile function, F^{-1}
Uniform dist.	`runif()`	`dunif()`	`punif()`	`qunif()`
Normal dist.	`rnorm()`	`dnorm()`	`pnorm()`	`qnorm()`
Exponential	`rexp()`	...	...	...
Binomial	`rbinom()`	...		
Geometric	`rgeom()`	...		
Poisson	`rpois()`	...		
...	...			

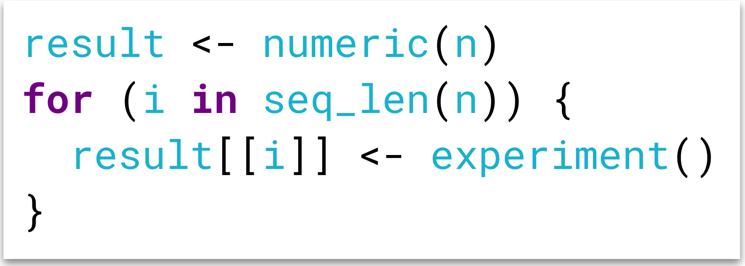
Random Numbers in R: Recipes

Do a random experiment 100 times

Random Numbers in R: Recipes

Do a random experiment 100 times

Dont:

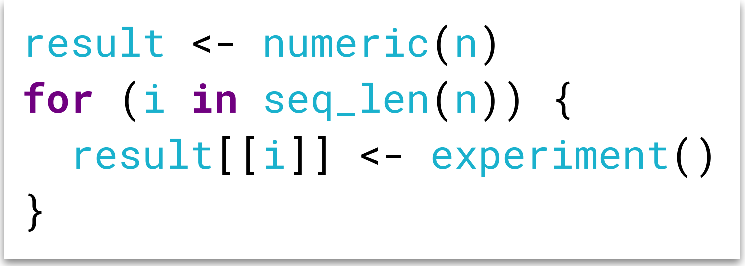


```r
result <- numeric(n)
for (i in seq_len(n)) {
  result[[i]] <- experiment()
}
```

Random Numbers in R: Recipes

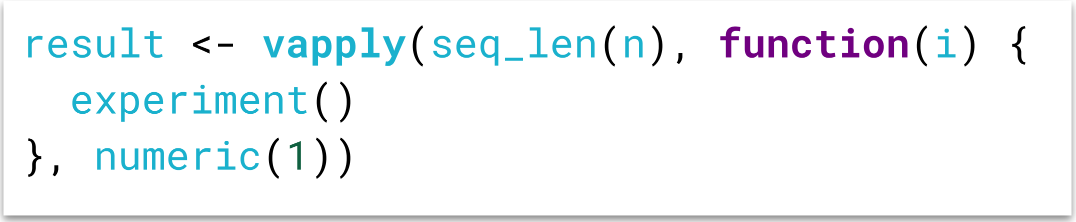
Do a random experiment 100 times

Dont:



```r
result <- numeric(n)
for (i in seq_len(n)) {
  result[[i]] <- experiment()
}
```

or:

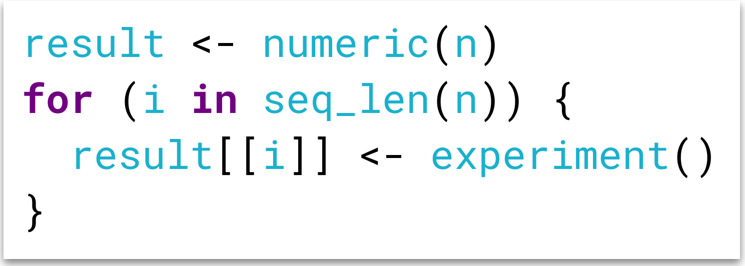


```r
result <- vapply(seq_len(n), function(i) {
  experiment()
}, numeric(1))
```

Random Numbers in R: Recipes

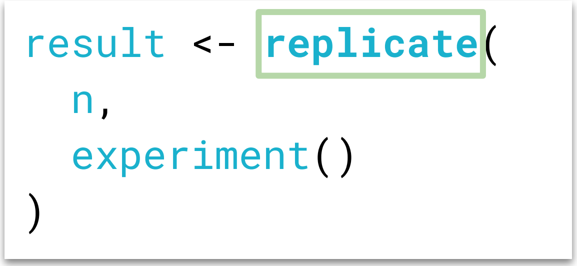
Do a random experiment 100 times

Dont:



