A Short Course in Python for Number Theory

Jim Carlson

Draft of May 21, 2004

Contents

1 Introduction

These very brief notes are intended as a way to get started using Python for number-theoretic computations. Python has several advantages, including a clean structure and arbitrarily large integers as a native data type. As an interpreted language it is not particularly fast. But it is fast enough for many purposes, and is an good tool for learning and experimentation. With just a few lines, one do a great deal. See, for example, the implementation of the algorithms for finding the greatest common divisor, for solving the Diophantine equation $ax + by = c$, and for computing $a^k \bmod n$. Python handles these computations with ease even when the numbers in question are hundreds of digits long.

Once the student understands well the examples presented below, the combination of curiosity, mathematical knowledge, and some good references on Python will be enough to do many interesting computations.

The reader would certainly profit from having a standard Python tutorial handy. See, for example, [2].

2 Python as a calculator

Python can be used as a calculator:

```
>>> 2*7 + 11*3
47
>>> 2**10
1024
>>> 3/2
1
>>> 3.0/2
1.5
 >>> 12345 % 2
1
```

Notice what happened with division: the quotient of integers is the "integer quotient" — the one you get from long division. However, if you use a decimal point in the quotient, the numbers are treated as floating point numbers — the computer's version of real numbers. The very last operation `%` is the "modulo" or remainder operator. The result of `12345 % 2` is the long division remainder of dividing 12345 by 2.

As a calculator Python has some unusual capabilities. First, it can deal with arbitrarily long integers:

```
>>> 2**1000
    107150860718626732094842504906000181056140481170553360744375
    038837035105112493612249319837881569585812759467291755314682
    518714528569231404359845775746985748039345677748242309854210
    746050623711418779541821530464749835819412673987675591655439
    46077062914571196477686542167660429831652624386837205668069376
```

Second, we can use variables:

```
>>> m = 23.1
>>> a = 7.05
>>> F = m*a
>>> F
162.85500000000002
```

Third, Python has many built-in functions:

```
>>> max([1,2,3,4])
4
>>> from math import *
>>> sqrt(2)
```

```
1.4142135623730951
>>> exp(1)
2.7182818284590451
>>> log(_)
1.0
```

Let's think about what we did. The first function, `max`, found the largest element of the *list* `[1,2,3,4]`. We had to tell Python that we wanted to use its math library, which we did by saying `from math import *`. Here the symbol "*" stands for "everything." Then we could compute the square root of two. We also computed the value of the exponential function at $x = 1$. The last line introduces the variable _, which stands for the value of the last computation. Thus we verified the identity $\log(\exp(1)) = 1$, which of course is true since log is the inverse of exp.

Fourth, we can easily define new functions in Python.

```
>>> f = lambda x: x + 1
>>> f(1)
2
>>> f(2)
3
>>>g = lambda x: 5*x % 11
>>> g(3)
4
```

The first function is straightforward enough: it just adds 1 to its input. Functions defined this way always begin with the keyord `lambda`, which is then followed by a list of independent variables, a colon, and the formula defining the function. Note that `f` is a variable whose value is a function!

The function $g(x) = 5x \bmod 11$ is more interesting. To compute it we multiply x by 5 and then compute the remainder upon dividing by 11. Thus $f(3)$ is the remainder of 15 after dividing by 11, which is 4. To test our function, we make a little table of its values:

```
>>> for k in range(0,11):
...   print k, g(k)
...
0 0
1 5
2 10
3 4
4 9
5 3
6 8
7 2
8 7
9 1
```

The phrase `for k in range(0,11):` defines a *loop*. As the variable k runs over the integers in the range $0 \leq k < 11$, the values of k and $f(k)$ are printed out. Loops are used to automate repetitive actions. The expression `range(0,11)` is really the list `[0, 1, 2, ..., 10]`. You can verify this by typing `range(0,11)` in Python.

With a table like the one we just computed, it is easy to answer questions like "for what x is $5x$ mod 11 equal to 1?" What is the answer? You have just solved the congruence

$$5x \equiv 1 \bmod 11.$$

To test your command of Python so far, solve the congruence $77x \equiv 1 \bmod 101$.

Problems. (a) Compute 2^{1000}. (b) Compute $2^{100} \bmod 101$. (c) Solve $1234x \equiv 1 \bmod 2345$. (d) Find the smallest integer k such that $2^k \equiv 1 \bmod 101$. (e) The *order* of an element modulo N is the least positive integer k such that $a^k \equiv 1 \bmod N$. Find the order order of 3, 4, and 5 modulo 101.

