

Cryptographically Secure Pseudorandom Number Generator

Osvaldas Stanelis - Data and Information Systems - Nigel Davies

Introduction:
Random numbers are essential in cryptography, statistics, and simulations — but how do we know if computer-generated numbers are truly random? This project explores and compares two Python-based random number generators, random and secrets, and evaluates their output using formal statistical tests. By combining coding with data analysis, the project aims to understand how secure and reliable these generators are for real-world use.



- Objectives:
- Generate random integers using Python’s random and secrets modules
 - Evaluate randomness using Runs and Chi-Squared statistical tests
 - Compare output quality and determine suitability for secure applications

Methodology:
I generated 100 random integers from both the random and secrets modules, within the range of 1 to 500. To evaluate their randomness, I applied two statistical tests:

- Runs Test - checked if numbers appear in a random order based on the mean.
- Chi-Squared Test - compared how often each number appeared.

I used Python tools including numpy, scipy.stats, and statsmodels to perform the analysis in a reproducible and consistent way.

Results:
I tested the output of both modules using two statistical tests (picture below). In both cases, the p-values are above 0.05, meaning we fail to reject the null hypothesis - the numbers appear random and uniformly distributed. The secrets module showed slightly more consistent performance, which aligns with its purpose as a cryptographically secure generator.

```
Generating and testing random data from two sources:

=== Random Module ===
Generated Random Numbers:
[302, 50, 65, 54, 169, 493, 414, 377, 397, 132, 400, 207, 252, 459, 162, 328, 99, 328, 358, 471, 173, 478, 335, 22, 6, 219, 289, 233, 312, 246, 472, 187, 40, 111, 482, 237, 45, 226, 210, 283, 50, 386, 238, 172, 71, 289, 37, 17, 428, 278, 477, 446, 283, 331, 142, 297, 494, 164, 200, 46, 206, 493, 125, 244, 215, 142, 401, 85, 438, 157, 45, 277, 172, 434, 328, 192, 265, 71, 466, 159, 420, 440, 400, 490, 314, 375, 332, 291, 425, 27, 500, 384, 158, 128, 89, 17, 335, 186, 103, 129]

Number of samples: 100, Range: [1, 500]
Runs Test -> Z-statistic: 0.2175, p-value: 0.8279
=> Sequence appears random (fail to reject H0).
Chi-Squared Test -> Chi-stat: 540.0000, p-value: 0.0994
=> Distribution appears uniform (fail to reject H0).

=== Secrets Module ===
Generated Secure Random Numbers:
[373, 443, 3, 118, 234, 24, 29, 334, 315, 109, 26, 500, 94, 156, 500, 477, 261, 251, 48, 128, 421, 401, 115, 165, 219, 347, 206, 307, 310, 127, 168, 189, 459, 253, 495, 106, 448, 272, 116, 118, 477, 281, 7, 66, 322, 208, 33, 176, 112, 171, 463, 247, 355, 232, 91, 125, 217, 436, 328, 291, 326, 247, 425, 24, 0, 434, 413, 231, 484, 431, 195, 447, 262, 205, 135, 500, 413, 217, 382, 360, 466, 114, 474, 178, 86, 406, 305, 3, 267, 186, 439, 194, 442, 468, 40, 73, 327, 311, 350, 237, 464]

Number of samples: 100, Range: [1, 500]
Runs Test -> Z-statistic: 0.8433, p-value: 0.3990
=> Sequence appears random (fail to reject H0).
Chi-Squared Test -> Chi-stat: 490.0000, p-value: 0.6047
=> Distribution appears uniform (fail to reject H0).
```

Interpretation:
The statistical tests showed that both the random and secrets modules produced numbers that were random in sequence and uniformly distributed - with p-values above 0.05 in both tests. This indicates that neither sequence showed signs of predictable patterns or skewed distribution. However, the secrets module was slightly more consistent, which supports its intended use in cryptographic contexts where security is essential. These findings are consistent with existing research and highlight the importance of choosing the right generator based on application.

Challenges & Learning:
During the project, one of the main challenges was learning how to interpret statistical outputs like p-values and what they imply about randomness. I also worked on writing clean, reusable Python code, which helped me improve my coding structure and clarity. Finally, by using tools like statsmodels and scipy.stats, I gained practical experience in applying real statistical tests through programming — something I had not done before.

