Pythonで実験しながら学ぶ「暗号の仕組み」

藤本一男*

初版 2004 年 12 月 4 日

1　実験しながら暗号の仕組みを学ぶ

このレポートは、公開鍵暗号 (RSA) の仕組みを、簡単な計算を通じて理解することを目的としている。公開鍵暗号、そして、PKI（公開鍵暗号基盤）を理解することは、今日の社会における技術と社会の関係を学ぶ上で不可欠の要素だからだ。

ところが、暗号を数学の数式だけで理解できる人もいないわけではないが、多くの人は、「そうかもしれないけど、ふーん」というところから先にいかない。実際に計算しながら理解していこうと思っても、暗号の計算では、非常に大きな数を扱うことになるため、いくら小さな素数を選んで計算しても、電卓のように、せいぜい 10 桁くらいまでしか計算できない環境だと、あれこれ試すのはやはり辛い。また、C 言語、Perl といったプログラミング言語を使っても、非常に大きな桁の素数の計算をやらせるには、一工夫も二工夫も必要になる。

しかし、やはり、実際に計算しながら試すという経験は、数式の理解も促進するし、そこで行われている暗号化/復号という計算の正体もわかってくるというものだ。

なので、実験のできる環境を用意したいと思う。いわば、「暗号電卓」だ。

そこで、Pythonである。今日、Pythonは、Zope(Webアプリケーションサーバ http://www.zope.org) との関係で話題になっているが、残念ながら、書店でもあまりみかけない。同系列の言語では、国産の Ruby の方が有名になっている。実は、この Ruby でも可能なのだが、Python では、桁数制限のない整数を使うことができる。そのために、大きな素数の計算を、桁あふれをおこさないで直接行うことができる。

加えて、Python は、インタープリッターである。同じ計算式で、値を少し変えて振る舞いをみる、ということを、いちいちソースプログラムを書き変えることなく確認できる。

こんな理由から、Python を使って、暗号計算用の関数群を作ってみる。

- 桁数無制限整数が使える数値に long 型というものがあり、メモリが許す限り大きな桁数の数値を扱える。数字のあとに L をつけて表記する。
- インタープリッターの手軽さ windows や Linux のコンソールから使うこともできる。また、IDLE という GUI 環境も用意されている。さらに、emacs の python モードもある。
- 対応プラットフォーム Windows、Linux、FreeBSD をはじめとして、多くのプラットフォームに移植されている。このレポートでの実験も、DOS 上の Python1.0.1 で確認しながら行った。次のところに、各プラットフォーム用の Python 情報がある。

  http://www.python.org/download/download_windows.html

*作新学院大学 人間文化学部 E-mail:fujimoto@sakushin-u.ac.jp

2　実験をすすめるのに必要な知識

2.1　数学領域

- Modulo 計算　本文中では、mod と表記している。mod N とは、ある数を N で割った余りのこと。Python の演算子としては、%を使う。
- べき乗　Python の関数は、pow(x,y)。2.1 よりあとであれば、pow(x,y,z) で、$x^y mod z$ を一度に計算できる。
- 素因数分解　これは、力ずく...。
- 最大公約数　ユークリッドの互除法を関数にする。
- フェルマーの小定理　N が素数ならば、任意の a に対して、$a^{n-1} mod n = 1$ が成り立つ。これを満たす i を擬素数と呼ぶ。
- オイラー関数。オイラー/久留島の公式。

2.2　Python の使い方

- インストール
- 起動/終了
- 関数の作成
- チュートリアルの使い方

3　「シーザー暗号」をプログラムしてみる

この章の目的は、modulo 計算の特性を理解することにある。そのため、すでに、modulo 計算の特性を理解している読者は、次の章まで飛ばしてもらってかまわない。

シーザ暗号は、換置暗号の代表だが、このアルゴリズムを理解することは、ある数をある数で割った余り、の振る舞いを理解することでもある。

3.1　シーザ暗号

シーザ暗号とは、文字をずらすことで実現される。3 文字ずらずと、A → D、 B → E、となるから、

```
ABCDEFGH ...
||||||
DEFGH....
```