Project. Find the order of each integer modulo 101. Is there a pattern to the results?

3 Basic programs

We will now write some simple programs in Python, doing just enough to see that with few lines of code one can accomplish a great deal. The main ideas are loops, conditional statements, and function definitions.

3.1 Series

The first program will use the idea of *loop*, which was introduced in the last section. Let us begin with the problem

Find the sum of the series $1 + 1/2 + 1/3 + 1/4 + \cdots + 1/100$.

Here is the program:

```
>>> sum = 0
>>> for k in range(1,101):
...   sum = sum + 1.0/k
...
>>> sum
5.1873775176396206
>>> k
100
```

We begin by setting the variable `sum` to zero. Then we repeatedly add `1.0/k` to `sum` using the loop `for k in range(1,101)`. Why did we have to use `range(1,101)` instead of `range(1,100)`? After the loop is complete, we type `sum` to find its value. We also check the value of `k`, just to be sure.

Below is another way of solving the same problem. We use a `while` loop instead of a `for` loop.

```
>>> sum = 0
>>> k = 1
>>> while k <= 100:
...   sum = sum + 1.0/k
...   k = k + 1
...
>>> sum
5.1873775176396206
```

You should become comfortable with both kinds of loops. Here is a problem to practice on:

Find the sum $1 + 1/2^2 + 1/3^2 + \cdots + 1/100^2$.

You can also investigate what happens when you take more terms in the series.

Let us now consider a slightly more difficult problem:

Find the sum $1 - 1/2 + 1/3 - 1/4 + \cdots + 1/99 - 1/100^2$.

To deal with this alternating series we add an `if-then-else` statement to our program:

```
>>> sum = 0
>>> k = 1
>>> while k <= 100:
...   if k % 2 == 1:
...     sum = sum + 1.0/k
...   else:
...     sum = sum - 1.0/k
...   k = k + 1
...
>>> sum
0.68817217931019503
```

When the remainder of k mod 2 is equal to 1, k is odd. Note that we test for equality using `==`. The symbol `=` is used to assign values to variables.

We are now going to investigate values of the sum

$$1 - 1/2 + 1/3 - 1/4 \pm \cdots + (-1)^{n+1}/n$$

for various n. To do so we shall modfiy the previous program so as to define a function `f` which computes the sum.

```
>>> def f(n):
...   sum = 0
...   k = 1
...   while k <= n:
...     if k % 2 == 1:
...       sum = sum + 1.0/k
...     else:
...       sum = sum - 1.0/k
...     k = k + 1
...   return sum
...
>>> f(100)
0.68817217931019503
>>> f(1000)
0.69264743055982225
>>> f(10000)
0.69309718305995827
```

Problem. Consider the series

$$1 + 1/2^2 + 1/3^2 + 1/4^2 + \cdots$$

Its n-th partial sum S_n is the sum of the first n terms. Study the behavior of S_n as n tends to infinity. Does it approach a limit? If it does, how accurately can you compute it? Repeat this analysis for the harmonic series

$$1 + 1/2 + 1/3 + 1/4 + \cdots$$

and the alternating harmonic series

$$1 - 1/2 + 1/3 - 1/4 \pm \cdots .$$

Problem. Define a function `factorial(n)` that computes $n! = 1 \cdot 2 \cdot 3 \cdots n$. Define also a function `binomial` to compute binomial coefficients. (a) In how many ways can we choose five cards (as in poker) from a deck of fifty-two cards. (b) In how many ways can we choose 13 cards from a deck of 52 cards, as in bridge? (c) Toss a coin 100 times. How many ways can this happen so that exactly 50 heads appear? (d) What is the probability that in 100 tosses of a fair coin, exactly half are heads?

Problem. Investigate the partial sums of the series

$$1 + 1/1! + 1/2! + 1/3! + \cdots$$

The limit of these sums exists and is, by definition, called e. Can you compute e to six decimal places? How many terms of the series are needed?

3.2 Counting

How many points are there with integer coordinates that lie in the circle of radius 100?

To study this problem we let $L(r)$ denote the number of lattice points inside the circle of radius r, where a *lattice point* is a point (m, n) with integer coordinates. It is easy to write a program to do the counting. We have used two nested `while` loops instead of two nested `for` loops.

```
def L(r):
  """L(r) = number of lattice points in circle of radius r."""
  count = 0
  m = -r
  while m <= r:
    n = -r
    while n <= r:
      if m*m + n*n <= r*r:
        count = count + 1
      n = n + 1
    m = m + 1
  return count
```

Project. Study the relationship between $L(r)$ and the area of the circle of radius r.

