Computations in Number Theory Using Python: A Brief Introduction

Jim Carlson

March 2003

Contents

1 Introduction

The aim of these notes is to give a quick introduction to Python as a language for doing computations in number theory. To get an idea of what is possible, we first observe that Python can be used as a calculator. Imagine that we are using our portable unix laptop, logged in as `student`. The first step is to type the command `python`. Then we type the actual python code. Here is an example:

```
student% python
>>> 2**1000
10715086071862673209484250490600018105614048117055
```

```
33607443750388370351051124936122493198378815695858
12759467291755314682518714528569231404359845775746
98574803934567774824230985421074605062371141877954
18215304647498358194126739876755916554394607706291
45711964776865421676604298316526243868372056680693
76L
>>> 2**1000 % 1001  # compute remainder
562L
<control-D>
student%
```

We first computed 2^{1000}. Then we computed 2^{1000} mod 10001: that is, we compute the remainder of 2^{1000} upon division by 1001. This is done using the operator `%`. Thus `5 % 2` is `1`. The suffix `L` stands for "long integer." Python can handle integers of arbitrary size, limited only by the available memory. To exit Python, type control-D.

One consequence of our computation is that 1001 is composite: a theorem of Fermat states that if n is an odd prime, then 2^{n-1} is congruent to 1 modulo n. Thus it is possible to prove that a number can be factored without actually factoring it. In section 6 we discuss an algorithm that implements this "Fermat test" very efficiently.

Consider next the fundamental problem of factoring an integer into primes. One can do this by trial division. Given n, divide by 2 as many times as possible, then by 3, etc. Record the factors found, and stop when there are no more factors to be found — when the running quotient equals one. A Python implementation of trial division is given by the code below.

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

As soon as the function `factor` is defined, we can use it:

```
>>> factor(1234)
[2, 617]
>>> factor(123456789)
[3, 3, 3607, 3803]
```

```
>>> factor(12345678987654321)
[3, 3, 3, 3, 37, 37, 333667, 333667]
```

In the definition of `factor` it is important to be consistent about indentation. We used two spaces for each level of indentation. The dots ... are typed by Python.

We will study the code for `factor` in detail in section 3. For now, however, let us note two important features. The first is the `while` construction. This is a *loop* which repeatedly performs certain actions (the indented text below the line beginning with `while`). The actions are repeated so long as the condition `n > 1` is satisfied. The second feature is the use of `if ... then ... else ...` statements to make decisions based on program variables. If `d` divides `n` we do one thing (add `d` to the list of factors, divide `n` by `d`. If it does not divide `n`, we do something else (increase the trial divisor `d` by `1`). The `if ... then ... else ...` construct is a *conditional.* Loops and conditionals are basic to all programming.

Another way of defining functions is *recursion.* A recursive function definition is one that refers to itself. The function which computes $n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$ is a good example:

```
>>> def factorial(n):
...   if n == 0:
...     return 1
...   else:
...     return n*factorial(n-1)
>>> factorial(3)
6
>>> factorial(100)
93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000L
```

For more information on Python, consult the on-line tutorial by Guido van Rossum [3], explore the `python.org` web site, or consult the O'Reilly books on Python.

Exercise 1 *Experiment with Python: try some computations using it as a calculator, then enter the code for* `factor` *and experiment with it. Use control-C to abort a computation if it takes too long.*

Exercise 2 *Devise and test a function* `fib(n)` *which returns the n-th Fibonacci number F_n. F_n is defined by the initial conditions $F_0 = 1$, $F_1 = 1$ and by the recursion relation*

$$F_n = F_{n-1} + F_{n-2}.$$

2 Python as a calculator

Let us look at some more examples of Python used as a calculator. The comments (preceded by "#") explain some of the fine points of the computations.

```
>>> (2 + 3)*(5 - 22)
-85
>>> 5/2           # note integer division
2
>>> 5/2.0         # decimal point for floating point arithmetic
2.5
```

We can also do arithmetic with complex numbers:

```
>>> z = 1 + 2j
(1+2j)
>>> w = z**2
(-3+4j)
>>> abs(w)
5.0
>>> w.real
-3
>>> w.imag
>>> 4
```

One can also say `z = complex(1,2)` to construct `1 + 2j`.

It is easy to set up variables and use them in computations. We illustrate this by computing our bank balance after 10 years, assuming just an initial deposit of $1000 and an interest rate of 5%:

```
>>> balance = 1000
>>> rate = .05
>>> balance = (1+rate)**10*balance
>>> balance
1628.894626777442
```

We could also simulate the balance in our account using a `while` loop:

```
>>> balance, rate = 1000, 0.05
>>> year = 0
>>> while year <= 10:
...   print year, balance
...   balance = (1 + rate)*balance
...   year = year + 1
```

```
...
0 1000
1 1050.0
2 1102.5
3 1157.625
4 1215.50625
5 1276.2815625
6 1340.09564063
7 1407.10042266
8 1477.45544379
9 1551.32821598
10 1628.89462678
```

In any case, it is wise to invest.

