International Journal of Science and Engineering Applications
Volume 13-Issue 12, 01 – 04, 2024, ISSN:- 2319 - 7560
DOI: 10.7753/IJSEA1312.1001

Analysis of Pseudorandom Number Generator in Python

Anton Novikau
Head of Mobile Development
Talaera Inc
Talaera, New York, USA

Abstract: This study evaluates the quality of the pseudorandom number generator (PRNG) implemented in Python's random module, which utilizes the Mersenne Twister algorithm. PRNGs are integral to numerous computational applications, and their statistical integrity directly impacts simulations, modeling, and cryptography. Using the ent toolset, we analyzed ten independent runs of the Python PRNG based on metrics including entropy, compression, chi-square tests, arithmetic mean, Monte Carlo π estimation, and serial correlation. Results indicate near-maximum entropy, a uniform byte distribution, accurate π estimation, and negligible serial correlation, demonstrating robust randomness properties suitable for general-purpose use. However, the deterministic nature of the Mersenne Twister limits its application in cryptographic contexts. These findings affirm the statistical reliability of Python's random module while highlighting the need for specialized algorithms for security-critical applications.

Keywords: python, Entropy analysis, Chi-square test, Serial correlation, Randomness evaluation

1. INTRODUCTION

Pseudorandom number generators (PRNGs) are mathematical algorithms designed to produce sequences of numbers that mimic the properties of randomness. They are distinct from true random number generators (TRNGs), which rely on inherently unpredictable physical phenomena, such as radioactive decay or thermal noise[2]. Instead, PRNGs operate deterministically, generating numbers from an initial input known as a seed[8]. This approach allows the sequence to be both predictable and reproducible—characteristics that are essential in many computational contexts[1].

The central problem addressed by PRNGs is the need for randomness in scenarios where obtaining true random numbers is either impractical or unnecessary. Randomness is a cornerstone of numerous applications, including statistical simulations, cryptography, gaming, and randomized algorithms. However, acquiring truly random numbers can be slow, expensive, or infeasible for the high-speed, high-volume demands of modern computation. PRNGs provide an efficient and scalable solution by producing numbers that, while not genuinely random, are statistically indistinguishable from random sequences for most practical purposes.

One of the strengths of PRNGs lies in their ability to generate large amounts of pseudorandomness quickly. The deterministic nature of these generators also enables reproducibility—a crucial feature for debugging and verifying results in scientific experiments and software development [3]. For example, a simulation using a PRNG can be repeated precisely by reinitializing the generator with the same seed, ensuring consistent outcomes. This is a key advantage over non-deterministic methods, where exact replication is often impossible.

Despite their utility, PRNGs must meet rigorous standards to be effective. High-quality PRNGs exhibit long periods before repeating sequences, uniform distribution, and minimal correlation between successive numbers. These properties are essential for maintaining the statistical integrity required in simulations, modeling, and cryptographic systems. Yet, no PRNG is perfect. The trade-off between computational efficiency and the complexity needed to achieve near-random behavior remains a fundamental challenge.

A poorly implemented PRNG may produce sequences that are predictable, especially if the internal state of the generator or its seed can be inferred [4]. This predictability can compromise systems that rely on randomness for security, such as cryptographic protocols. For instance, if an attacker can determine or guess the seed, they may be able to recreate the sequence of numbers and exploit the system.

A bad PRNG may fail tests for randomness, producing sequences that are unevenly distributed or exhibit discernible patterns. Such flaws can have cascading effects in domains like gaming, randomized algorithms, and scientific modeling. In gaming, for example, biased distributions could favor certain outcomes, leading to unfair advantages. In simulations, non-random sequences could fail to accurately represent the modeled system.

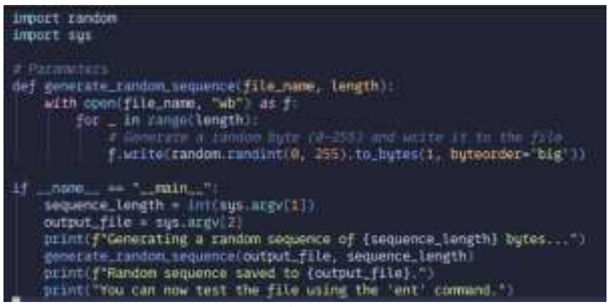
The random module in Python uses the Mersenne Twister algorithm as its core pseudorandom number generator (PRNG) [6]. This algorithm is well-known for its high speed and excellent statistical properties, making it a widely used PRNG in general-purpose applications. The Mersenne Twister has a period of $2^{19937} - 1$, meaning the sequence of numbers it generates will not repeat for a very long time, ensuring high-quality randomness for non-secure applications. Given the same seed, the random module will always produce the same sequence of numbers. This makes it suitable for reproducibility in simulations and debugging. The Mersenne Twister is not suitable for cryptographic purposes because its deterministic nature and internal state can be inferred if enough output is observed [7]. In this study, we aim to find out the quality of the generated random number sequences using the Python random module.