これを式で計算すると、各文字に番号をふれば、その番号に+3 することで暗号文が得られる。復号するには、暗号文から-3 して、文字に直せばいい。

では、実際にプログラムを組んでみる。一文字だけを暗号化/復号する関数なら、以下のようになる。

```
def caesar_e(a,n):
    return a+n

def caesar_d(a,n):
    return a-n
```

3.2 'z' を超える値をどうする if 文による条件分岐

しかし、これでは、うまくいかない。+3 した結果が、Z を超えるものは、A に戻してやらないといけないからだ。こういう場合にはどうするか。そう、if 文で条件をわけてやることが必要になる。a+n が、26 よりも大きい場合には、26 を引いてやる、と。改良版は、以下のようになるだろうか。復号の場合は、0 よりも小さくなったら、26 を足してやることになる。

```
def caesar_e(a,n):
        if a+n >= 26 : return a+n-26
        return a+n

def caesar_d(a,n):
        if a-n <= 0 : return a-n+26
        return a-n
```

これでどうにかなった。しかし、スマートではない。実は、このように、ある数（今は、アルファベットの 26 文字）を超えたら、0 に戻ってくる数の世界を表現するには「ある数をある数で割った余り」という関係を使う。これが、modulo 計算である。Excel でも、@mod() として関数が用意されている。

3.3 26 文字一回転を計算する剰余計算 (modulo)

M という数に n を加える。この結果、M+n が、26 になったら、0 に戻してやるには、

$$(M+n)mod(26)$$

という式を考えてやればよい。
同じように、ある数で一回転する例として次のようなものが身近なところにある。

- 曜日 mod 7

  曜日を、0 から 6 で表現する。月曜:0、火曜: 1 、... 土曜:6、という関係である。今日が、火曜日だとしよう。つまり、1 である。今日から 235 日後は、何曜日か。一週間が 7 日だから 253 を 7 でわって、その余りが... という計算をすることになる。これは、$(1+235)mod7$ で得られる。Python で計算させると次のようになる。

  ```
  >>> (1 + 235 ) % 7
  5
  ```

  つまり、5 なので、金曜日ということになる。

- 時間 mod 24, mod 60

  同じように、時間の計算でも、この mod は、使うことができる。今、18 時だとする。今から 58 時間後は何時か、という計算は、

```
>>> (18 + 58 ) % 24
4
```

  つまり、4 時だ。

この式を使うと、どのように一回りしているのかも手にとるようにわかる。7 ごとに 0 に戻っていることを確かめて欲しい。そりゃそうだろう、ということを目で見て足しかめらえる形にすることは大切である。時々、とんだ勘違いを発見することもある。ここが、実験の面白いところでもある。

```
>>> for i in xrange(30):
        print i, i % 7

0 0
1 1
2 2
(略)
6 6
7 0   <---
8 1
9 2
(略)
13 6
14 0   <---
15 1
16 2
(略)
20 6
21 0  <---
22 1
23 2
(略)
27 6
28 0 <---
29 1
>>>
```

3.4 if 文ではなく、剰余計算で、シーザー暗号を確かめる

この *mod* 計算 (プログラム上は、%という演算子) を使うと、先のシーザ暗号は、以下のように改造できる。

```
def caesar_e(a,n,m):
    return (a+n) % m

def caesar_d(a,n,m):
    return (a-n) % m
```

平文を a、鍵を n、アルファベットなので、m = 26 とすれば、0 〜 25 の値が循環する環境を表現することができる。

```
>>> a= 14 ; n = 16 ; m= 26
>>> casar_e(a,n,m)
4
>>> casar_d(4,n,m)
14
>>>
```

このように、平文 = 14 が、鍵 = 16 に対して、暗号化され 4 になり、それが、復号されて 14 に戻っていることがわかる。

これが連続した i の値に対してどう動くかどうかは、次のコマンド入力してみればよい。

```
>>> for i in range(29):
        print i, casar_e(i,14,26), casar_d(casar_e(i,14,26),14,26)

0 14 0
1 15 1
2 16 2
（中略）
9 23 9
10 24 10
11 25 11
12 0 12
13 1 13
（中略）
20 8 20
21 9 21
22 10 22
23 11 23
24 12 24
25 13 25
26 14 0
27 15 1
28 16 2
```

