

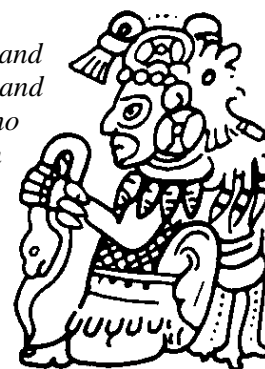
# *Python Vuh: Mayan Calendrical Mathematics with Python*

Ivan Van Laningham (ivanlan@callware.com)  
Callware Technologies, Inc.

---

*In the beginning, there was nothing but sky. Sovereign Plumed Serpent, God L, and other Creator Gods were sitting around in the ocean-sky, smoking their cigars and drinking pulque. Whatever might yet be was simply not there: no animals, no people, no rocks, nothing but white noise, nothing but boredom. Sovereign Plumed Serpent and his friends talked and thought and worried, then they joined their thoughts and words, then they agreed. They knew what to do: they conceived the growth of trees, of bushes, of the growth of life, of humankind, there in the blackness, in the early dawn. They invented the universe, those great knowers, those great thinkers in their very being.*

—Suggested by the *Popol Vuh*<sup>1</sup>



---

## Abstract

The Mayan calendar is well suited to computer calculation, but existing programs are not extensible and are generally written in compiled languages, which limits their portability. Python is portable, extensible, and has builtin features that make processing dates in the Mayan calendar reasonably straightforward. A basic introduction to the Mayan calendar is presented, followed by discussion of some of the problems encountered using conventional languages, and some alternative approaches using Python are given. The areas of computerized parsing and special class methods in Python are covered. A discussion of recovering dates from partial inscriptions follows, with highlights of a CGI program to allow users to enter such partial dates and receive a list of possible solutions. Future directions for Mayan calendrical research with Python are suggested. The conclusion suggests that archaeologists and epigraphers in the field could use Python to help them pin down otherwise indeterminate dates in the Mayan inscriptions.

## 1 Introduction

When I told my wife that I was going to write a paper for the Python Conference in Houston, she asked what I was going to write about, and I said, “The Mayan calendar.” Not being familiar with Pythonists, she said “I thought that they wanted articles about *practical* applications for Python?”

While many people would not think of arcane calendars as a practical application for a tool, the Mayan calendar lends itself rather well to computer calculation, and there are some Mayanists out there now who know enough about computers to write programs; there are even more that are able to use the software once it exists.

My guess is that there are around 1000 people worldwide who consider themselves Mayanists; of these, perhaps half are “field archaeologists,” scientists not directly concerned with epigraphy or the calendar, except insofar as such collateral information helps date an excavation or dig. Of the 500 or so left, perhaps 20-50 are able to make their living in the field of epigraphy; another one or two hundred professionals’ lives are affected by advances in epigraphy; while the remainder are either students or, like myself, enthusiastic (sometimes obsessive) amateurs. Epigraphers are concerned not with the quotidian phenomena that interest field archaeologists, but with the texts the Mayans left us. Since so many of the inscriptions contain dates referring to contemporary elite persons, to cosmology or to gods in that cosmology, the study of the Mayan

writing system is intertwined with and inextricable from the study of their calendar.

Many epigraphers study Mayan political history and the ways in which the Mayan elite legitimized their temporal, secular power through the use of spiritual, sacred, location (Culbert, 1991; Schele, Grube and Martin, 1998). One of the ways the elite did this was to contrive large numbers representing time passed and then state on public monuments that such-and-such a ruler acquired great power because he was born exactly so many days after a particular deity was born, which just happened to be a nice round number of several sacred cycles (Lounsbury, 1978; Schele and Freidel, 1990; Schele and Mathews, 1998). Given such a world view, it is not surprising that calendrical statements permeate the monuments and codices. Familiarity with the Mayan calendar, therefore, is a prerequisite for epigraphic study.

While many epigraphers can use programs, and some can write them, most of these programs aren't very convenient. They have been written in conventional languages like C, C++, Pascal or Visual Basic, and such languages have little or no support for the multitrillion year calculations that can be commonplace in Mayan calendrical mathematics. Additionally, since these are all compiled languages, and compilers differ drastically between platforms, many routines are "write once, rewrite forever." It can be nearly impossible to get the algorithms to behave properly on a new platform.

I spent years (and I don't really want to admit *how* many years) writing libraries of C routines useful in working with the Mayan calendar. Among other fun tasks, I had to track down a multiprecision arithmetic library and then port it to my current Unix platform. Since it wasn't written very well, I spent a lot of time fixing bugs. When I moved to different Unices, I spent even more time porting the library. I am

reminded of the time in high school when I bought a 59-cent ship model that came with no detailing: it was maybe 4 inches long, and I decided to add full rigging. I spent the *entire* summer rebuilding that thing, and I ended up with a 59-cent model ship with \$2000 rigging.