2. METHODS

Testing the quality of a pseudorandom number generator (PRNG) requires a combination of statistical methods to ensure it produces sequences that are sufficiently random and free of discernible patterns [5]. One effective approach involves using tools like the ent toolset, which provides a suite of tests tailored for evaluating randomness. The chi-square test is a foundational method included in such tools, comparing the observed frequencies of generated values to their expected frequencies under a uniform distribution. This test highlights biases or uneven distributions in the PRNG's output.

Additionally, the Monte Carlo method, which involves simulating probabilistic systems (e.g., estimating π through random points), can be used to verify whether the PRNG produces results consistent with known theoretical probabilities. Another critical measure is the serial correlation coefficient, which assesses the relationship between successive numbers in the sequence. Ideally, there should be no significant correlation, indicating that the numbers are independent of one another. Together, these methods, encapsulated in the ent suite, offer a robust way to evaluate both the statistical quality and practical reliability of a PRNG.

To analyze Python's PRNG quality, following program was developed:



```
import random
import sys

# Parameters
def generate_random_sequence(file_name, length):
    with open(file_name, "wb") as f:
        for _ in range(length):
            # Generate a random byte (0-255) and write it to the file
            f.write(random.randint(0, 255).to_bytes(1, byteorder='big'))

if __name__ == "__main__":
    sequence_length = int(sys.argv[1])
    output_file = sys.argv[2]
    print(f"Generating a random sequence of {sequence_length} bytes...")
    generate_random_sequence(output_file, sequence_length)
    print(f"Random sequence saved to {output_file}.")
    print("You can now test the file using the 'ent' command.")
```

Figure. 1 Source code of random sequence generator

This code generates a set of random sequences using Python's random module and saves it in the file. Results are stored in the output file, and then each generated sequence is analyzed using ent utility to find out its properties. The length of generated random sequences is 10,000,000. For each run seed was randomized.

3. RESULTS

Run	Entropy	Compression (%)	Chi Square Samples	Chi Square Value
1	7.999980	0	10000000	282.19
2	7.999984	0	10000000	221.10
3	7.999983	0	10000000	229.40
4	7.999981	0	10000000	256.98
5	7.999981	0	10000000	265.14
6	7.999983	0	10000000	231.63
7	7.999984	0	10000000	226.14
8	7.999980	0	10000000	273.37
9	7.999981	0	10000000	257.42
10	7.999983	0	10000000	242.69

Table. 1 Part 1 of analysis results

Run	Chi Square Exceed Percent (%)	Arithmetic Mean	Monte Carlo Pi Value	Monte Carlo Error (%)	Serial Correlation
1	11.65	127.4710	3.141145256	0.01	-0.000349
2	93.86	127.5141	3.139926056	0.05	-0.000429
3	87.38	127.4942	3.141128456	0.01	0.000452
4	45.34	127.4721	3.141222056	0.01	0.000153
5	31.82	127.5094	3.142904457	0.04	-0.000948
6	85.05	127.5112	3.141090056	0.02	-0.000219
7	90.31	127.4466	3.142755657	0.04	0.000864
8	20.50	127.4651	3.143214057	0.05	-0.000025
9	44.58	127.4845	3.138368455	0.10	0.000379
10	70.01	127.4991	3.140518856	0.03	0.000214

Table. 2 Part 2 of analysis results

The analysis of the Python random module's pseudo-random number generator (PRNG) was conducted using the ent tool across ten independent runs. The following metrics were assessed to evaluate the quality of randomness in the generated data:

1. **Entropy**: The entropy values consistently averaged **7.99998 bits per byte**, with negligible variation across runs. This value approaches the theoretical maximum entropy of **8 bits per byte** for 8-bit data,

indicating nearly perfect uniformity in byte distribution.