これをみると、a が 26 よりも大きい場合は、平文と復号文が一致してないことがわかる。これは、暗号化の関数で入力値の検査をおこななわなかったためである。入力された値が、0〜25 以外の場合は、エラーを表示するように改造してみてほしい。

以上、modulo 計算を実際に行いながら、その特徴を体験してもらった。

4　べき乗した値の modulo 計算を使った暗号を考える

さて、この *mod* の性格を使って、次のような暗号関数を考えてみる。M を平文。e を暗号鍵。N を素数。暗号文 C は、

$$C = M^e mod\ N$$

ここで、*mod* N の性格として、更に、C をなんども掛け合わせていくと、どこかで、M になる。もとに (つまり、余りが 1 になる) 戻ってくるようにできる。その数 ($C^d modN = 1$ となるような数 d) を見つけられればいい。それが、復号鍵 d になる。

辻井 [1999] は、次のような数を例に説明している（p174)。平文は、5。

$$M = 5$$

これを 3 回かけて、それを 11 で割って余りを計算する。(e,N)=(3,11) が暗号鍵)。

$$C = 5^3 mod\ 11$$

$5 \times 5 \times 5 = 125$ これを 11 で割るので、余り 4 となり、4 が暗号文である。ここで、次のことを確認する。

M を N-1 回かけあわせて N で割った余りは、1 になる（フェルマーの小定理）ことから、ここでは、5 を 11-1 回かけあわせれば、それを 11 で割った余りは、1 となる。実際に計算して確かめると以下のようになる。HP200LX では、Appnedix で説明する、powxem() を使用する。

```
>>> pow(4,4,5)       # N=5
1
>>> pow(7,733-1,733)   # N=733
1
```

つまり、e = 3 を暗号鍵にしたので、e^d を $N-1$ で割った余りが 1 となるような d が復号鍵になる。こうやって、d を求めることができる（力ずくでやってもいい)。この d を用いて、$C^d mod(N-1)$ を計算すると、5(=M) になっている。つまり、復号されたわけだ。

ところで、d を計算するのに必要な値（e,N）を公開してしまったら誰でも見れるので、受信者だけの秘密にはできない。残念ながら、これでは、まだ、公開鍵方式としては使えない。

次に、N を二つの素数で表現する、というアイデアが導入され RSA 暗号となる。そうすると、N は、素数ではなく、合成数となるが、この M を合成数 $(L = (p-1)(q-1))$ 回かけあわせた数を N でわった余りが 1 となるのである（これがオイラー/久留島の公式)。ここから、復号鍵 d を決めることができる。

この L という数字がわかれば、d は計算できるが、N という合成数が大きな二つの素数の積であるときは、この素因数分解は、きわめて困難である。そのために、L(つまり、$(p-1)(q-1)$) は、容易には、計算できない。

それで、暗号鍵 (e, N) が公開されていても、あらかじめ、復号鍵 d を知っている受信者だけが復号できる暗号方式が可能になるのである。

5 RSA 公開鍵暗号

ここからが本題である。まず最初に、必要となる数をあらかじめ用意した上で、暗号化/復号の計算をやってみる。それで、暗号化/復号、という処理が、どのような処理であるのかを理解してもらいたい。

なお、鍵を公開する暗号方式は、いろいろなものが考案されている。ここで、実験する仕組みは、RSA 暗号と呼ばれるものである。

5.1 まずは、公開鍵暗号を計算してみよう

- 暗号化　M を e 乗し、それを N で割った余りを C とする

M=5、N=77 (11×7)、e = 13

$$C = M^e mod N = 5^{13} mod 77$$

この計算をさせるには、Python の pow という関数を使えばよい。値をそのまま入力すれば、計算してくれる[1]。

```
>>> pow(5,13,77)
26
```