3.3 Brute-force search

We now give a second set of examples of simple programs organized around the theme of solving a problem by brute-force search. Brute-force search is what we do when we know of know better method or are too lazy to use it.

This time we will assume that the programs are written in a text editor and loaded into Python using an `import` statement. Our first problem is

Solve the congruence $ax \equiv b$ mod N.

Consider, for example, the congruence $1234x \equiv 1 \bmod 54321$. It is easy to tell whether a given x satifsfies the congruence. For example, if $x = 199$ we compute $1234 \times 199 = 245566$, then compute the remainder when we divide 245566 by 54321. The result is 28282. Thus $x = 199$ is not a solution. Our program simply tries all the integers $0 \leq k < 54321$, stopping when it finds a solution.

```
def csearch(a,b,N):
  for k in range(0,N):
    if a*k % N == b:
      return k
```

To solve the problem, we import the above definition, which resides in a file `search.py`:

```
>>> from search import *
>>> csearch(199,1,54321)
45313
>>> 199*_ % 54321
1
```

Note that we checked our answer. It is correct. This method of solving congruences is the least efficient possible. The standard, extremely fast method, is based on the Euclidean algorithm.

Let us now try a problem in two variables:

Find positive integer solutions to the equation $x^2 - 2y^2 = 1$.

Below is the program. It uses two loops, one nested inside the other. One loop controls x, the other y.

```
def psearch(n):
  for x in range(0,n):
    for y in range(0,n):
      if x*x - 2*y*y == 1:
        print x, y
```

Here is how the program is used:

```
>>> psearch(1000)
1 0
3 2
17 12
99 70
577 408
```

Project. Write a program to list Pythagorean triples. These are positive integer solutions of the equation $x^2 + y^2 = z^2$. Your program should eventually be designed to list only *primitive* triples. These are triples that have no common factor. When you have your data — a table of primitive Pythagorean triples — look for patterns.

Project. The *fundamental solution* of the equation $x^2 - Ny^2 = 1$ is its smallest positive integer solution. For each integer N in the range $[1, 100]$ find the fundamental solution if it exists. What general comments can you make about the existence and nature of fundamental solutions?

4 Some standard algorithms

We present here some standard number-theoretic algorithms: trial division for factoring integers, the Euclidean algorithm for finding the greatest common divisor and solving linear Diophantine equations $ax + by = c$, and the repeated squaring algorithm for computing $a^k \bmod N$.

4.1 Trial division

Trial division is very much like a brute-force search for factors. To factor n, set the trial divisor to $d = 2$. (a) If d divides n, print out the factor d and replace n by n/d. Otherwise replace d by $d+1$. (b) If $n = 1$, stop. There are no other factors. Otherwise, go to (a). Here is how the algorithm reads in Python:

```
def factor(n):
 d = 2
 while n > 1:
    if n % d == 0:
      n = n/d
      print d,
    else:
      d = d + 1
```

Here is how `factor` is used:

```
 >>> factor(1234)
2 617
>>> factor(4321)
29 149
```

The comma after the `print` statement is what causes the output to be printed on a single line. You should experiment with factorizations of other numbers.

There are various improvements that one can make to the trial division algorithm. We will give just one, based on the observation that if n has a factor, it has a factor no greater than $\sqrt{n}$. You should compare how the new algorithm performs relative to the old one.

```
def factor2(n):
  d = 2
  while n >= d*d:
     if n % d == 0:
       n = n/d
       print d,
     else:
       d = d + 1
  if n > 1:
    print n
```

We will give on other modification of the program. Instead of printing out the factorization, we will return it as a list of numbers. This list could then be used as the input of another program.

```
def factor3(n):
  d = 2
  factors = [ ]
  while n >= d*d:
     if n % d == 0:
       n = n/d
       factors.append(d)
     else:
       d = d + 1
  if n > 1:
    factors.append(n)
  return factors
```

We have added a statement `factors = [ ]` which sets up an empty list. Each time a new factor is found, it is added to the list: `factors.append(d)`. The statement `return factors` tells Python that `factors` contains the value to be returned when an integer is given as input.

```
>>> factor3(11223344)
[2, 2, 2, 2, 11, 43, 1483]
>>> len(_)
7
```

Note once again the use of the last-computed-value variable "_." Thus the expression `len(_)` computes the length of the list of factors.

