ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 8| August 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

IMPLEMENTATION OF leap_year_verifier() USING QUAD-CENTURY CONDITIONAL LOGIC - A CASE STUDY

Cadet G.Kathiravan, Cadet K.Sujith, Cadet M.Magizharasan,
Class - XII 2025-26, Sainik School Amaravathinagar
Post: Amaravathinagar, Udumalpet Taluk, Tirupur Dt, Tamilnadu State.

ABSTRACT

Every calendar library, scheduler, and billing system eventually asks the same one-line question: does this year have 366 days or 365? The answer looks trivial until a program has to handle a century year, at which point a rule most people never think about – the exception to the exception – decides whether the check is right or quietly wrong.

This paper documents the design, implementation, and verification of leap_year_verifier(), a function that determines Gregorian leap years through three nested conditional statements rather than a single compact Boolean expression. We refer to the third and final check, the one governing century years exactly divisible by four hundred, as the quad-century branch, since it is the test that revives a leap day the centennial rule had just taken away.

The function is evaluated against equivalence partitioning, boundary value analysis, and McCabe's cyclomatic complexity metric, and it is validated empirically across one complete four-hundred-year Gregorian cycle, where it agrees with Python's own calendar.isleap() on every one of four hundred years and reproduces the historically documented count of ninety-seven leap years without a single exception.

KEYWORDS: *Leap Year Verifier (LYV), Quad-Century Rule (QCR), Gregorian Calendar (GC), Boundary Value Analysis (BVA), Equivalence Partitioning (EP), Cyclomatic Complexity (CC).*

1.INTRODUCTION

Julius Caesar's calendar reform in 46 BCE fixed the length of a year at 365.25 days, implemented by adding a single intercalary day every fourth year [1]. It was a good approximation for its time, but not a correct one: the true tropical year is closer to 365.2422 days, a difference of roughly eleven minutes that accumulates year after year. By the time Pope Gregory XIII moved to correct the problem in 1582, the Julian calendar had drifted ten full days out of step with the seasons, pushing the spring equinox away from its expected date near the 21st of March [3]. The fix, attributed to the astronomer Aloysius Lilius, kept Caesar's every-four-years rule largely intact but added a second layer: century years would no longer automatically qualify as leap years, except for the ones divisible by four hundred [1][2]. That second layer, small as it looks on paper, is the entire reason a leap year check cannot be written as "divisible by four" and be done with it.

This paper is not really about the calendar. It is about a single function, leap_year_verifier(), that encodes the rule above using nested conditional logic, and about what happens when that function is subjected to the ordinary tools of software testing: equivalence partitioning, boundary value analysis, and McCabe's cyclomatic complexity metric. We call the decisive third check – the one that restores a leap day to a century year exactly divisible by four hundred – the quad-century branch, since it is the clause that keeps the calendar synchronised on the scale of four-century cycles. Sections II and III set the historical and mathematical context; Sections IV to VII cover the design, algorithm, and code; Sections VIII to X analyse the function's complexity from three different angles; Sections XI and XII test and validate it; and Section XIII argues, with a documented case from the year 2000, that the quad-century branch is not a theoretical curiosity but the specific place where real leap-year code has actually failed.

2. RELATED WORK

The leap year check has a long history as a teaching exercise. Coding Dojo and Codurance both catalogue a "Leap Years Kata" used to introduce test-driven development to beginners, precisely because the domain logic is small enough to hold in one's head while the mechanics of red-green-refactor are still unfamiliar [5][6]. That same smallness is what makes the function a convenient subject for a complexity case study: it is short enough to analyse exhaustively, yet it already contains a genuine, historically motivated exception-to-an-exception, which most toy examples do not.