In the same way, the C libraries I wrote for my Mayan calendar programs provided an interesting project for a very long time, but have fulfilled their purpose. The math functions provide a place to start, but they are only a starting point; it usually makes more sense to reevaluate what I'm trying to do and write something new in Python, which generally forces a cleaner, and almost always more accurate, solution.










The source code for Mayan dates (class *mayanum*), which is the Python replacement for and improvement of the C libraries, can be found at my website, along with some documentation: [mayalib.py](http://mayalib.py).

Before I can give examples of the benefits of using Python for the calculation of Mayan dates, some basic principles of Mayan calendrical mathematics need to be explained. In the interests of economy, I do not give a complete description here: for (many) more details, you may wish to refer to [An Introduction to the Mayan Calendar](#) and [The Calendar Round](#) on my website. Other websites are linked from there; however, the best references remain the original sources. I cite some of these sources below and in the two pages just mentioned.

## 2 Basics of the Mayan Calendar

### 2.0 An Example Date

An example of a Mayan date is "12.19.5.5.0 8 'Ahaw 13 Sots". This date is shown in Mayan hieroglyphic writing in **Table 1**.

The Long Count					The Calendar Round			
								
12.19.5.5.0					8 'Ahaw 13 Sots			

**Table 1: A Mayan Date**

## 2.1 The Long Count

The “12.19.5.5.0” part (the first 10 glyphs in the diagram above) is known as the **Long Count (LC)**. The Long Count is known as such because it is a linear count of days from a base date, which is usually taken to be Wednesday, August 13, -3,113 in our calendar, the Gregorian. There is some debate about the exact correlation date between the two calendars, but a majority of Mayanists prefer this one. Of the first ten glyphs above, the five that look like pictures are glyphs that stand for periods of time, and the bar-and-dot symbols are numbers. A dot is one, a bar is five, and the lobed symbol is zero. Mayan numbers are base (*radix*), 20, and are read either left-to-right as are our numbers, or from top to

bottom when numbers appear vertically. Mayan *dates*, however, modify the pure base 20 convention by using base 18 in the second position. Apparently, the motivation for this was to make certain calendrical calculations simpler, as it means that each unit of the third place had a value of 360 days instead of the expected 400, thus giving an approximation of the solar year.

Formulae ordinarily treat the Long Count positions as numbered from left to right starting at one, but it makes more sense, computationally, to begin numbering these positions at the right and start with an index of zero, as in **Table 2**, which shows the glyphs for the time periods as well as their names.











LC Digit	Mayan Digit	LC Position	Name	Glyph	Radix	Number of Days	Years
0		0	<i>k'in</i>		20	1	—
5		1	<i>winal</i>		18	20	—
5		2	<i>tun</i>		20	360	~1
19		3	<i>k'atun</i>		20	7200	~20
12		4	<i>bak'tun<sup>2</sup></i>		20	144000	~400

Table 2: The Long Count





The Tzolk'in (260 days)		The Haab (360 days)	
The Trecena	The Veintena	The Haab Day	The Haab Month
			
Value Ranges			
1-13 (8 shown)	0-19 (0 shown)	0-19, or 0-4 (13 shown)	0-18 (3 shown)

Table 3: The Calendar Round

## 2.2 The Calendar Round

The “8 ‘Ahaw 13 Sots” part of the date shown in **Table 1** is called the **Calendar Round (CR)**, and the various parts are shown in **Table 3**. It is a cycle of 18,980 days, about 52 years; its use was widespread throughout Mesoamerica, in many different forms. For a sweeping review of these many other calendars, see Edmonson, 1988.

While the Long Count is a strictly linear count from a base date, which is ordinarily written as “0.0.0.0.0 4 ‘Ahaw 8 Kumk’u,” arithmetic for computations involving the Calendar Round is mostly *modular*, that is, the four parts of the Calendar Round are not all positional numbers similar to what is used in the Gregorian calendar (or any ordinary numbers in everyday use), but represent *remainders that result from the division of a number*. The closest thing we have in the Gregorian calendar is the 7-day cycle of the days of the week. The four units that make up the Calendar Round are called the coordinates of the Calendar Round.

The Calendar Round consists of two cycles, one of 260 days (the *tzolk’in*) and another of 365 days (the *haab*). While  $260 \cdot 365$  is 94,900, 260 and 365 have a *greatest common divisor* of 5, which means that you can divide either 260 or 365 by 5 and multiply the answer times the other number. That is, the length of the Calendar Round is 18,980 days, since  $260 \cdot 73$  and  $365 \cdot 52$  both equal 18,980.