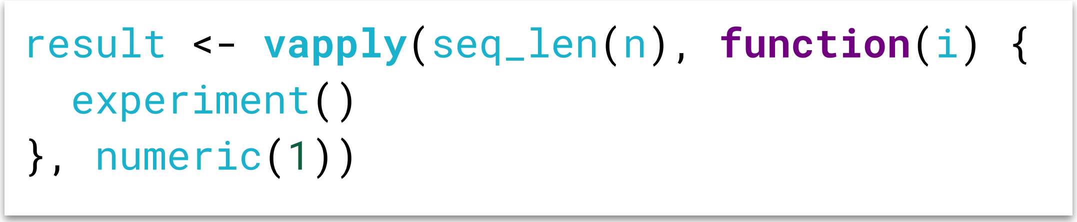
```r
result <- numeric(n)
for (i in seq_len(n)) {
  result[[i]] <- experiment()
}
```

Do:



```r
result <- replicate(
  n,
  experiment()
)
```

or:



```r
result <- vapply(seq_len(n), function(i) {
  experiment()
}, numeric(1))
```

Random Numbers in R: Recipes

Do a random experiment 100 times

Dont: ☹

```r
result <- numeric(n)
for (i in seq_len(n)) {
  result[[i]] <- experiment()
}
```

or: ☹

```r
result <- vapply(seq_len(n), function(i) {
  experiment()
}, numeric(1))
```

Do: ☺

```r
result <- replicate(
  n,
  experiment()
)
```

Use `replicate()` if your expression does not depend on `i`!

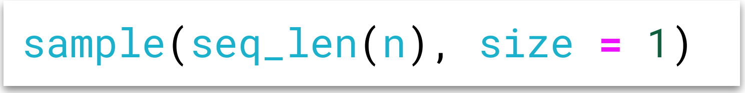
Random Numbers in R: Recipes

Get a random integer between 1 and n

Random Numbers in R: Recipes

Get a random integer between 1 and n

Dont:

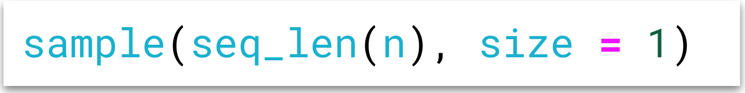


```r
sample(seq_len(n), size = 1)
```

Random Numbers in R: Recipes

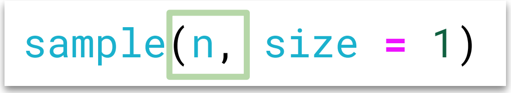
Get a random integer between 1 and n

Dont:



```
sample(seq_len(n), size = 1)
```

Do:

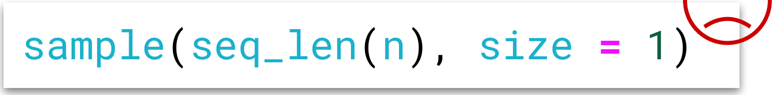


```
sample(n, size = 1)
```

Random Numbers in R: Recipes

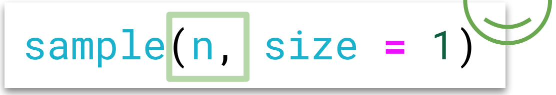
Get a random integer between 1 and n

Dont:



```r
sample(seq_len(n), size = 1)
```

Do:



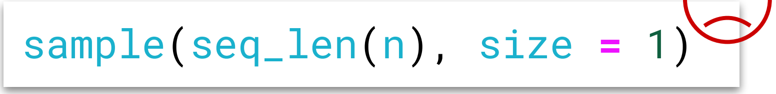
```r
sample(n, size = 1)
```

`sample()` with a scalar n samples from 1 to n by itself

Random Numbers in R: Recipes

Get a random integer between 1 and n

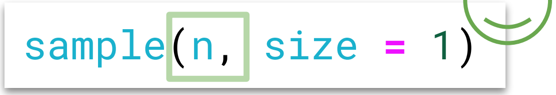
Dont:



```
sample(seq_len(n), size = 1)
```