Production language runtimes converge on the same three-part rule even though their surface syntax differs. CPython's calendar module defines isleap() as a single return statement, year % 4 == 0 and (year % 100 != 0 or year % 400 == 0), using short-circuit Boolean evaluation to fold three comparisons into one expression [10]. Java's java.time.Year.isLeap(), by contrast, is documented and commonly implemented as sequential conditionals evaluated under the ISO-8601 proleptic Gregorian system, which applies today's rule uniformly across the whole timeline rather than switching to Julian dating before 1582 [11]. Both are correct; neither is written the way this paper writes leap_year_verifier(), which keeps each clause of the rule as its own if statement so that a coverage tool can point at each decision individually. Cyclomatic complexity, introduced by McCabe in 1976 as a graph-theoretic count of the linearly independent paths through a control-flow graph, is the standard vocabulary for describing exactly how "branchy" a function like this one is [7][8]. Boundary value analysis and equivalence partitioning remain

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 8| August 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

the standard black-box techniques for turning a rule built on thresholds into a concrete, minimal set of test inputs [9].

3.THE QUAD-CENTURY RULE: FROM DRIFT TO A THIRD CONDITIONAL

The arithmetic behind the 1582 reform is straightforward once it is laid out. Over four hundred years, the Julian rule adds a leap day in exactly one hundred of them – every year divisible by four, with no exceptions. The Gregorian reform keeps that pattern but disqualifies three of those hundred: the century years not divisible by four hundred. Of the four centesimal years in any four-hundred-year span, three lose their leap day and one keeps it, so 1600, 2000, and 2400 remain leap years while 1700, 1800, and 1900 do not [1][2][3]. That leaves ninety-seven leap years per four hundred, and a mean calendar year of 365 + 97/400 = 365.2425 days, within about half a minute of the true tropical year [1][3]. Cl. Tøndering's widely cited calendar reference puts the residual error at roughly one day of drift every 3,300 years – accurate enough that, in the more than four centuries since the reform, nobody has needed to act on it [4].

What matters for this paper is not the astronomy but the shape of the rule it produces: three clauses, each narrowing the previous one, where the third clause reverses the second rather than extending it. A year divisible by four is provisionally a leap year; a year additionally divisible by a hundred loses that status; a year additionally divisible by four hundred gets it back. We refer to this last, reversing clause throughout this paper as the quad-century check, since a century year only ever reaches it after already surviving the first test, and it is the one test that only ever fires on years that close out a full four-century block.

4. DESIGN OF leap_year_verifier()

leap_year_verifier() was written under one governing constraint: every clause of the rule in Section III should correspond to exactly one conditional statement, in the same order a person would apply the rule by hand. This is a deliberate departure from the compact idiom in Python's own standard library. CPython's isleap() short-circuits two of the three comparisons into a single Boolean expression joined by and and or [10]; it is correct and it is fast, but a branch-coverage tool sees it as one statement with an internal and/or structure rather than three separate decision points. For a case study whose whole purpose is to reason about branches, that compression works against the goal, so leap_year_verifier() instead nests three plain if statements, keeping the function's control-flow graph and the rule's three clauses the same object viewed two ways.

5. ALGORITHM: leap_year_verifier()

STEP 01: START
STEP 02: RECEIVE THE INTEGER YEAR AS INPUT.
STEP 03: CHECK WHETHER YEAR IS DIVISIBLE BY 4.
STEP 04: IF NOT DIVISIBLE BY 4, RETURN FALSE (COMMON YEAR) AND STOP.
STEP 05: IF DIVISIBLE BY 4, CHECK WHETHER YEAR IS DIVISIBLE BY 100.
STEP 06: IF NOT DIVISIBLE BY 100, RETURN TRUE (LEAP YEAR) AND STOP.
STEP 07: IF DIVISIBLE BY 100, CHECK WHETHER YEAR IS DIVISIBLE BY 400 – THE QUAD-CENTURY CHECK.
STEP 08: IF NOT DIVISIBLE BY 400, RETURN FALSE (COMMON YEAR) AND STOP.
STEP 09: IF DIVISIBLE BY 400, RETURN TRUE (LEAP YEAR) AND STOP.
STEP 10: OUTPUT THE RESULT.