2. **Compression Percentage**: All runs resulted in a compression percentage of **0%**, demonstrating a lack of detectable patterns or redundancy in the data.
3. **Chi-Square Test**:
    - Chi-square values ranged from **221.10** to **282.19**, with corresponding exceed percentages (p-values) between **11.65%** and **93.86%**.
    - The majority of p-values fell within the generally accepted range for random data (**10%–90%**), indicating no significant deviations from a uniform distribution of byte values.
4. **Arithmetic Mean**: The observed arithmetic means ranged from **127.4466** to **127.5141**, closely aligning with the expected mean of **127.5** for a uniform distribution of byte values between 0 and 255.
5. **Monte Carlo π Estimation**: Estimates for π based on the Monte Carlo method ranged from **3.138368455** to **3.142904457**, with error percentages between **0.01%** and **0.10%**. These results demonstrate a high level of randomness, as the accuracy of π estimation depends on the quality of the underlying random data.
6. **Serial Correlation Coefficient**: Serial correlation coefficients ranged from **-0.000948** to **0.000864**, with values consistently near zero, indicating negligible correlation between successive bytes and strong independence.

4. DISCUSSION

The results demonstrate that the Python random module's PRNG exhibits robust randomness properties across multiple statistical dimensions:

1. **Entropy and Compression**: The near-maximum entropy values and 0% compression percentage indicate a uniform and pattern-free distribution of byte values, which is a hallmark of high-quality random sequences.
2. **Chi-Square Test**: The acceptable range of chi-square p-values suggests that the observed distribution closely matches the theoretical uniform distribution. This implies that the PRNG effectively randomizes byte values with minimal bias.
3. **Arithmetic Mean**: Observed means remain consistent with the theoretical expectation of 127.5, confirming the even distribution of generated byte values.
4. **Monte Carlo π Estimation**: Accurate π estimations with minimal error highlight the adequacy of the PRNG for applications requiring reliable randomness, as the Monte Carlo method is sensitive to imperfections in random input.
5. **Serial Correlation**: The near-zero serial correlation coefficients indicate a lack of dependency between successive values, affirming the independence of generated data.

Overall, these findings suggest that the Python random module's PRNG produces high-quality random sequences that meet rigorous statistical benchmarks. While this level of randomness is suitable for general-purpose applications and simulations, further investigation into cryptographic robustness may be required for security-sensitive use cases. The observed metrics align closely with theoretical expectations for a well-designed PRNG, validating its effectiveness in generating statistically random data.

5. REFERENCES

1. Antunes, B., & Hill, D. R. C. (2024, January 30). Reproducibility, energy efficiency and performance of pseudorandom number generators in machine learning: a comparative study of python, numpy, tensorflow, and pytorch implementations. arXiv.org. https://arxiv.org/abs/2401.17345
2. Zhang, Y., Zhu, M., Yang, B., & Liu, L. (2021). Study on post-processing algorithms for true random number generators [Conference: 2021 6th International Symposium on Computer and Information Processing Technology (ISCIPT)]. https://doi.org/10.1109/iscipt53667.2021.00067
3. Antunes, B., & Hill, D. R. C. (2024a, January 30). Reproducibility, energy efficiency and performance of pseudorandom number generators in machine learning: a comparative study of python, numpy, tensorflow, and pytorch implementations. arXiv.org. https://arxiv.org/abs/2401.17345
4. Kim, H., Kwon, Y., Sim, M., Lim, S., & Seo, H. (2021). Generative Adversarial Networks-Based Pseudo-Random number Generator for embedded processors. In Lecture notes in computer science (pp. 215–234). https://doi.org/10.1007/978-3-030-68890-5_12
5. L'Ecuyer, P., & Simard, R. (2007). TestU01. ACM Transactions on Mathematical Software, 33(4), 1–40. https://doi.org/10.1145/1268776.1268777
6. random — Generate pseudo-random numbers. (n.d.). Python Documentation. https://docs.python.org/3/library/random.html
7. Vigna, S. (2019, October 14). It is high time we let go of the Mersenne Twister. arXiv.org. https://arxiv.org/abs/1910.06437
8. Barker, E., Barker, W., Burr, W., Polk, W., Smid, M., Computer Security Division, & National Institute of Standards and Technology. (2012). Recommendation for Key Management - Part 1:

General (Revision 3). In NIST Special Publication 800-57. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-57p1r3.pdf

9.