Project. How does the time needed to factor a number increase with its size? Measure the size by the number of digits.

Project. Let $\pi(x)$ denote the number of primes less than or equal to x. Investigate this function. Can you compute $\pi(100)$, $\pi(1000)$, $\pi(10000)$? Etc?? How do these numbers grow?

Project. Investigate the series

$$1 + 1/2 + 1/3 + 1/5 + 1/7 + 1/11 + \cdots + 1/p + \cdots$$

where p is prime. In words, this is the series of reciprocals of primes.

Project. Investigate the factorization of numbers of the form $1 + N^2$. Can you draw any general conclusions or make any educated guesses?

4.2 The Euclidean algorithm

The Euclidean algorithm is a recipe for finding the greatest common divisor of two positive integers a and b. Assume that $a > b$. Take as many multiples of b from a as possible. Call the result r. Now repeat the procedure with b and r in place of a and b. Continue until b divides a. The last b computed is the greatest common divisor. Below is an implementation of the Euclidean algorithm. We have included a print statement so that you can see how the algorithm works. It can be eliminated once you are satisfied with it.

```
def gcd(a,b):
 r = a % b
 while r > 0:
   print a, b, r
   a, b, r = b, r, b%r
 return b
```

Here is an example of `gcd` in action:

```
 >>> gcd(12345,54321)
12345 54321 12345
54321 12345 4941
12345 4941 2463
4941 2463 15
2463 15 3
3
```

There is another way of designing the gcd function. It also uses the Euclidean algorithm, but in a sneakier way. Note that if b divides a, we already know the gcd: it is b. If b does not divide a, write $a = bq + r$, where q is the quotient and r is the remainder obtained from long division. Thus $0 \leq r < b$. If d divides a and b then d also divides b and r. Thus $gcd(a, b) = gcd(b, r)$. This observation is the basis of the following *recursive* implementation of gcd:

```
def gcd2(a,b):
  r = a % b
  if r == 0:
    return b
  else:
   return gcd2(b,r)
```

,

Problem. (a) Find the gcd of 123456789 and 987654321. (a) Find the gcd $10^{100} + k$ and $10^{100} - k$ for the integers k in the range $[1, 10]$. (c) Find the gcd of other pairs of interesting numbers.

Project. Investigate the probability that two positive integers chosen at random are relatively prime. The code below gives some idea of how to generate random numbers.

```
>>> from random import random
>>> for k in range(0,5):
...   print random(),
...
0.338646013248 0.747982433615 0.904639530247 0.967022080119 0.533538201167
>>> for k in range(0,10):
...   print int(5*random()),
...
2 3 2 2 0 4 0 3 3 0
```

4.3 Linear Diophantine equations

We will use the recursive implementation of the Euclidean algorithm to design a function `isolve` for solving the Diophantine equation

$$ax + by = c. \tag{1}$$

Such equations are solvable if and only if the $gcd(a, b)$ divides c. First note that if b divides a then we can write down a solution:

$$x = 0, \quad y = c/b.$$

If b does not divide a, write $a = bq + r$ as in the (long) divsion algorithm and then substitute into (1):

$$(bq + r)x + by = c.$$

Rearrange as

$$b(qx + y) + rx = c.$$

Set

$$u = qx + y, \quad v = x$$

and substitute again to obtain the equation

$$bu + rv = c. \tag{2}$$

We have reduced the equation (1) to the equivalent equation (2) with smaller coefficients. If we can solve the new equation, then we recover a solution of the old equation by the formulas

$$x = v, \quad y = u - qv.$$

The process eventually terminates (by the theory of the Euclidean algorithm) with an equaiton where the second coefficient divides the first.

These ideas are embodied in the Python code below. Note the use of the `divmod` operation to compute the quotient and remainder at the same time. The function `divmod` returns a pair of numbers, the quotient and remainder, which we store in `q, r`. Note also the use of lists for the return value.

```
def isolve(a,b,c):
  q, r = divmod(a,b)
  if r == 0:
    return( [0,c/b] )
  else:
    sol = isolve( b, r, c )
    u = sol[0]
    v = sol[1]
    return( [ v, u - q*v ] )
```

Here is an example: we solve the equation $12345x + 54321y = 3$.

```
>>> isolve(12345, 54321, 3)
[3617, -822]
```

Problem. Solve the congruence

$$98765432123456789 + 123456789 = 9.$$

Then solve the congruence

$$(10^{100} - 17)x + (10^{100} - 17)y = 1.$$

In each case, what is the number of digits of the solution compared to the number of digits of the coefficients?