☹

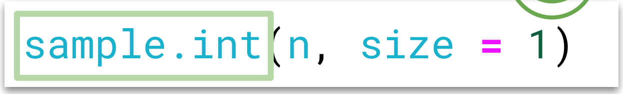
Do:



```
sample(n, size = 1)
```

☺

or:



```
sample.int(n, size = 1)
```

☺ ++

`sample()` with a scalar n samples from 1 to n by itself

Random Numbers in R: Recipes

Shuffle a vector v

Random Numbers in R: Recipes

Shuffle a vector v

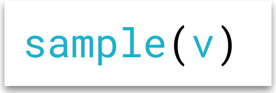
Dont:

```
sample(v)
```

Random Numbers in R: Recipes

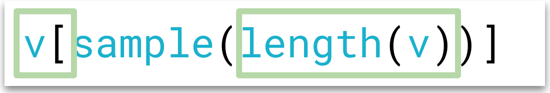
Shuffle a vector v

Dont:



```
sample(v)
```

Do:



```
v[sample(length(v))]
```

Random Numbers in R: Recipes

Shuffle a vector v

Dont: ☹

```
sample(v)
```

Do: ☺

```
v[sample(length(v))]
```

If v happens to be length 1 (scalar) then `sample()` turns it into `seq_len(v)`!

Random Numbers in R: Recipes

Shuffle a vector `v`

Dont: ☹

```
sample(v)
```

Do: ☺

```
v[sample(length(v))]
```

If `v` happens to be length 1 (scalar) then `sample()` turns it into `seq_len(v)`!

!! Important !!

Random Numbers in R: Recipes

Shuffle a vector v

Dont: ☹

```
sample(v)
```

Do: ☺

```
v[sample(length(v))]
```

or: ☺ ++

```
v[sample.int(length(v))]
```

If v happens to be length 1 (scalar) then `sample()` turns it into `seq_len(v)`!

!! Important !!

Random Numbers in R: Recipes

Put rows of table in random order:

```
iris[sample.int(nrow(iris)), ]
```

Subsample to 10% of your data:

```
iris[sample.int(nrow(iris), size = nrow(iris) * 0.1), ]
```

Sample rows with replacement (e.g. Bootstrap):

```
iris[sample.int(nrow(iris), replace = TRUE), ]
```

Reorder a given column of a table:

```
i2 <- iris
i2$Sepal.Width <- i2$Sepal.Width[sample.int(nrow(i2))]
```

RNGs and Reproducibility

Different RNGs in R

- Select different RNG algorithm with `RNGkind()`, or with additional arguments of `set.seed()`: `set.seed(n, kind = ...)`
- Particular properties go beyond this course (read `?Random` if interested)
- Defaults are often fine.
- Important for us:
    1. Default rng algorithms sometimes change between R versions.
        - Use `RNGversion("<R version>")` to set the PRNG `kind`, `normal.kind`, `sample.kind` to the default settings of R of version `<R version>` (e.g. `"3.6.0"`).
        - Use this if you want **reproducibility with an old R version**, but note that old R versions sometimes used sub-optimal PRNG kinds.
    2. `"L'Ecuyer-CMRG"` kind is interesting since it can be used when doing parallelization (we will see this in a later session)

PRNGs, Entropy, Security

- **If I know your seed and your code, I know your random numbers!**

E.g.:

Around 1999, there was some online poker software that shuffled cards with an algorithm that
(1) was public and
(2) used a seed based on its system clock

- If you knew the server time approximately, and you knew a few cards, you could guess the rest of the deck

PRNGs, Entropy, Security

- If I know your seed and your code, I know your random numbers!
- **Depending on the RNG, if I know enough random values, I can deduce the RNG state and predict the following random numbers**

- E.g. If you give me three `runif()` random values, and they are

  `0.26550866 0.37212390 0.57285336`

  The next value is quite likely `0.90820779` (since I assume you used seed 1).