The *tzolk’in* is made up of a cycle of 13 day *numbers*, called the *trecena*, and a cycle of 20 day *names*, called the *veintena*. The *tzolk’in* and its constituent parts, the *trecena* and the *veintena*, are shown in **Table 3** in the leftmost half. The *haab* contains 18 named months, each of which has 20 days, numbered from 0 to 19. The last month, *Wayeb*, has only 5 days numbered 0-4. The *tzolk’in* is a *modular, reentrant* cycle, while the *haab* is an ordinary linear cycle, just like our year.

A cycle that is modular and reentrant is a cycle in which it is never necessary to resort to summing terms in order to find the position in the cycle, as one must for a linear cycle. For example, to find the number of the day of the year in the Gregorian calendar, you have to know the lengths of all the months in the year, add up all the whole months before the current month, and then add the number of the current day in the current month. In the *haab*, while we have to do basically the same thing, it is a little easier because all months are the same length, except the last. To find the a position in the *tzolk’in*,

we do not have to add; this position may always be recovered by the application of a *modular arithmetic formula*. This is discussed in more detail in section 2.4, below.

Whereas we stuff extra days into our years now and then, the length of the Mayan *haab* never changed. Since the *haab* drifts through the seasons in a 1460 year cycle, it is sometimes referred to as the “vague year,” because it has only a vague connection to the tropical year. The *haab* and its parts, the *haab day* and the *haab month*, are shown in the rightmost half of **Table 3**.

If we know all four coordinates of the Calendar Round, there a several pieces of information we can derive from them.

1. Numeric values representing each coordinate; once we have these, we can determine
2. The numeric position in the *tzolk’in*, and
3. The numeric position in the *haab*, and from those two positions,
4. The numeric position in the Calendar Round.

## 2.3 Conversion of Calendar Round Coordinates into Their Mathematical Equivalents

In Mayan calendrical mathematics as practiced by Mayanists today, the names of the *veintena* days and the names of the *haab months* are to be converted to their numeric equivalents; “‘Ahaw” is, mathematically, 0; the day name occupying position 1 is “‘Imix.” A complete list of *veintena* day names is available on my website. The *trecena*, “8,” is directly usable as a number. The *haab* components are just as simple; “13” is just 13, while “Sots” is the fourth month of the vague year. “Pohp” is month 0, so Sots is, numerically, 3. A complete list of the *haab* month names is also available on my website.

Conversion of our sample Calendar Round into its mathematical equivalent, then, gives us (in Pythonic terms) (8, 0, 13, 3). In the downloadable code, the function *parsecr()* will convert any reasonable Calendar Round string into a 4-tuple; the *parsecr()* function takes the process further, and converts the 4-tuple into a position in the Calendar Round.

## 2.4 Finding the Position in the Tzolk’in

Given the *tzolk’in* coordinates (8, 0) from section 2.3, we can determine the numerical position they refer

to. What we are doing here is recovering a *positional* number from two *remainders* obtained by dividing by two *moduli*; Knuth (1998) has a full discussion. Floyd Lounsbury (n.d.) provided several widely used formulae for working with the Mayan calendar; all the formulae here in section 2 are Python translations of these.

```
def p2601 (tr, v):
    return((40*((tr-1)-(v-1)))+(v-1))%260
```

For the example (8, 0):

```
tz = ((40 * ((tr - 1) - (v - 1))) + (v - 1)) % 260
tz = ((40 * ((8 - 1) - (0 - 1))) + (0 - 1)) % 260
tz = 59
```

Thus, converting “8 ‘Ahaw’” to its mathematical equivalent gives us 59, our position in the *tzolk’in*.

## 2.5 Finding the Position in the *Haab*

With the *haab* coordinates (13, 3) obtained in section 2.3, we can likewise determine the position in the *haab*, but with a simpler formula (again taken from Lounsbury, n.d.):

```
def phaabl(hd, hm):
    return(hm * 20) + hd
```

For the example (13, 3):

```
h = (hm * 20) + hd
h = (3 * 20) + 13
h = 73
```

## 2.6 Finding the Position in the Calendar Round

There are two steps in this process, the first of which involves finding the minimum number of 365-day units that separate the day we are interested in (8 ‘Ahaw 13 Sots) and the day that begins the Calendar Round: this is the *number of whole haabs* (**nH**). Finding nH requires the coordinates we found in the previous two steps, the position in the *tzolk’in* (**tz**) and the position in the *haab* (**h**), or (59, 73).

```
def nHl(tz, h):
    return(tz - h)%52
```

For the example (59, 73):

```
nH = (tz - h) % 52
nH = (59 - 73) % 52
nH = 38
```

The second step requires only two of the answers from the previous steps, the number of whole *haabs*

and the position in the *haab*:

```
def pCRL(tz, h):
    nH = nHl(tz, h)
    return(365 * nH) + h
```

For the example (38, 73):

```
cr = (365 * nH) + h
cr = (365 * 38) + 73
cr = 13943
```

Thus, the day “8 ‘Ahaw 13 Sots’” is equivalent to position 13943 in the 18980-day Calendar Round. It is also quite important to realize that the Calendar Round is locked to the Long Count in a particular way. Day 0 of the Long Count, “0.0.0.0.0,” is set to day “4 ‘Ahaw 8 Kumk’u” of the Calendar Round, which is position 7283. As each day goes by *both* the Long Count *and* the Calendar Round advance by one. More details are available on my website.