Project. For Diophantine equations $ax + by = 1$, with $gcd(a, b) = 1$, investigate the relationship between the number of digits of the coefficients and the number of digits of the smallest solution.

Project. Design a function to solve the congruence $ax + by = c$ that does not use recursion.

4.4 Modular powers

Let us now find a way to efficiently compute powers like 71^{101} mod 107. The first step in writing a program is to understand how to do a typical computation by hand. The idea is to write the exponent as $101 = 2 \cdot 50 + 1$: we divide by 2, computing the quotient and remainder. Thus,

$$71^{101} = (71^2)^{50} \cdot 71^1 \equiv 12^{50} \cdot 71 \text{ mod } 107.$$

Thus we can compute 71^{101} mod 107 if we can compute 12^{50} mod 107. Our problem has been reduced to a smaller problem: the telltale sign that there is a recursive algorithm. Indeed, let us suppose that there is a function $modpower(a, k, n)$ which computes a^k mod n. Write $k = 2q + r$, where $r = 0$ or $r = 1$. Then

$$modpower(a, k, n) = modpower(a^2 \ \% \ n, q, n) * a$$

if $r = 1$ and otherwise

$$modpower(a, k, n) = modpower(a^2 \ \% \ n, q, n)$$

We can now write the corresponding Python code. Of course we have to make sure that we take care of the cases which make the recursion stop, namely, when the exponent is zero or one.

```
def modpower(a,k,n):
  if k == 0:
    return 1
  if k == 1:
    return a
  q,r = divmod(k,2)
  if r == 1:
    return modpower(a*a % n, q, n)*a % n
  else:
    return modpower(a*a % n, q, n)
```

Problem. Compute $2^{13739062}$ mod 13739063. What can you conclude from this computation?

Project. Define the *Fermat test* as follows: If $n > 2$ and $2^{n-1} \equiv 1 \bmod n$, then n is prime. Is this a foolproof test? Investigate and comment.

Project. Design a function to compute modular powers that does not use recursion.

5 Miscellaneous

5.1 Testing for randomness

Let $f(x)$ be a function which we apply repeatedly to a seed value to generate a pseudorandom sequence. To test it, we first design a function `orbit(f,a,n)` which computes an n-element list `[f(a), f(f(a)), ... ]`.

```
def orbit(f,a,n):
  """
  return the n-element list [a, f(a), f(f(a)), ... ]
  """
  r = a
  L = [r]
  k = 1
  while k < n:
    r = f(r)
    L.append(r)
    k = k + 1
  return L
```

For example, we set

```
>>> f = lambda x: 171*x % 30269
>>> orbit(f,1,5)
[1, 171, 29241, 5826, 27638]
```

Now we are going to study the behavior mod 2 of a long list. For this we need a function to compute frequencies for occurrence of events:

```
def tally(L):
  """
  tally(L):
  example -- tally([1,1,1,2]) = [[1,3],[2,1]].
  """
  T = []
  L.sort()
  curr = L[0]
  n = 1
  for x in L[1:]:
    if x == curr:
      n = n + 1
    else:
      T.append([curr,n])
      curr = x
      n = 1
  T.append([curr,n])
  return( T )
```

Now we can proceed. We will first reduce a partial orbit of f modulo 2 and count whether the the 0's and 1's are roughly evenly distributed. To do this, generate part of an orbit and store the resulting list in a variable `L`. To reduce the orbit modulo 2, we map the function `lambda x:  x % 2` onto the list, storing the result in a new list `LL`. Then we tally the list `LL`.

```
>>> L = orbit(f,1,1000)
>>> LL = map( lambda x: x%2, L)
>>> LL[0:5]
[1, 1, 1, 0, 0]
>>> tally(LL)
[[0, 524], [1, 476]]
```

If f is a good random number generator, we expect a tally of roughly five hundred 0's and five hundred 1's. But we don't expect an exact even split to occur very often. What is reasonable? The tally should lie within a few standard deviations of the expected value. The standard deviation is a measure of the spread of a random variable around its mean value. If the random variable is the number of heads in N coin tosses, then the expected number of heads is $N/2$ and the standard deviation is $\sqrt{N}/2$.

Project. Try further tests. Is f a good random number generator?