- E.g. "Sploosh Kaboom Probability Calculator" [link] [video (worth it!)]

PRNGs, Entropy, Security

- If I know your seed and your code, I know your random numbers!
- Depending on the RNG, if I know enough random values, I can deduce the RNG state and predict the following random numbers

➔ Be careful if you generate random numbers that you don't want someone else to guess

➔ Guessing your PRNG-random number is only as difficult as guessing your seed

➔ so make sure the seed is *actually* random (i.e. has "high entropy") and use *cryptographically secure* PRNGs

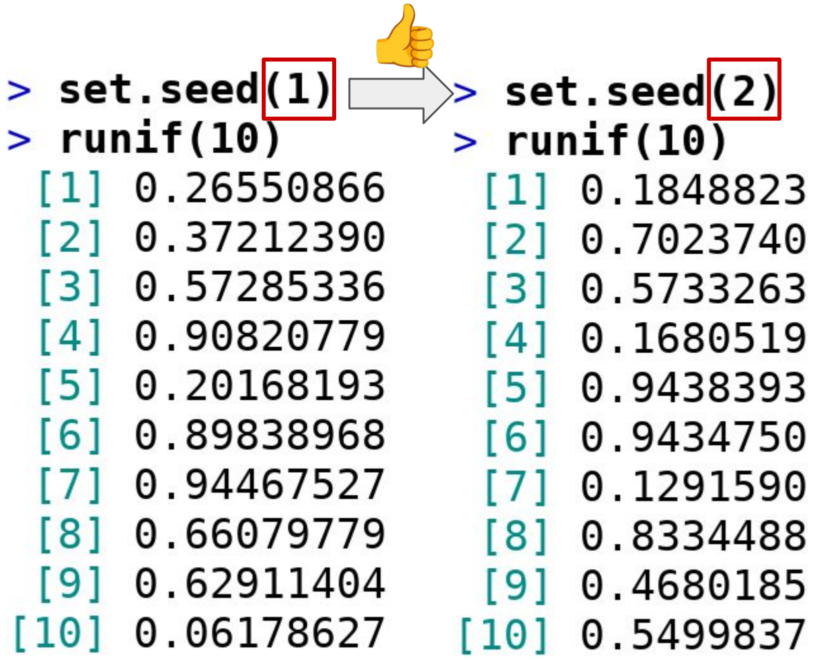
➔ As always in computer security: you better *really* know what you are doing if someone's money depends on it.

Setting a Seed

- If no seed is given, R generates one from system time and process ID
  - ➔ This is not really reproducible!
  - ➔ Always set a seed for "production" code
  - ➔ (And don't shuffle your poker decks with this)
- Seed is rounded down to an integer value, so `set.seed(1)` and `set.seed(1.8)` have the same effect.
- Seed should be between $(1-2^{31})$ and $(2^{31}-1)$ (i.e. about $\pm 2*10^{9}$)
- When doing randomised experiments, use different seeds for different instantiations, otherwise your runs are not really independent

Setting a Seed

Different seeds -- different values?