- 復号

```
>>> pow(26,37,77)
5
```

この復号鍵 d = 37 は、

$$13 \times dmod(11-1)(7-1) = 1$$

となる d である。

```
>>> (13*37) % ((11-1)*(7-1))
1
```

暗号化/復号の計算とは、これだけ、ということになる。

それでは、必要な数を Python で計算しながら、各ステップの内容を確めていこう。ステップは以下の通りである。

- 鍵生成：暗号鍵 (e)、復号鍵 (d) を生成する。
- 暗号化する
- 復号する
- (力ずくで) 復号を試みる。いわゆる暗号破り！

6 Python で計算しながら各ステップを確かめる

先の章で用いた計算用の関数をリストをあげて説明する。以下のものを用いて、自分で暗号文を暗号化し、それを復号することをやってみてもらいたい。

[1] 芹沢 [2002] でも、べき乗をそのまま計算するな、と書いてある (p155)。辻井［1999］も同様である。理由としては、直接計算したら、電卓などで計算可能な桁を超えてしまうからなのだが、それ以外に、modulo 計算をうまくつかえば、「小さな数」の積で値を計算できる、というところも理由になっているようである。しかし、どのような計算をしているのか、ということを理解する前の段階で、そのことを理解するのに、準備段階で、modulo の性格を理解し、何度も計算をおこなうのは、混乱のもとである。そういうことが、初学者の壁になることが考えられるので、本稿では、まずは、直接計算することを行う。桁数がおおきくなると、より効率的な計算を考えていく必要にせまられるが、それは、この段階を終了してからでいい。

6.1 鍵生成

鍵生成には、以下のステップを伴う。

- 素数を二つ手に入れ、p、q とする。
- 暗号鍵 e を計算する。
- 復号鍵 d を計算する。

6.2 素数を手に入れる

以下の方法は、厳密ではない。ある数 n が素数であるならば、n は、

$$2^{i-1} mod\ i = 1$$

という関係を満たすというフェルマーの小定理を使って i に対してチェックを行っているだけでだからだ。つまり、すべての素数は、この関係を満たすが、この関係を満たす数が、必ず素数になるということではない。そのために、この条件を満たす値を、擬素数と呼ぶ。より厳密に、素数であることを確認するには、ｘｘの確認方法をとることで可能である。ここでは、３桁か４桁の素数（である可能性が非常に高い）数を計算させて、暗号化、復号化の実験を行う。

```
>>> for i in xrange(1,1000):
        if pow(2L,i-1) % i == 1 : print i,

3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101
103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193
197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293
307 311 313 317 331 337 341 347 349 353 359 367 373 379 383 389 397 401
409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509
521 523 541 547 557 561 563 569 571 577 587 593 599 601 607 613 617 619
631 641 643 645 647 653 659 661 673 677 683 691 701 709 719 727 733 739
743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853 857 859
863 877 881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991
997
>>>
```

314 は、11×31　であるし、561 は、$3 \times 11 \times 17$ と素因数分解される。後者は、カーマイクル数、絶対擬素数と呼ばれる数である (一松,p178)

6.3 暗号鍵を計算する

二つの素数 p,q を選び、L=(p-1)(q-1) と互いに素な数 e を決める。

p、q は、この上の素数生成ルーチンの結果から選ぶ。$N = p \times q$ となる。ここで、$L = (p-1)(q-1)$ を計算し、その値よりも小さく、互いに素な数、e を決める。これが、暗号鍵となる（公開するのは、e と N)。

e も上の素数のリストから選べばいいが、それが、L と互いに素であることを確認するために、最小公倍数 (gcd) を計算し、それが 1 になることを確かめておく。

```
#
```

```
# ユークリッドの互除法による最小公倍数の計算
#
def gcd(a,b):
    while a:
        a, b = b % a, a
    return b
```

6.4 復号鍵の計算

すでに計算した、e,L(N ではない) から、復号鍵 d を計算しておく。ここで、d は、次の関係を満たす値である。

$$(e \times d) \ mod \ L = 1$$

これを、力ずくで求めてみる。(本当は、ユークリッドの互除法と平行してある計算をすれば回答がえら得るらしいが (太田、黒澤、p132,4.1 式)、うまくいってないので、当面、力ずくで計算。)

```
def dkey(e,l):#
    for i in xrange(l):
         if (long(i) * e )% l == 1 : return i
```