Moreover, since the Mayan calendar is rivaled in complexity only by the Gregorian (did you ever study epacts<sup>3?</sup>), it can have many more cycles and periods than I have described here (Carlson, 1981). I will mention just two additional cycles which can be useful in determining a precise Mayan date:

1. The nine-day Lord of the Night cycle, which operates much like the 7 days of the Gregorian week (Thompson, 1929); and
2. The 819-day cycle, containing 7 coordinates: four Calendar Round coordinates and a three-digit backward count of days (Thompson, 1943).

## 3 Representation and Conversion of Mayan Dates

Mayan dates, then, are composed of two major parts; a *mixed-radix* portion (the Long Count) and a *modular* portion (the Calendar Round). For use in computer programs, we need to convert user friendly (or at least Mayanist friendly) strings such as “12.19.5.5.0” and “8 ‘Ahaw 13 Sots’” into arrays, lists, structs or classes, and we need to be able to add, subtract, occasionally multiply, convert to Gregorian, and perform other functions on the resulting objects. Computer representation of these date objects is not simple; in C we could use structs, in C++, classes. In either one, converting from a string representation would require parsing the “12.19.5.5.0” input string, allocating memory and/or creating an instance of the struct or class, and filling in the appropriate fields in the struct/class from parsed values found in the

string. While it is possible to create a class constructor in C++ such that one could say

```
m = new mayanum("12.19.5.5.0");
```

and one could even extend the notation to the natural

```
m = m + "1.0.0";
```

but the equally natural

```
m = "1.0.0" + m;
```

is illegal, because the rules for operator overloading in C++ do not allow the first argument to be anything but the class for which the operator is defined. In Python the special class method mechanism, specifically `__radd__`, invites such intuitive usage, and the algorithms become simple to implement. Converting the Calendar Round string “8 ‘Ahaw 13 Sots’” is not quite as simple, but is certainly far easier than the equivalent method from C. Strings can be used as dictionary keys, so it is easy to allow users to type in names in several variant spellings and still be able to convert the names to numbers; the functions `matchveintena()` and `matchhaabmonth()` in the supplied code do exactly this.

Although most Mayan dates encountered have only 5 places, as seen above, these dates are essentially unbounded. One example, from *Coba*<sup>4</sup>, is “13.13.13.13.13.13.13.13.13.13.13.13.13.13.13.13.13.0.0.0;” this represents about 28 octillion years. (This date used to break nearly all PC programs; with Python, calculation is not difficult because of the built in support for long integers.)

In C, we could declare an array to store 5 places: (`int x[5];`) then, when parsing the input string given above, just ensure that we store the final “0” in `x[0]`. Not too difficult, but what about times when we have to evaluate a string like the date from *Coba*? Or one even longer? During parsing, we would have to figure out how many places, allocate the appropriate memory, and so on. If we added two such dates, then some routine would have to reallocate memory somewhere. Making sure that all the bases (pun intended) are covered isn’t easy—except in Python. Parsing and conversion of unlimited Long Count strings can be accomplished with very few lines of code:

```
def stringtomaya(s):
    list = string.splitfields(s, ".")
    lcl = map(string.atoi, list)
    lcl.reverse()
    return lcl # Put k'in in slot 0
```

In the supplied code, `stringtomaya()` is a class method. In order to deal with negative Mayan dates, the method is slightly longer than shown here.

## 4 Special Class Methods for Mayan Dates

Because Python has special class methods (“`__add__`,” etc.), it is almost trivial to implement the methods required to convert various representations to Mayan dates, and little more work to implement the actual addition, subtraction and multiplication methods that automatically convert those representations to the required Mayan dates and perform the appropriate functions. For example, one way addition could be built is to first convert Mayan dates to the equivalent **long**:

```
def mayatolong(m): # Reversed list
    n = long(0)
    bs = 20
    i = 0
    j = len(m)
    while i<j:
        if(i == 1):
            bs = 18
        else:
            bs = 20
        n = n+(m[i]*bs)
    return n
```

and then just add the two **longs**:

```
def addmaya(n, m):
    n1 = mayatolong(n)
    m1 = mayatolong(m)
    return n1 + m1
```