1. **PROGRAM: leap_year_verifier()**

```
# leap_year_verifier() function definition
def leap_year_verifier(year):
    if year % 4 == 0:
        if year % 100 == 0:
            if year % 400 == 0:
                return True
            else:
                return False
        else:
            return True
    else:
        return False

# Calling function
if __name__ == "__main__":
    test_year = int(input("Enter a year: "))
    result = leap_year_verifier(test_year)
    print(f"{test_year} is a leap year: {result}")
```

6. FUNCTIONING OF THE CODE

The function accepts a single integer, year, and returns a Boolean. The first if, year % 4 == 0, eliminates three out of every four years immediately – 2019, for instance, fails here and the function returns False without touching either remaining check. A year that survives, such as 2020, is provisionally a leap year unless the second if, year % 100 == 0, is also true; 2020 fails this second test and returns True directly. Century years are the only ones that ever reach the third if. 1900 satisfies the first two tests and then fails the quad-century check, year % 400 == 0, so it returns False. 2000 satisfies all three and is the only kind of year that passes through every conditional in the function before returning True. Four inputs, four distinct routes through the code – and, as Section X will show, four is not a coincidence.

7. SPACE COMPLEXITY

leap_year_verifier() allocates nothing that scales with its input. The only data involved are the parameter year and, implicitly, the three intermediate Boolean comparisons the interpreter evaluates and discards on each call; none of them are stored in a structure whose size depends on the magnitude of year or on how many times the function has been called before. Space complexity for leap_year_verifier() is therefore O(1).

8.TIME COMPLEXITY

Each of the three conditionals evaluates a single modulo operation on a fixed-width integer, an operation whose cost on

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 8| August 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

real hardware does not depend on the value being divided, only on its bit width. Because the function performs at most three such operations regardless of which branch it takes, time complexity is O(1). This was measured rather than merely asserted: leap_year_verifier() was timed over batches of randomly generated four-digit years ranging from one thousand to one million calls, using Python's timeit module, with the results in Table I.

Calls (N)	Total Time (s)	Time/Call (ns)
1,000	0.000069	69.08
5,000	0.000250	50.05
10,000	0.000491	49.09
50,000	0.004382	87.65
100,000	0.009939	99.39
500,000	0.043971	87.94
1,000,000	0.052305	52.30

Table I. Measured execution time of leap_year_verifier() for increasing numbers of calls (random four-digit years, CPython, timeit module).

Total time scales with the number of calls made, not with the magnitude of the years being tested, and the per-call time stays within a narrow band of roughly forty-nine to ninety-nine nanoseconds across three orders of magnitude of call volume. The residual wobble is consistent with ordinary interpreter noise – garbage-collection pauses, CPython's small-integer caching, and system scheduling jitter – rather than any genuine dependence on input size, which is exactly what an O(1) claim predicts once measurement noise is accounted for.

9.CYCLOMATIC COMPLEXITY AND BRANCH COVERAGE

McCabe's metric counts the number of linearly independent paths through a program's control-flow graph, and for a structured function with no loops it reduces to a simple rule: cyclomatic complexity V(G) equals the number of decision points plus one [7][8]. leap_year_verifier() contains exactly three decision points – the three if statements traced in Section VII – so V(G) = 3 + 1 = 4. Those four paths correspond exactly to the function's four terminal return statements, which is worth pausing on: it is not a coincidence that a rule with three clauses, expressed as three nested ifs, produces four distinct outcomes, because n mutually exclusive binary decisions chained in sequence always terminate in n + 1 leaves. McCabe's own guidance treats V(G) as an upper bound on the number of test cases required to exercise every branch [7][8]; here that bound is tight, since achieving full branch coverage requires – and requires no more than – one test case per equivalence class, taken up next.

10.TEST CASE DESIGN: EQUIVALENCE PARTITIONING AND BOUNDARY VALUE ANALYSIS

Equivalence partitioning groups inputs that a program is expected to treat identically, while boundary value analysis targets the edges of those groups, on the well-supported assumption that defects cluster at thresholds rather than in the middle of a range [9]. leap_year_verifier()'s three-clause rule produces exactly four equivalence classes, listed in Table II together with the boundary years chosen to represent them.