こうして、鍵生成は、

- 二つの素数、p、q、を選ぶ
- その積、$N = p \times q$ を求める。
- $L = (p-1)(q-1)$ を計算し、e を求める。
- さらに、d を求める

という過程であり、公開鍵は、(e,N)、秘密鍵は、(d,N) となる。これらの値を使って、メッセージ M を暗号化し、その暗号文 C を復号する。

6.5 暗号化：メッセージ M の e 乗を求め、それを N でわった余りを求める

以上の値、e,d,N があれば、暗号化、復号化の過程を計算しながら確認することができる。
そこで、必要になる計算式は、べき乗 (pow(x,y))[2]と、法 (modulo)(演算子は%) である。

$$C = M^e \ mod \ N$$

ここで、M が、暗号化される平文である。
これらを使うと次のように計算することが確認できる。
M=21 が、暗号鍵 (e,N)=(149,237077) で暗号化され、C=154833 になり、この C を、復号鍵 (d,N)=(114077,237077) で、復号し、ちゃんと 21 になっていることが確認できる。

```
>>> p=619 ; q=383
>>> N = p * q
```

[2]HP200LX でこの機能を使うためには、関数を書かなくてはならない。Appendix を参照のこと。

```
>>> L = (p-1)*(q-1)
>>> N,L
(237077, 236076)
>>> M = 21
>>> e = 149
>>> gcd(N,e)
1
>>> C = pow(M,e,N)
>>> C
154833L
>>> dkey(e,L)
114077
>>> pow(C,114077,N)
21L
```

7　公開鍵から秘密鍵を計算する困難を味わう

以上が、暗号化された暗号文を、復号鍵をもっている人が復号する過程である。ここで、公開鍵 (e, N) だけから、d を計算することをやってみたい。

7.1　鍵は素因数分解にある

d は、$(e \times d) mod (p-1)(q-1) = 1$ という関係を満たしている。そのために、N を素因数分解して、p と q の値を求めることが、目標となる。ある素数の積である N をもとの素数に分解するには、素因数分解を行う。しかし、これには、決め手となる方法は、ない。そこで、素数を生成しながら、順番に割り算をして、割り切れる数がないか確認することになる。なお、いくつまでやればいいかというと、$\sqrt{N}$ までである。

先に、素数を生成するルーチンを作成したが、それを使って、N を素因数分解するルーチンをつくることができる。それは、以下のようなものになる。

```
>>> def prime(N):
        for i in xrange(2,int(pow(N, 0.5))):
            if N % i == 0 : return i, N/i

>>> prime(77)
(7, 11)
```

ここでは、77 というように、小さな素数であるから、瞬時に、7,11 という値がえらえる。これを、さらに大きな素数の積を与えて計算してみて欲しい。この時間の増大が、暗号解読の困難さである。

7.2　力ずくの素因数分解にかかる時間を測定してみる

ちなみに、処理にかかった時間を測定することができる。time モジュールを import することで、time.time() が、現在時点を、秒で表現できるので、処理のはじめの時間を、処理終了時点の値から引けばよい。先の prime を改造して計算時間も出力するようにしてみよう。

```
>>> import time
```

```
>>> def prime(N):
        start = time.time()
        for i in xrange(2,int(pow(N, 0.5))):
            if N % i == 0 : return i, N/i ,time.time()-start

>>> prime(77)
(7, 11, 0.0)
>>> 977*953
931081
>>> prime(931081)
(953, 977, 0.070000171661376953)
>>> 2663*2797
7448411
>>> prime(7448411)
(2663, 2797, 0.0)
>>> 2963*2113
6260819
>>> prime(6260819)
(2113, 2963, 0.0)
>>> 9900601*2879
28503830279L
>>> prime(28503830279L)
(2879, 9900601L, 0.051000118255615234)
>>> 9900623*9900697
98023068434231L
>>> prime(98023068434231L)
(9900623, 9900697L, 113.0220000743866)
```