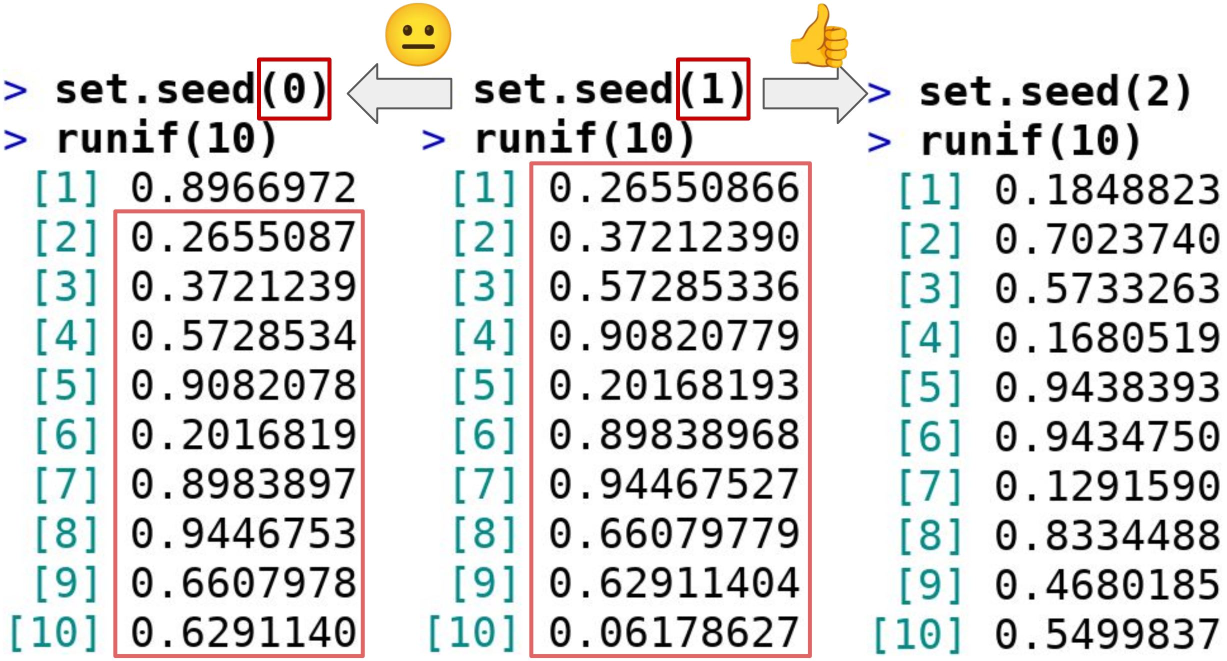


```
> set.seed(1)
> runif(10)
 [1] 0.26550866
 [2] 0.37212390
 [3] 0.57285336
 [4] 0.90820779
 [5] 0.20168193
 [6] 0.89838968
 [7] 0.94467527
 [8] 0.66079779
 [9] 0.62911404
[10] 0.06178627
```

```
> set.seed(2)
> runif(10)
 [1] 0.1848823
 [2] 0.7023740
 [3] 0.5733263
 [4] 0.1680519
 [5] 0.9438393
 [6] 0.9434750
 [7] 0.1291590
 [8] 0.8334488
 [9] 0.4680185
[10] 0.5499837
```

Setting a Seed

Different seeds -- different values?