This would certainly work, but what if we wanted to perform the addition using something like the methods employed by Mayans? We know perfectly well that they did not convert their mixed radix numbers to base 10; we also know that they were more than capable of adding and subtracting huge, multiplace numbers without error, casting forward and backward enormous distances in time. In the following discussion, I will ignore complications like signs and negative numbers; Mayans could and did deal with negative numbers, but not the way we would. What they usually did was to say “count backwards so many days from a date, and you will arrive at a Calendar Round with coordinates so-and-so.” This is somewhat like saying, “count backwards 100 digits from +10, and the place you get to has a final digit of 0.” The actual location reached, -90, would never be written down that way, but it would be quite obvious that that was indeed the value we meant. Given that the Mayans seemed to view the integers as gods, referring to negative values without

ever writing down their names appears quite logical.

From various indications (Thompson, 1936; Justeson, 1989), we can infer that Mayan methods for adding were not too different from our own. That is, write the two numbers down, one above the other, and add each column. If the total is more than the maximum value for that column, depending on the radix, then simply carry to the next column. A simple implementation might look like this:

```
def addmaya(lc1, plus):
    t = []
    bs = 20
    carry = 0
    j = 0
    for i in lc1:
        n = i + plus[j] + carry
        carry = 0
        if j == 1:
            bs = 18
        else:
            bs = 20
        if (n > (bs - 1)):
            carry = n / bs
            n = n % bs
        j = j + 1
        t.insert(0, n)
    while(carry > 0):
        if j == 1:
            bs = 18
        else:
            bs = 20
        t.insert(0, carry % bs)
        carry = carry / bs
        j = j + 1
    return t
```

If we put this implementation into a class, then we can build a `__coerce__` method into it, and automatically have the ability to perform

```
n = n + "1.0.0.0"
```

and

```
n = n + 18980
```

which are both things Mayanists find themselves frequently having to do. Mayan monuments (“*stelae*”) usually start off by establishing a base date, known as the “Initial Series,” and then using what are called “Distance Numbers” to count from the Initial Series date, or from any of the secondary dates reached by counting from the Initial Series date. Often, these Distance Numbers will count backwards, so our class needs to support subtraction; sometimes the dates reached by these Distance Numbers will be before the zero day (0.0.0.0.0 4 ‘Ahaw 8 Kumk’u). A Mayan date class must support negative numbers and therefore signs, along with other attendant baggage. To fully support addition, we can rewrite our `__add__` method to support signed addition. With a little forethought, we can determine the cir-

cumstances under which a signed subtraction becomes a signed addition, and a signed addition becomes a signed subtraction. As long as simple addition is always supported by the `__add__` method and simple subtraction is always supported by the `__sub__` method, we do not have to worry about recursion, and can implement parts of each method in terms of the other. The full implementation of both `__add__` and `__sub__` can be found in the downloadable code.

There are some details found in that code that I have not talked about here. For instance, there is a class member, *sign*, which is the sign of the number; it is either 1 or -1. There are *grow()* and *shrink()* methods which are used to ensure that numbers subtracted or added have the same number of places in them. A *radix* member allows us to treat Mayan Long Counts (modified base 20) differently from Mayan *numbers* (unmodified base 20) automatically. The method *iszero()* lets us ensure that we don’t end up with something silly like “-0.” Several other special class methods are provided, such as `__abs__`, `__coerce__`, `__radd__`, `__mul__` and `__cmp__`.

Multiplication by a single integer can be done for mixed radix numbers without much trouble; it is much like addition, requiring the same attention to signs and carrying. However, you cannot multiply two mixed radix numbers directly. The only way it can be done is to convert both numbers to either the integer equivalent or to their equivalents in uniform radix notation; *e.g.*, to multiply 1.0.0 times 2.7.9, we need to convert either to integers ( $360 \cdot 869$ ) or to pure base 20 ( $18.0 \cdot 2.3.9$ ). I believe that the Mayans had some way to perform multiplication, whatever the multiplier, so there is no reason we can’t multiply  $2.7.9 \cdot 360$ . Mayanists have been surprised, many times over, by the sophistication of the Mayans’ numeric toolbox.

Using the `__coerce__` method, we can ensure that `__mul__` never sees numbers, or Mayan dates, it cannot deal with. We can also define a `__div__` method that works the same way, *i.e.*, changes the divisor to a single integer or converts both the divisor and dividend to pure base 20. While these implementations are nontrivial, I won’t describe them here; they too are in the downloadable code.