7 桁の素数の積の素因数分解で、2 分弱かかっている (Clock は、800Mhz)。この値は、桁が触れるほど大きくなっていく。それは、素数を生成する早さと同じ程度に大きくなるわけである。さまざまな桁の素数（擬素数）を計算し、この素因数分解にかかる時間を体感してみていただきたい。100 桁の素数を計算する時間を味わってみよう。

なお、同じ計算を、CPU クロックの異なるマシンで実行すると、処理にかかる時間の違いわかる。なお、HP200LX の計算をさせたら、N=28503830279L で xrange(N) が生成されなくなるようで、結果が出力されなくなった。そのため、xrange/range を使用しない prime を作成してみた。Appendix 参照。

A HP200LX(もしくは MS-DOS) で Python を使うために

A.1 Python1.0.1

このレポート執筆時点で、python は、2.4 が発表されている。python は、Linux、FreeBSD、Windows をはじめ、さまざまなプラットフォームにポーティングされており、DOS 版もあるのだが、残念ながら、DOS 版の「進化」は、停まっている。HP200LX で使おうとすると、Python1.0.1 しか選択肢はない。このファイルは、以下のところで手に入る。Lib や DOC も、1.0.1 用のものが参考になるが、DOS という制限されて環境にポーティングされているため、通常の 1.0.1 よりも機能としては制限されている。

http://www.python.org/ftp/python/pc/16python.exe.gz

http://www.python.org/ftp/python/src/python1.0.1.tar.gz

しかし、以下の点をクリアすれば、HP200LX でも使うことができるようになる。

A.2 8087 エミュレートモジュールの常駐 em87 /L

浮動小数点計算、数学モジュール (math)、時間モジュール (time) を使おうとすると、エラーが表示される。フローテイング・ポイント計算用のコプロセッサがない、というのだ。これは、8087 をエミュレートする em87 をロードして回避することができる。ダウンロード先は、以下の URL。

http://senior.billings.k12.mt.us/files/sw-dos/EM87.ZIP

なお、これを入れても math モジュールは使えるようにはならない。time は使うことができる。そのため、動作時間を秒で測定るうようなステップは以下のように書くことができる。

```
import time
def now():
    import time
    return time.time()

a=now()
なにかを実行
now()-a
```

A.3 内観 (introspection) で内部をさぐる

Python1.0.1 のパッケージは入手可能である。問題は、DOS 用のものは、それのサブセットになっているというところにある。Doc にあっても、そのようには実装されていないことがある。

しかし、Python には、内観という方法あり、それを使えば、どのプログラムがどのような機能を有しているかを調べていくことができる。

- dir()、–methods–などで、使える世界をさぐる

機械伯爵さんによる、内観のメモ。

http://www.python.jp/pipermail/python-ml-jp/2002-July/001565.html

このスレッドは、私がなげた DOS 用の Python についてのものであり、上のものは、その書いてでかかれている。

A.4 使える関数、使えない関数、数値計算のためのポイント

- string は、strop という名前。文字列処理の基本的なものは見つかる。
- re はない。regrex を使う。正規表現もなんとかなる。
- math モジュールを import しても math の関数を使用できないのがつらいが、基本的なところは、なんとかなる。pi と e は、定数として使える。
- 小さな数でも整数 Overflow が起ることがあるが、これは、処理する数を long にキャストすれば回避される。

  long()
- 平方根の計算に、math.sqrt() を使えないが、em87 をロードして、pow の引数に、0.5 を与えてやれば、平方根を計算する。

```
>>> pow(2L,2)
4L
>>> pow(2,0.5)
1.4142135623730951
>>>
```