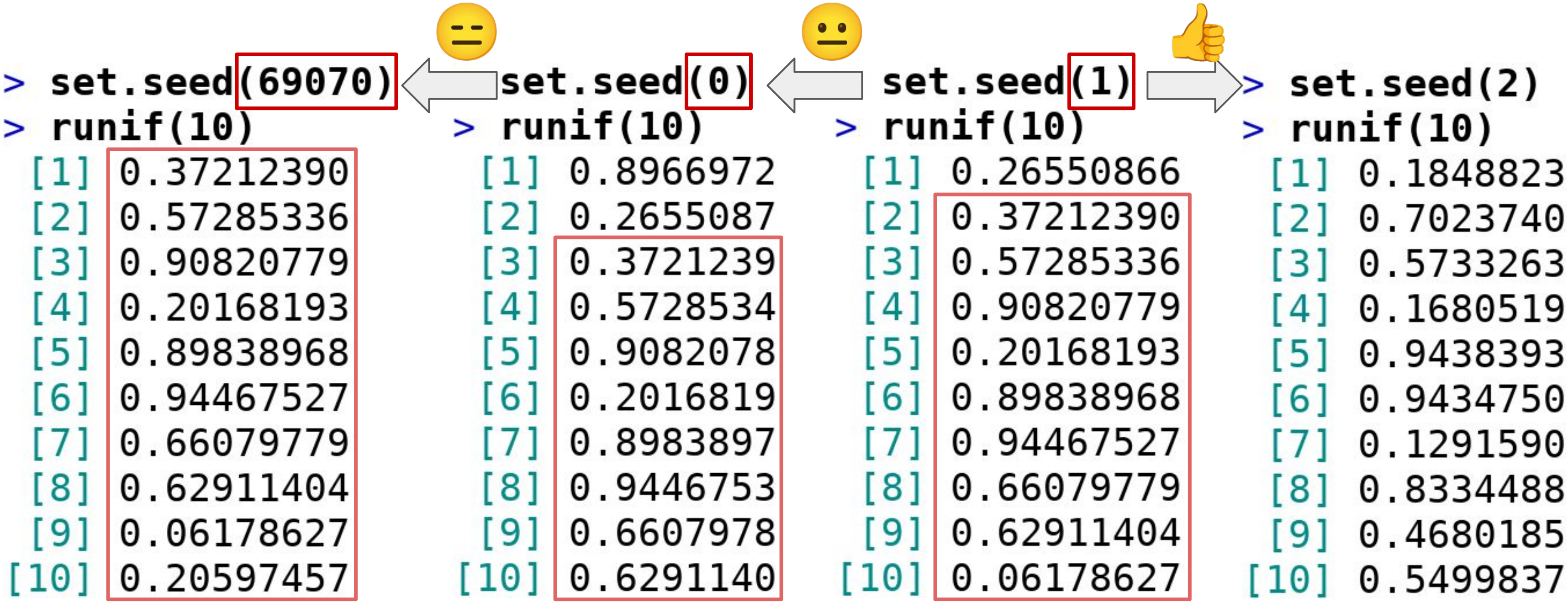
```
> set.seed(0)
> runif(10)
 [1] 0.8966972
 [2] 0.2655087
 [3] 0.3721239
 [4] 0.5728534
 [5] 0.9082078
 [6] 0.2016819
 [7] 0.8983897
 [8] 0.9446753
 [9] 0.6607978
[10] 0.6291140
```

```
> set.seed(1)
> runif(10)
 [1] 0.26550866
 [2] 0.37212390
 [3] 0.57285336
 [4] 0.90820779
 [5] 0.20168193
 [6] 0.89838968
 [7] 0.94467527
 [8] 0.66079779
 [9] 0.62911404
[10] 0.06178627
```

```
> set.seed(2)
> runif(10)
 [1] 0.1848823
 [2] 0.7023740
 [3] 0.5733263
 [4] 0.1680519
 [5] 0.9438393
 [6] 0.9434750
 [7] 0.1291590
 [8] 0.8334488
 [9] 0.4680185
[10] 0.5499837
```

Setting a Seed

Different seeds -- different values?



```
> set.seed(69070)
> runif(10)
 [1] 0.37212390
 [2] 0.57285336
 [3] 0.90820779
 [4] 0.20168193
 [5] 0.89838968
 [6] 0.94467527
 [7] 0.66079779
 [8] 0.62911404
 [9] 0.06178627
[10] 0.20597457
```

```
set.seed(0)
> runif(10)
 [1] 0.8966972
 [2] 0.2655087
 [3] 0.3721239
 [4] 0.5728534
 [5] 0.9082078
 [6] 0.2016819
 [7] 0.8983897
 [8] 0.9446753
 [9] 0.6607978
[10] 0.6291140
```

```
set.seed(1)
> runif(10)
 [1] 0.26550866
 [2] 0.37212390
 [3] 0.57285336
 [4] 0.90820779
 [5] 0.20168193
 [6] 0.89838968
 [7] 0.94467527
 [8] 0.66079779
 [9] 0.62911404
[10] 0.06178627
```

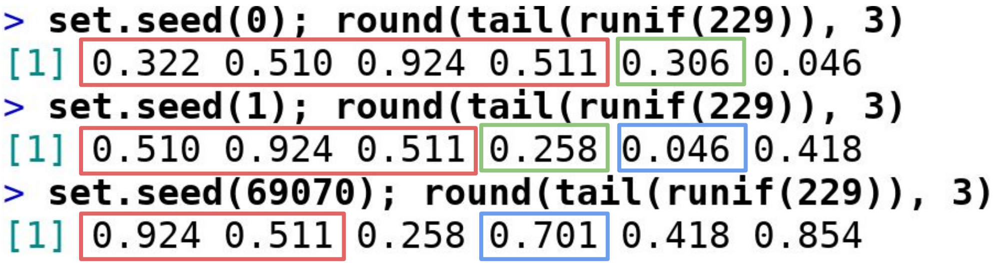
```
> set.seed(2)
> runif(10)
 [1] 0.1848823
 [2] 0.7023740
 [3] 0.5733263
 [4] 0.1680519
 [5] 0.9438393
 [6] 0.9434750
 [7] 0.1291590
 [8] 0.8334488
 [9] 0.4680185
[10] 0.5499837
```

Setting a Seed

Different seeds -- different values?