Note. If `f` is a function, then

```
map(f, [1,2, 3])
```

is the list `[f(1), f(2), f(3)]`. The expression

```
lambda x:  x*x
```

is the function which takes x as input and gives x^2 as output. Thus the value of the expression `map(lambda x:  x*x, [1,2,3])` is the list `[1,4,9]`. Of course we could write `f = lambda x: x*x` if want to re-use the squaring function. Then we could say, for instance, `map(f, [1, 2, 3])`.

5.2 Numerical integration

Recall that the integral of a positive function $f(x)$ for $a \leq x \leq b$ is the area of the region R under the graph, as in the figure below.

INSERT FIGURE

One way to compute an approximate value of the integral is to approximate the region under the graph by the region R' under a curve that is a broken line, as in the figure below.

INSERT FIGURE

The region R' is just a union of trapezoids, whose areas are easy to compute. We can summarize the computation as follows. Divide the interval $[a, b]$ into n equal subintervals with endpoints $x_0 < x_1 x_2$, etc., where rightmost endpoint is x_n. Let $f_i = f(x_i)$. Let $h = (b - a)/n$. Then

$$\text{Area}(R') = (f_0 + 2f_1 + 2f_2 + \cdots + 2f_{n-1} + f_n)h/2$$

Notice the form of the sum. It is a sum of terms $w_i f_i$ where the weights w_i are either 1 or 2. Other choices of weights give greater accuracy for roughly the same computational effort. One popular choice is Simpson's rule:

$$(f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \cdots + 4f_{2n-1} + f_{2n})h/3.$$

Below is an implementation of Simpson's rule.

```
def simpson(f,a,b,n):
  A = 0
  if n % 2 == 1:
    n = n + 1
  h = float(b-a)/n
  A = f(a) + f(b)
  x = a
  i = 1
  while i < n:
    x = x + h
    if i % 2 == 1:
      A = A + 4*f(x)
    else:
      A = A + 2*f(x)
    i = i + 1
  return A*h/3
```

Problem. Compute

$$\log 2 = \int_1^2 \frac{dx}{x}$$

to four decimals of accuracy.

Project. Compute

$$\text{li}(10^9) = \int_2^{10^9} \frac{dx}{\log x}$$

to 1 decimal of accuracy. This quantity (by the Prime Number Theorem) gives a good approximation to the number of primes less than 10^9. *Hint:* The function $\log x$ does not vary too much on intervals of the form $[a, 2a]$. On intervals iike this Simpson's rule gives accurate answeres with very few subdivisions.

Note. If $\pi(x)$ denotes the number of primes less than or equal to x, then the prime number theorem states that $\mathrm{li}(x)$ is an approximation to $\pi(x)$ in the sense that

$$\lim_{x\to\infty} \frac{\pi(x)}{\mathrm{li}(x)} = 1. \tag{3}$$

The prime number theorem was stated by Gauss but proved (independently) by Hadamard an de la Vallée Poussin. An even better approximation, due to Riemann, is

$$\pi(x) = \mathrm{li}(x) - \mathrm{li}(\sqrt{x}). \tag{4}$$

Riemann gave approximations that are even better yet. See [1] p. XX.

Project Use (4) to estimate the number of primes less than a billion.

6 Pseudorandom numbers

```
# Wichman-Hill random number generator.
#
# Wichmann, B. A. & Hill, I. D. (1982)
# Algorithm AS 183:
# An efficient and portable pseudo-random number generator
# Applied Statistics 31 (1982) 188-190
#
# see also:
#       Correction to Algorithm AS 183
#       Applied Statistics 33 (1984) 123
#
#       McLeod, A. I. (1985)
#       A remark on Algorithm AS 183
#       Applied Statistics 34 (1985),198-200

# This part is thread-unsafe:
# BEGIN CRITICAL SECTION
x, y, z = self._seed
x = (171 * x) % 30269
y = (172 * y) % 30307
z = (170 * z) % 30323
self._seed = x, y, z
# END CRITICAL SECTION


return (x/30269.0 + y/30307.0 + z/30323.0) % 1.0
```

References

[1] Edwards, Zeta Function, Dover.

[2] The O'Reilly books on Python are highly recommended.

[3] Havil's book re probability that random numbers are relatively prime.

[4] . A. Wichmann, B. A. and I.D. Hill, Algorithm AS 183: An efficient and portable pseudo-random number generator Applied Statistics 31 (1982) 188-190. see also: Correction to Algorithm AS 183, Applied Statistics 33 (1984) 123, and A. I. McLeod, A remark on Algorithm AS 183 Applied Statistics 34 (1985),198-200