A.5 Python1.0.1 で、pow(x,y,z) を行う

Python2.3.4 などでは、pow(x,y,z) を使えば、$x^y mod z$ を直接計算できる。しかし、DOS で使える python1.0.1 の pow() は、x,y しか引数にとれない。$pow(x,y)\%z$ という計算も可能であるが、pow(x,y) を直接計算すると非常に時間がかかる。そこで、powxem という関数を作成する。この関数は、M をべき乗する数、e を 2^k の積で表現して、計算ステップを減らしている。そのために、e の値を二進数で表現し (getbin(e))、それをリストとして保持し計算させている。(太田 1995、p222)

```
#
# powxem(x,e,n)
#
def powxem(x,e,n):
    p = 1
    k = 0
    for i in getbin(e)[1]:
        if i == 1 :
            p = p * pow2km(long(x),k,n)
        k = k + 1
    return p % n
#
#
def getbin(x):
    bin = []
    i = 0
    while(1):
       d = x >> i
       if d == 0 : return (i,bin)
       bin.append(d & 1)
       i = i + 1
#
# pow2km(x,k,n)
#
def pow2km(x, k, n):
    for i in range(k):
         x = pow(x, 2L) % n
    return x
```

A.6 for/xrange()/raneg() を使わない prime()

xrange()/range() は、引数に整数をとる。Long は、とれない。そこで、整数はいくつまでなのかを確認してみると、

```
>>> import sys
>>> sys.maxint
2147483647
>>> '%x' % sys.maxint
```

```
7fffffff
```

先に、xrange() でリストが生成できなかった数は、この maxint が示す数よりも小さい。なお、16 進表記でわかるように、整数は、31bit である。整数の桁あふれが問題ではなさそうだ。DOS 版の実装のバグだろうか。

そこで、for 文を用いないで、ループさせる prime() を以下のように作ってみた。

```
>>> def prime(N):
        i = 2 ; end = int(pow(N, 0.5))
        start = time.time()
        while(1):
                if N % i == 0 : return i, N/i, time.time()-start
                i = i + 1
                if i >= end : return

>>> prime(77)
(7, 11, 0.0)
>>> prime(28503830279L)
(2879, 9900601L, 19.76999999809)
```

この 19 秒というのが、HP200LX(クロック 32M) での計算時間であるが、800M のノート PC では、0.019999980926513672 であった。

B WindowsCE での Python

B.1 CE2.11 で Python1.5.2

http://starship.python.net/crew/mhammond/ce/old.html

B.2 PPC2002 で、Python2.2+

http://www.murkworks.com/Research/Python/PocketPCPython/Overview
Telion さんのページ http://pages.ccapcable.com/lac/PythonCEj.html

C どこでも Python！

HP200LX や PocketPC などで、Python を動かせると、通勤途中でプログラムを組むことができる。ノート PC でやってもいいのかもしれないが、電卓気分で、かなり高度なことを実験できるのがなんともいえない。

PocketPC でも Python は動いているが、いかんせんキーボードがないので、苦しいところがある。その点、HP200LX は、いい。先に書いたように、Python の version は古いし、カレントの Version では当たり前に使えるものが使えなかったりするが、それは、がんばって書けばいいわけで、それもまた、楽しみになっている。そんなわけで、HP200LX は、いまだに現役。

D　おまけ：解析的感性 vs 整数論的感性

学生時代は、理工学部に在籍していたので、数学は一応やっている。だから、暗号の計算も数式を追っていけるだろうと思ったのだが、かなり険しい現実を思い知らさせた。本レポートで書いた *modulo* の計算をなかなか感覚としてつかむことができなかったのだ[3]。0 から 25 までは一つづつ値増えていく。それはいいのだが、それが、次の瞬間 (!)、0 に戻る。これについていけなかった。微分方程式の世界では、このような急激な変化はない。自然界は、なだらかな連続量の世界として記述され、その運動の秘密が、微分方程式に表現されていた。その意味で、私は、高校時代から、微分・積分で表現される自然法則が好きだった。また、その応用である、演算子法による自動制御の世界も好きだった。つまり、アナログの世界の感受性を育ててきたということのようである。