- Usually yes
- 0 and 69070 are "special" (see here)
  - (The problem is: For the default RNG, `.Random.seed` is initialized with a much simpler RNG)
  - Another example: `{set.seed(9423) ; runif(2)}` and `{set.seed(27884) ; tail(runif(9), 2)}`
- Using seeds 1, 2, 3, 4, 5, ... is mostly fine if you don't use too many (<10'000s)
- Use the `"L'Ecuyer-CMRG"` RNG with `parallel::nextRNGStream()` if you need many independent RNG streams

Also, the streams diverge eventually!



```
> set.seed(0); round(tail(runif(229)), 3)
[1] 0.322 0.510 0.924 0.511 0.306 0.046
> set.seed(1); round(tail(runif(229)), 3)
[1] 0.510 0.924 0.511 0.258 0.046 0.418
> set.seed(69070); round(tail(runif(229)), 3)
[1] 0.924 0.511 0.258 0.701 0.418 0.854
```

RNGs, Seed, and Reproducibility

You want the same code to produce the exact same result.

- Set the same seed
- Draw random numbers in the same order
  - This breaks if you add / remove calls to `runif()` or similar between runs!
  - This breaks if you step through code manually and run `runif()` yourself sometimes!
  - This breaks if you have parts of your code that is run "optionally", e.g. may or may not use cached values!

    Solutions:
    - `global.seed <- 123` (or some other number) in the beginning of your script,
      `set.seed(global.seed <- global.seed + 1)` after each optional code block, if you don't have too many
    - Better: `set.seed(123, kind = "L'Ecuyer-CMRG")`
      Then `next.seed <- parallel::nextRNGStream(.Random.seed)` before an "optional" section that may e.g. be cached
      Then `set.seed(0) ; .Random.seed <- next.seed` afterwards
    - Further Reading

RNGs, Seed, and Reproducibility

You want the same code to produce the exact same result.

- Set the same seed
- Draw random numbers in the same order
    - This breaks if you add / remove calls to `runif()` or similar between runs!
    - This breaks if you step through code manually and run `runif()` yourself sometimes!
    - This breaks if you have parts of your code that is run "optionally", e.g. may or may not use cached values!

      Best solution: **Cache the RNG State**!
        - Save RNG-state: `s <- .Random.seed`
        - Restore RNG-state: `.Random.seed <- s`
        - Advanced info:
            - Some RNG kind have "extra" RNG state not in `.Random.seed`. In this case:
            - Save: `s <- .Random.seed ; set.seed(1) ; .Random.seed <- s`
            - Restore: `set.seed(1) ; .Random.seed <- s`

RNGs, Seed, and Reproducibility

You want the same code to produce the exact same result.

- Set the same seed
- Draw random numbers in the same order
  - This breaks if you add / remove calls to `runif()` or similar between runs!
  - This breaks if you step through code manually and run `runif()` yourself sometimes!
  - This breaks if you have parts of your code that is run "optionally", e.g. may or may not use cached values!

    Best solution: **Cache the RNG State**!
    - Save RNG-state: `s <- .Random.seed`
    - Restore RNG-state: `.Random.seed <- s`
    - From inside functions: `.Random.seed <<- s`, since it is a *global* variable!

RNGs, Seed, and Reproducibility

You want the same code to produce the exact same result.

- Set the same seed
- Draw random numbers in the same order
  - This breaks if you add / remove calls to `runif()` or similar between runs!
  - This breaks if you step through code manually and run `runif()` yourself sometimes!
  - This breaks if you have parts of your code that is run "optionally", e.g. may or may not use cached values!
  - This can break when package versions change & packages change their behaviour

RNGs, Seed, and Reproducibility

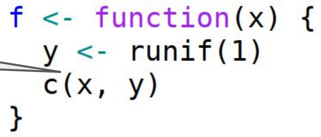
You want the same code to produce the exact same result.