Further mathematical operations could be defined, such as a `__pow__` method, but since I’m only barely convinced of the utility of the `__mul__` and `__div__` methods, I have not done so. The most

useful methods when dealing with simple Mayan dates are `__add__`, `__sub__`, `__cmp__`, and `__coerce__` (and their corresponding `__r*` forms) although I have occasionally found use for `__lshift__`. None of these methods, however, requires the coordinates of the Calendar Round to be either calculated or known for the arithmetic to work properly. We can supply a `calculate()` method which, given a Long Count date, can easily provide matching Calendar Rounds. This method can be found in the downloadable code. Another extremely useful method is `gregorian()`, which does exactly what it says: Mayan dates can calculate the Gregorian calendar equivalent of themselves. Since there is still much debate over the exact correlation of Mayan dates and our own, Gregorian, calendar (Thompson, 1937), a means is provided to change the correlation date. All of these methods are quite useful when most or all of the various cycles and components are known, but it is an unfortunate truth that very many Mayan dates from the *stelae* have partially eradicated or unreadable dates, in which one can discern only parts of the Long Count and/or parts of the Calendar Round. What would be useful, then, is a set of methods and non-class functions designed specifically to deal with partial Long Counts and Calendar Rounds. This is something I always wanted to do, but could not with C, as the amount of work required for such a low level language was overwhelming. Using Python convinced me that the project could be easily managed, and so it proved.

## 5 The Recovery of Partial Mayan Dates

Lounsbury (n.d. and 1978) has described formulae for determining a set of Long Count dates from any given Calendar Round coordinates; since the Calendar Round recurs every 52 years, though, the formulae expect the user to have at least some idea of the bak'tun: this is a not unreasonable expectation, since the vast majority of Mayan dates recorded on the *stelae* are within the 9th bak'tun (435-830 CE), with some few in the 8th (41-435 CE) and some also in the 10th (830-1224 CE). Once you have such a list of Long Count dates, additional factors can help to determine the exact match for any given monument, the most notable factor being the nine-day cycle of Lords of the Night, referred to by "G numbers" (G9, G1, G2 and so on), since we don't know the names of these gods. If a Calendar Round, a bak'tun and a Lord of the Night are known, the exact Long Count date can be precisely determined.

However, the condition of some monuments can

reduce the amount of information available. Sometimes, a full Long Count and a full Calendar Round are not known. Most such loss of information is due to erosion or recent vandalism and looting. The Mayans would sometimes deface public monuments in such a way that faces and name glyphs of public personages became unreadable, but never, to my knowledge, deliberately obscured date glyphs. So I started to think about this; what is needed is a blank template into which users could insert all the items of information about a date that they could find on a particular monument, submit what they know, and get back a list of possible candidates. This should apply to any component of a Mayan Date, not just to the Calendar Round coordinates and a bak'tun. I thought about this a little more, and realized that if people were allowed to input just one number or day name, for example, the possible candidates would be infinite without some restrictions; even with restrictions, the candidate list could be extremely large, even though technically finite. The problem is not one of not finding *the* "correct" answer, but one of being overwhelmed with too many possibly correct answers. I then realized that the number of items in the candidate list could easily be precomputed; users could submit possibilities iteratively until the potential list became sufficiently limited to be comprehensible, and then choose, if possible, among the short list of choices.

I have implemented such a multistep interactive CGI program at

<http://www.pauahtun.org/cgi-bin/possible.py>.

It allows users to fill in a template which is then used to precalculate the number of possible answers; if there are not too many possibilities, it allows users to view a list of candidates and request more information on interesting candidates. The program only provides for five places of the Long Count, however.

One of Python's more useful features is run-time typing; this allowed me to build the input template with a menu system that lets users specify "wild cards" as digits in Mayan dates. When querying the user entries, the program just checks to see if any digits have been entered as "**None**." Those entries are wild cards, while digits actually entered come back as numbers. For example, a user might enter a Long Count as **12.16.13.None.None**; a Python function to calculate the number of possibilities inherent in this Long Count is actually fairly simple. You just multiply the possible values in each digit together; the maximum number of possibilities in



each place is the same as the radix in that place. If a digit is present, then there is only one possibility for that place. For the given example, the total is:

$$p = 1 \cdot 1 \cdot 1 \cdot 18 \cdot 20$$
$$p = 360$$

For Mayan Long Counts in the normal range (1-5 places), the maximum possibilities number 2,880,000, although since Long Counts are essentially unlimited in length, these maxima increase greatly with each addition of a digit. However, while these maxima are easily calculated for Long Counts, the Calendar Round is another story. Since the maximum radix is 18980, and there are only four components making up the Calendar Round, it turned out that the fastest and simplest way to determine maxima was to empirically determine all possible combinations and use a big **if** statement—there are only 24 different values that need to be returned. For example, if the user enters something like **4 'Ahaw 8 None**, then we know that there are only 18 possible dates in the entire 18980-day Calendar Round. This is because Calendar Round dates with the three coordinates ‘4 ‘Ahaw 8’ ((**4, 0, 8**)) can occur in each month of the *haab*, and there are 18 months. Again, we have a relatively simple procedure that can be implemented in not too many lines of Python. A method for the calculation of the *combined* number of possibilities, however, was not (and still is not) obvious to me.