暗号論の世界は違った。隣に突然、予期せぬ (ではないのだろうが) 値が現れる。連続量ではなく離散値の世界。これが、数式で表現されることに慣れるには、かなり時間がかかった、というのが正直なところである。デジタルとはこういうことをいうのだと思い知らされた。時計に文字盤があるのが「アナログ」で、数字で表示されていると、それが「デジタル」などという通俗的な解説がいかにインチキかがわかった。時の流れは、アナログなのだ。それを文字盤と針で表していようと、数字で表していようと、表現されている時間は連続量である。デジタルな時計をつくったら、23 秒の次に、56 秒が表示されるようなものになるのだろう。

そんな風に考えるようになり、暗号のことを考えるのであれば、この整数論的、離散値的感受性をやしなわなくてはないと思うようになった。そして、その方法を探していたのだが、やはり、実際に計算をしてみることだろうと考えた。思いたったのが二年前。今の職場に転職した夏休みにここに書いたような「暗号電卓」のようなものを作ろうとしたのだが挫折。それから二年たって、やっと、素数の世界で遊ぶ環境を手に入れることができた。

やはり、実際に計算してみて、式が持っている意味を確かめられることの意味は大きい。教科書、解説書、は、限られた例しか載せてないが、「暗号電卓」を使えば、実にいろいろな例を試すことができる。解析的感受性が整数論的感受性に転換したのか、はたまた、両方の感受性を保持することになったのかは、よくわからないが、素数を扱い、暗号の仕組みをあれこれ計算することが面白くなる。わかってしまえば「なんだ、そんなことか」という面もある。しかし、それは、自分の理解をプログラムという表現で確かめられたという実感に支えられているのだろう。こうした実感を学生諸君にも味わってもらいたい。それも、理系の学生はもちろん、自分は文系で、数学に縁がありませんでした、というような学生達にだ。この暗号電卓をそんな実習用のものへと成長させていこうと思う。

コンピュータ以前の自動制御は、機械や油圧の平衡系を基礎にしたものであったが、コンピュータは、言語の原理をもとにした自動制御を可能にしたと言う（xxx『コンピュータと社会科学』大月書店)。まさに、不連続の世界。コンピュータが偏在化する社会とは、この不連続の世界をあやつることを求めている。そうしたものを内包した社会を運営していくには、こうした、整数論的な感性も必要になるように思う。もちろん、そういった不連続量の代表は、言語なので、まったくあたらしい世界というわけではないはずなので、実験しながら理解を確かめていけば、こいつの正体も落ち着いて見極めることができるのではないだろうか。秩序をもった不連続の世界で遊ぶ面白さを共有しながら、文系だ理系だという垣根もとりはらっていきたい。＜ふ＞

[3]そういえば、受験問題でも、mod の問題は苦手であった。

参考文献

[1] 太田和夫・黒澤馨・渡辺治『セキュリティの科学』講談社ブルーバックス、1995

[2] 一松 信『暗号の数理 作り方と解読の原理』講談社ブルーバックス,1980

[3] 辻井重男『暗号と情報化社会』文春新書、1999

[4] ルドルフ・キッペンハーン (訳:赤根洋子)『暗号攻防史』新潮文庫,2001

[5] Simson Garfinkel(監訳：山本和彦)『ＰＧＰ　暗号メールと電子署名』オライリー・ジャパン、1996

[6] 芹沢正三『素数入門』講談社ブルーバックス,2002

[7] http://www.python.jp PyJUG(日本 Python ユーザ会)

[8] http://www.python.org Python 本家

[9] 山根信二さんの「クリプトアナーキスト (暗号自由主義者) のページ」http://www-vacia.media.is.tohoku.ac.jp/~s-yamane/articles/crypto/

[10] Eric Raymond,『ハッカーになろう』このなかに、プログラミング勉強用の言語として Python が推奨されている。http://cruel.org/freeware/hacker.html

[11] Mark Lutz, David Ascher, 紀太　章：訳『初めての Python』オライリー・ジャパン, (2000/9 2004/11 に第二版。)

[12] Mark Lutz, 飯坂剛一：監訳, 金森玲子：訳『Python デスクトップリファレンス』, オライリー・ジャパン,1999/8 (誤訳、誤植が多いのだが、それを訂正しながら読んでいくと、自分だけのデスクトップ・リファレンスができあがるので、それもよしかと思うようにしている。)

目 次