Class	Description	Years	Exp.
A	Not divisible by 4	2019, 2021	False
B	Div. by 4, not 100	2020, 2024	True
C	Div. by 100, fails quad-century check	1700, 1800, 1900, 2100	False
D	Div. by 400, passes quad-century check	1600, 2000, 2400	True

Table II. The four equivalence classes produced by the Gregorian rule and their representative years.

Classes C and D are the ones that matter for this paper, since they are the only classes that exercise the quad-century branch at all, and they are also the rarest: only one year in a hundred is even eligible to reach the third if, and only one in four of those is eligible to pass it. Ten boundary years spanning all four classes – 2019, 2020, 1900, 2000, 2024, 2100, 2400, 1600, 1700, and 1800 – were run against leap_year_verifier() and checked against Python's own calendar.isleap() as an independent reference implementation. Every result matched exactly, as shown in Table III.

Year	leap_year_verifier()	calendar.isleap()
2019	False	False
2020	True	True
1900	False	False
2000	True	True
2024	True	True
2100	False	False
2400	True	True
1600	True	True
1700	False	False
1800	False	False

Table III. Boundary test results, leap_year_verifier() versus the CPython reference implementation.

11.EMPIRICAL VALIDATION ACROSS A FULL QUAD-CENTURY CYCLE

A ten-point boundary check is sufficient for branch coverage, but it does not, by itself, demonstrate that the function is correct across an entire Gregorian cycle. To test that more strongly, leap_year_verifier() was run against every one of the four hundred years from 1600 to 1999 inclusive – one complete four-century block – with each result compared against calendar.isleap(). The function returned True for exactly ninety-seven of the four hundred years, with zero disagreements against the reference implementation across the entire range. That figure is not merely internally consistent; it is also an external check against the historical record, since both the U.S. Naval Observatory and Tøndering's calendar reference independently report the same ninety-seven-leap-years-per-four-hundred-years figure that falls directly out of the 1582 reform's arithmetic [3][4]. Internal consistency, against calendar.isleap(), and external validity, against the documented historical count, agree completely.

12.DISCUSSION: WHY THE QUAD-CENTURY BRANCH IS THE ONE THAT BREAKS?

The equivalence classes in Section XI are not evenly likely to be exercised by accident. Roughly one year in four is

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 8| August 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

divisible by four; roughly one in a hundred of those is additionally a century year; and only one in four hundred survives the quad-century check as well. A programmer who tests a leap year function against whichever years happen to be convenient – the current year, a birthday, a handful of recent years – will, essentially always, land in classes A and B and never touch C or D at all. That is not a hypothetical failure mode. As the calendar rolled from 1999 into 2000, a number of systems reportedly mishandled 29 February 2000 for exactly this reason, applying the centennial exception without the quad-century exception to the exception. Documented cases from that period include disrupted lottery ticket sales in Arizona, a Queens taxi company that began charging an extra day's fare a day early, a UK hospital whose equipment briefly malfunctioned, and reports that the United States Federal Aviation Administration's own systems rejected 29 February 2000 as an invalid date [13][14]. Every one of these systems had, in effect, run for years on a leap year check that had never been fed a year divisible by four hundred, simply because 2000 was the first one to come along since those systems were built. The failure was not a coding mistake in the ordinary sense; it was a branch that existed in the rule but not, in practice, in anyone's test suite.

The rule has been proposed to go one clause further. The astronomer John Herschel, and later Camille Delambre, suggested disqualifying years divisible by four thousand from leap-year status, which would trim the mean year to about 365.24225 days and correct the roughly one-day drift the Gregorian rule still accumulates every 3,300 years [4][12]. Implemented literally, that proposal would add a fourth nested if to leap_year_verifier(), raising cyclomatic complexity from four to five and adding a fifth equivalence class that no year alive today will ever be able to test. No civil calendar has adopted the rule, and it is worth noting why: a correction whose only test case is the year 4000 is a correction no engineering process can realistically validate now.