Once we have the functions to calculate the number of possibilities, we need functions to actually build lists. These turn out to be somewhat harder, although the function for the Calendar Round is not too difficult; since there are only four components to the Calendar Round, the function can be written as four nested **for** loops. The complicating factors are:

1. the days of the *veintena* are legal, because of the mathematics, only on certain days of the *haab* month; and
2. the last month of the *haab* has only five days.

Thus, most of the code to return a list of possible Calendar Round positions is occupied with input verification.

The Long Count function appears deceptively simple: just cruise through the places, and, any time a **None** is found instead of a digit, use a **for** loop. But since the length of the Long Count component is essentially unlimited, it is somewhat harder than that. The only way that I think such a function could be written in **C** or **C++** would be to use recursion, since

those languages cannot compile and execute code they have written. Python can, and this proved to be the ideal solution. Two functions were required, though, not just one; one to look through the Long Count list and generate the Python code that changes for each selection depending on which places are wild cards, and another one to execute the written code which returns a list of Mayan dates that represent the possibilities.

Since I was not able to see a means to compute the number of possibilities using lists of possibilities from both Long Count and Calendar Round functions, I decided that the best way to determine the final list was to:

1. Use a two-step CGI program to precalculate LC possibilities of 8000 or less<sup>2</sup>; and CR possibilities of 949 or less, then
2. determine the actual list of CR possibilities;
3. pass the CR list to the LC function that writes Python code, which
4. uses the CR list to eliminate Mayan dates from the final LC list, and
5. executes the Python code to produce a final short list of Mayan dates, from which the user can obtain detailed information using another CGI program, by clicking on the desired date.

The final version of the Long Count function calculates Mayan dates for all possibilities in the submitted Long Count list, but each time it does so it checks the list of Calendar Round possibilities to see if the calculated Mayan date can possibly occur on any of the given Calendar Rounds; impossible Long Count/Calendar Round combinations are ignored. In **mayalib.py**, the function *newwritethecode* (*llc, crlist*) takes care of writing code from a supplied Long Count (*llc*) and passing through a Calendar Round list (*crlist*). Here is the function it wrote to run through all five of the normal positions in the Long Count (**[None, None, None, None, None]**):

```
from mayalib import *
tls = []
for s0 in range(20):
    for s1 in range(20):
        for s2 in range(20):
            for s3 in range(18):
                for s4 in range(20):
                    ls = [s0,s1,s2,s3,s4,]
                    tmp = mayanum(ls)
                    tmp.calculate()
                    if tmp.CR in crlist:
                        tls.append(tmp)
```

The function would change ranges if any **None** were replaced with a number. In the function to execute the above code, `llc` is the Long Count list and `tcr` is the Calendar Round list, and these variables are placed in the namespace of the `exec`'ed code:

```
def newexecthecode(str, llc, tcr):
    xx = [str]
    code = []
    for stmt in xx:
        code.append(compile(stmt, \
            "(execthecode)", "exec"))
    ns = {"llc":llc, "crlist":tcr}
    for stmt in code:
        exec stmt in ns, ns
    tls = ns["tls"]
    return tls
```

In `mayalib.py`, look for the function `listactuals()` to see how the actual lists are generated.

## 6 Future Directions

Knuth (1997) discusses permutations, which might be a fruitful area of study. Some means is needed to precalculate a final list of date possibilities from multiple input lists without generating all possible Long Count dates and rejecting some (or most). Such brute force approaches do not take advantage of the real power of Python.

The function that writes a function could easily be improved by adding more optional arguments, such as a list of possibilities for the Lords of the Night G series and another for the 819-day count. There are many more cycles which could be incorporated, but

## 8 Notes

1. The Plumed Serpent: I have paraphrased a section here from Dennis Tedlock's outstanding translation of the *Popol Vuh* (Tedlock, 1996: pp. 64-65); without the proper context, isolated quotations from this Mayan creation story can sound distressingly new age, which is about as far from the *Popol Vuh* as you can get. *Popol* means "council" and "vuh" is "book." Just as *Popol Vuh* means "Council Book," *Python Vuh* means "Serpent Book." In Classic Mayan religion, as near as we can determine, the Plumed Serpent is the Milky Way, and the *Popol Vuh* is a sky map in words (Schele, 1992). *Python Vuh*, then, is a serpent map in words.

Thanks are due to Lloyd Anderson, who helped me keep some of my estimates honest, and to Karl Taube, who kindly gave his permission to use the drawing of the Plumed Serpent (other drawings are my own). I would also like to acknowledge the assistance of Jeremy Hylton and Audrey Thompson, who suggested many improvements.