- Set the same seed
- Draw random numbers in the same order
  - This breaks if you add / remove calls to `runif()` or similar between runs!
  - This breaks if you step through code manually and run `runif()` yourself sometimes!
  - This breaks if you have parts of your code that is run "optionally", e.g. may or may not use cached values!
  - This can break when package versions change & packages change their behaviour
- Be extra careful when doing parallelization

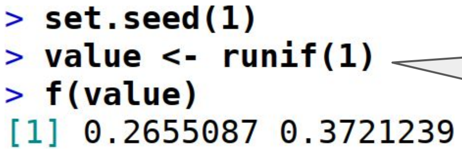
Random Numbers: Surprises

The order in which random functions are called is really important. Because function arguments are only evaluated (or "forced") when they are first referenced, this can lead to surprises:

'x' is only forced (i.e. evaluated) *after* the call to 'runif()'

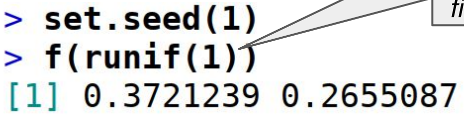


```
f <- function(x) {
  y <- runif(1)
  c(x, y)
}
```



```
> set.seed(1)
> value <- runif(1)
> f(value)
[1] 0.2655087 0.3721239
```

this runif(1) is evaluated *before* the function call --> 'y <- runif(1)' was the *second* call to runif



```
> set.seed(1)
> f(runif(1))
[1] 0.3721239 0.2655087
```

this runif(1) is evaluated *only when 'x' is evaluated inside the function* --> 'y <- runif(1)' was the *first* call to runif

➔ Results are different, even though these two code snippets look like they should be the same.

➔ It is safest to assume that *any* changes to your code change results of *all* random function calls

Random Numbers: Seed and Reproducibility II

- Sometimes other sources of randomness appear. Make sure you also set seeds for these
    - E.g. when calling an external program that uses randomness. These often have the option of setting a seed.
    - When calling python code that creates randomness, e.g. using the `reticulate` package, make sure to set the relevant seeds
        - Some old python versions may have randomized behaviour as a security precaution, even if they don't do something explicitly random
    - Sometimes there are bugs in packages that make them ignore seeds, unfortunately. It is always good to check if running the script twice gives the same result (... and a true pain to debug things if this is not the case).

Random Numbers: Seed and Replications

- Sometimes you need to repeat an experiment (involving randomness) many times, so that you can do statistics with the results (estimate expected value, standard error, etc.)
- If they are in different R sessions, you need to set a different seed for all these runs.
- Using just increasing seeds > 0 is usually fine because they produce wildly different random values
- However: Try to avoid setting the *same seed* for *different code*
  - Because it could create some correlation in your results that is only due to the correlated PRNG-results
  - E.g. if your function sets a seed to `initseed` at one point, and to `initseed + 1` at another, don't call this function with values for `initseed` that differ by only 1

What We Expect You to Know

- Random value generation: `runif()`, `rexp()`, `rbinom()`, `rgeom()`, ...
- Sample and shuffle integers using `sample.int()`
- Use `sample()` on vectors if you are *sure* they are not numerics
- Use `set.seed()` to initialize the PRNG and get reproducible results
- Use `RNGversion()` to make sure the same PRNG is used in different R versions
- Get the PRNG-state from `.Random.seed`; you can save it, and then restore it by assigning a saved value to `.Random.seed`.
  - With some random functions, you have to call `set.seed(dummyvalue)` before to reset them.
  - `.Random.seed` is *not* the same value as given to `set.seed()`.
- Changing your code (or even the version of a package) can have an influence on PRNG call order (and therefore your results) in surprising ways; consider it possible that all values change once something in the code in between has changed.
- Using cached values or conditionally skipping parts of your code will lead to differing PRNG states; cache the `.Random.seed`, or call `set.seed()` again after optional chunks, to avoid the problem
- Do not run different parts of your code with the same seed.