13.LIMITATIONS

leap_year_verifier() implements the proleptic Gregorian rule uniformly across all years, consistent with the ISO-8601 convention used by java.time [11]; it does not model the actual historical transition from the Julian calendar, which took effect in different countries at different times between 1582 and the early twentieth century, nor does it validate that year is a plausible calendar year at all – negative years, non-integer input, and very large magnitudes are all accepted without complaint. Both omissions are deliberate: the function is scoped to the single decision this paper is about, and folding calendar-conversion or input-validation logic into it would blur the branch count the whole case study depends on.

14.CONCLUSION

leap_year_verifier() implements the Gregorian leap year rule as three nested conditionals rather than one compact expression, a choice made specifically so that the function's control-flow graph mirrors the rule's own three clauses. That choice pays off directly in the analysis: a cyclomatic complexity of four matches four natural equivalence classes with no slack in either direction, and the resulting test suite – eight boundary years plus a full four-hundred-year sweep – both passes and independently reproduces the historically documented count of ninety-seven leap years per cycle. The quad-century branch, the third and rarest of the three checks, is also the one the historical record shows programmers actually getting wrong, in the leap-year failures reported around the year 2000. The lesson of this case study is not really about leap years at all: a rule with a rare, reversing exception needs a test suite built deliberately around that exception, because ordinary or incidental test data will not find it on its own.

15.ACKNOWLEDGEMENT

We express our sincere gratitude to Almighty God for the strength to complete this research paper, and to our parents for their constant encouragement throughout its preparation.

We express our deep sense of gratitude to the Principal, Captain (Indian Navy) K. Manikandan, Sainik School Amaravathinagar, who has continuously motivated and extended his helping hand to us.

We express our sincere thanks to the Vice Principal, Lt. Col. Kaushok Nandini Arun, Sainik School Amaravathinagar, for constant encouragement and guidance provided during this research.

Our sincere thanks also to Mr. Praveen Kumar Murigeppa Jigajinni, Master In-charge, guide, and mentor, who critically reviewed this paper and helped us resolve the difficulties encountered while implementing and testing leap_year_verifier().

16.REFERENCES

1. *Wikipedia, "Gregorian calendar," https://en.wikipedia.org/wiki/Gregorian_calendar*
2. *Wikipedia, "Century leap year," https://en.wikipedia.org/wiki/Century_leap_year*
3. *U.S. Naval Observatory, "Leap Years," https://aa.usno.navy.mil/faq/leap_years*
4. *C. Tøndering, "The Gregorian Calendar," https://www.tondering.dk/claus/cal/gregorian.php*
5. *Coding Dojo, "Leap Years Kata," https://codingdojo.org/kata/LeapYears/*
6. *Codurance, "Leap Year Kata," https://www.codurance.com/katas/leap-year*
7. *T. J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.*
8. *Verifysoft, "McCabe Metrics," https://verifysoft.com/en_mccabe_metrics.html*
9. *Guru99, "Boundary Value Analysis and Equivalence Partitioning,"https://www.guru99.com/equivalence-partitioning-boundary-value-analysis.html*
10. *note.nkmk.me, "Determine, Count, and List Leap Years in Python," https://note.nkmk.me/en/python-calendar-leap-year/*
11. *Oracle, "Year (Java Platform SE 8)," https://docs.oracle.com/javase/8/docs/api/java/time/Year.html*
12. *J. Forster, "Solving The Perpetual Calendar Problem: A Modest Proposal," https://jackforster.substack.com/p/solving-the-perpetual-calendar-problem*
13. *SmarterMSP, "The Leap Year Bug, Y2K's Lesser Known Counterpart," https://smartermsp.com/tech-time-warp-the-leap-year-bug-y2ks-lesser-known-counterpart/*
14. *WordSpy, "Y2K leap year bug," https://wordspy.com/words/y2k-leap-year-bug/*