2. *Bak'tun*: While the other terms shown in the table are attested to and used by Mayans, *bak'tun* seems to be an invention of Mayanists rather than Mayans. Recent advances in translation have shown that the glyph for the 144,000-day period should most probably be translated as *pi* or *pih*, a term meaning "bundle." None of the terms for periods greater than 144,000 days (of which there are many) are attested. They should be recognized for what they are: terms invented for the convenience of Western anthropologists, archaeologists and epigraphers. See any of the recent *Workbooks* by Linda Schele for a detailed discussion.

it is not really necessary. The Long Count, Calendar Round, Lords of the Night and the 819-day count are the major cycles found on the monuments, and they are sufficient (without going into further detail<sup>6</sup>) to fix any Mayan date precisely within

**9305547427296816673725170526315789473684210  
5263157894736842105263157894736842105263157  
89473682240000**  
days, or about  
**2549465048574470321568539870223503965392934  
3907714491708723864455659697188175919250180  
245133376000**  
years.

That should do for a while.

## 7 Conclusion

I've described some new methods for Mayan date calculation in this paper; given access to a computer, and some training in the use of Python, many Mayanists may be able to make further discoveries on their own. With sufficient computing power in a laptop, some of the programs and Python functions described here may someday help epigraphers in the field to pinpoint the date of a newly discovered Mayan monument.

If that ever happens, I would like them to be able to say, "I couldn't have done it without Python." Or maybe, "... not without the Sovereign Plumed Serpent."



Scribners, 1978. Maya File 316e.

Schele, Linda and David Freidel. *A Forest of Kings: The Untold Story of the Ancient Maya*, New York: William Morrow, 1990.

Schele, Linda. *Workbook for XVIth Maya Hieroglyphic Workshop at Texas: Origins*. Austin, TX: Department of Art and Art History and The Institute of Latin American Studies, University of Texas, 1992.

Schele, Linda, Nikolai Grube and Simon Martin. *Notebook for the XXIIst Maya Hieroglyphic Forum: Deciphering Maya Politics*. Austin, TX: Department of Art and Art History, The College of Fine Arts, and The Institute of Latin American Studies, University of Texas, 1998.

Schele, Linda and Peter Mathews. *The Code of Kings: The Language of Seven Sacred Maya Temples and Tombs*. New York: Scribners, 1998.

Taube, Karl Andreas. *The Major Gods of Ancient Yucatan* (Studies in Pre-Columbian Art & Archaeology Number Thirty-Two). Washington, DC: Dumbarton Oaks Research Library and Collection, 1992.

Tedlock, Dennis. *Popol Vuh: The Definitive Edition of the Mayan Book of the Dawn of Life and the Glories of Gods and Kings*, Revised and expanded. New York: Simon and Schuster, 1996.

Thompson, J. Eric. "Maya Chronology: Glyph G of the Lunar Series." *American Anthropologist* n.s., 31, 1929: 223-31.

Thompson, J. Eric. "Maya Chronology: The Correlation Question." *Contributions to American Archaeology*, Volume III, Nos. 13 to 19; No. 14 (pp. 51-104). Washington, DC: Carnegie Institution of Washington, 1937.

Thompson, J. Eric S. "Maya Arithmetic." *Contributions to American Archaeology*, Volume VII, No. 36 (pp. 34-67). Washington, DC: Carnegie Institution of Washington, 1941.

Thompson, J. Eric S. "Maya Epigraphy: A Cycle of 819 Days." *Notes on Middle American Archaeology and Ethnology* No. 22, October 30, 1943: 122-151.

Thompson, J. Eric S. *Maya Hieroglyphic Writing: An Introduction*, Third edition. Norman: University of Oklahoma Press, 1971.

Van Laningham, Ivan. "Somewhere in Time: New Mathematical Methods for the 819-Day Count." Forthcoming in *U Mut Maya VII*, edited by Tom and Carolyn Jones. Arcata, CA: Humboldt State University Press.

The notation "Maya File" indicates that the paper so marked is available for a nominal fee from Kinko's, 2901-C Medical Arts Boulevard, Austin, TX (512 476-3242).

Those reading this paper in hardcopy may have difficulty clicking on the links. These are:

- An Introduction to the Mayan Calendar: <http://www.pauhtun.org/basic.html>
- The Calendar Round: <http://www.pauhtun.org/calround.html>
- The downloadable code: <ftp://www.pauhtun.org/mayalib.py>
- Numerology and the Astronomy of the Maya: [http://www.pauhtun.org/carlson\\_table.html](http://www.pauhtun.org/carlson_table.html)
- Interactive CR and LC Guessing program: <http://www.pauhtun.org/cgi-bin/possible.py>
- Other Python-powered cgi tools: <http://www.pauhtun.org/tools.html>
- The home page for the site is: <http://www.pauhtun.org/Default.htm>