Future Work:
If time allowed, I would expand the dataset to at least 10,000 or 100,000 values to obtain stronger statistical confidence. I would also include additional randomness tests such as the Kolmogorov-Smirnov test and autocorrelation analysis to evaluate deeper structure. Adding visual tools like histograms, QQ-plots, and autocorrelation plots would provide more intuitive insights. Finally, comparing Python’s output with similar generators in R, Java, or C++ would help explore cross-platform differences and performance trade-offs.

Conclusion:
This project successfully evaluated two different approaches to random number generation in Python. Statistical testing confirmed that both methods produced random and well-distributed outputs, with the secrets module showing slightly better consistency. The work allowed me to combine my interest in mathematics and coding while gaining hands-on experience with statistical testing and data validation. It strengthened my understanding of how randomness is applied and measured in real-world systems, especially in secure computing.

```
### Random integers generator ###
def generate_random_integers_random_module(n, lower_bound, upper_bound):
    """
    Generates `n` random integers using the `random` module.
    Parameters:
        n (int): Number of random integers to generate.
        lower_bound (int): The smallest possible integer in the range.
        upper_bound (int): The largest possible integer in the range.
    Returns:
        A list of random integers.
    """
    random_integers = [] # Initializes an empty list to store generated integers
    for _ in range(100): # Iterates n times to generate n random integers
        rand_int = random.randint(lower_bound, upper_bound) # Uses random.randint
        random_integers.append(rand_int) # Appends the generated integer to the list
    return random_integers # Returns the list of random integers
```

```
### Run test ###
def runs_test(data):
    """
    Performs a Runs test to check the randomness of a sequence. It converts
    each value to 1 if >= mean, otherwise 0, then uses runstest_1samp.
    Parameters:
        data (list): Sequence of integer values to test.
    Returns:
        (float, float): A tuple containing the Z-statistic and the p-value.
    """
    mean_val = np.mean(data) # Compute mean of the data
    binary_seq = [1 if x >= mean_val else 0 for x in data] # Convert each value to binary
    z, p_value = runstest_1samp(binary_seq) # Perform the Runs test on the binary sequence
    return z, p_value # Return the test statistic and p-value
```

```
### Cryptographically secure random generator ###
def generate_random_integers_secrets(n, lower_bound, upper_bound):
    """
    Generates `n` cryptographically secure random integers using the `secrets` module.
    Parameters:
        n (int): Number of random integers to generate.
        lower_bound (int): The smallest possible integer in the range.
        upper_bound (int): The largest possible integer in the range.
    Returns:
        list: A list of cryptographically secure random integers.
    """
    random_integers = [] # Initializes an empty list to store cryptographically secure integers
    for _ in range(100): # Iterates n times to generate n cryptographically secure integers
        rand_int = lower_bound + secrets.randbelow(upper_bound - lower_bound)
        # secrets.randbelow(x) returns an integer in [0, x), so we offset by lower_bound
        random_integers.append(rand_int) # Appends the cryptographically secure integer to the list
    return random_integers # Returns the list of cryptographically secure integers
```

```
### Chi-square test ###
def chi_square_test(data, low, high):
    """
    Performs a Chi-squared test to check whether the distribution
    of integers is uniform over [low, high].
    Parameters:
        data (list): List of integers to test.
        low (int): Minimum integer in the range (inclusive).
        high (int): Maximum integer in the range (inclusive).
    Returns:
        (float, float): A tuple containing the Chi-squared statistic and the p-value.
    """
    bins = high - low + 1 # Number of distinct values in [low, high]
    freq = [0] * bins # Initialize frequency array for each possible integer value
    for val in data: # Count how many times each integer appears
        freq[val - low] += 1
    chi_stat, p_value = chisquare(freq) # Perform the Chi-squared test
    return chi_stat, p_value # Return the chi-squared statistic and p-value
```

References:

1. D. E. Knuth, The Art of Computer Programming, vol. 2, 1997.
2. M. Matsumoto and T. Nishimura, “Mersenne Twister,” ACM, 1998.
3. NIST SP 800-22, Apr. 2010.
4. A. Wald, J. Wolfowitz, “Runs Test,” 1940.
5. K. Pearson, “Chi-Squared Test,” 1900.
6. M. Luby, Pseudorandomness and Cryptographic Applications, 1996.
7. P. L’Ecuyer, “Random number generation,” in Handbook of Computational Statistics, Springer, 2012.
8. Python Software Foundation, Python 3 Documentation: secrets module, [Online]. Available: <https://docs.python.org/3/library/secrets.html>