As noted in the introduction, we can define new functions. Consider, for example,

$$f(x) = x^2 \text{ modulo } 101$$

It can be defined in Python by

```
f = lambda x: x*x % 101
```

or by

```
def f(n):
  return x*x % 101
```

The first expression assigns the *lambda expression* `lambda x:  x*x % 101` to the name `f`. A lambda expression consists of two parts. The first part, from `lambda` to the colon tells us what the list of independent variables is. The second part, following the colon, is an expression which defines the function value.

Whichever style of definition we use, f defines a secquence of numbers $\{x_n\}$ by the rule

$$x_{n+1} = f(x_n)$$

where x_0 is given. For example, if $x_0 = 2$, find, calculating by hand, that $\{x_n\} = \{2, 4, 16, 54, ...\}$. The same computation can be done in Python as follows:

```
>>> f(2)
4
>>> f(_)  #   _ = result of last computation
16
>>> f(_)
54
>>> f(_)
88
```

We could also use a loop to compute elements of the sequence:

```
>>> while k <= 10:
...   print k, x
...   x = f(x)
...   k = k + 1
...
0 2
1 4
2 16
3 54
4 88
5 68
6 79
7 80
8 37
9 56
10 5
```

The orbit of $x_0 = 2$ under f is the largest nonrepeating sequence $x_0,\ x_1,\ x_2,\ \ldots, x_n$. What is the orbit in this case?

Python supports many data types besides integers, long integers, floating point numbers, and complex numbers. Among the other types are strings, lists, tuples, and dictionaries. See [3]. However, lists are especially important, so we consider them now. To begin, we set up a list `L` and then add an element to it.

```
>>> L = [11,12]         # two-element list
>>> L.append(13)
>>> L
[11, 12, 13]
>>> primes = [ ]        # empty list
>>> primes.append(2)
>>> primes.append(3)
>>> primes.append(5)
>>> primes
[2, 3, 5]
```

We can add lists, assign the result to a new variable, and compute the length of a list:

```
>>> L2 = L + primes
>>> L2                  # display L2
[11, 12, 13, 2, 3, 5]
>>> len(L2)             # length of list
6
```

We can also work with individual list elements:

```
>>> L[0]
11
>>> L[0] = -1
>>> L
[-1, 12, 13]
```

One way of constructing lists is to use `range`:

```
>>> L = range(1,10)
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Think of `range(a,b)` as the list with elements `x` satisfying the inequalities `a <= a < b`. With a `for-in` loop one can scan through all elements in a list, performing an action which depends on the current element. We illustrate this in the next example, which computes a table of square roots:

```
>>> from math import sqrt
>>> for x in range(0,10):
...   print x, ":", sqrt(x)
...
0 : 0.0
1 : 1.0
2 : 1.41421356237
3 : 1.73205080757
4 : 2.0
5 : 2.2360679775
6 : 2.44948974278
7 : 2.64575131106
8 : 2.82842712475
9 : 3.0
```

Note that in order to use the `sqrt` function we had to `import` it from the `math` module. If we had wanted to import all of the functions in `math`, we would have said `from math import *`. For a description of the Python modules, see `python.org/doc/current/modindex.html`.

Another useful feature of Python is the ability to apply a function to all elements of a list. In the example below we use `map` to apply `lambda x:x*x` to the list `range(0,10)`. The result will be a list of squares.

```
>>> map( lambda x: x*x, range(0,10) )
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Exercise 3 *Experiment with Python as a calculator.*

Exercise 4 *Find all the orbits of* $f(x) = x^2$ *mod* 101 *on the set* $\{0,\ 1,\ 2,\ \ldots, 100\}$.

3 Case study: factoring

Let us now study the code for `factor`, which we will call `factor1` in this section. To begin, we read through the commented version below:

```
>>> def factor1(n):          # def sets up definition
...   d = 2                  # set trial divisor d to 2
...   factors = [ ]          # set list of factors to empty list
...   while n > 1:           # loop while n > 1
...     if n % d == 0:       # does d divide n?
...       factors.append(d)  # if so, append d to list of factors
...       n = n/d            # then divide n by d
...     else:                # otherwise,
...       d = d + 1          # increase the trial divisor by 1
...   return factors         # loop complete, return list of factors
...
```

Every function definition begins with `def`, followed by the name of the function and a parenthesized list of its arguments, its independent variables. This is the line `def factor1(n)`. Most function definitions end with a statement of the form `return <foo>`, where `<foo>` is the value computed from the arguments. In our case the last statement is `return factors`. The intent is to return a list of prime factors of `n`. To do this we set the trial divisor `d` to `2`, and we set the variable `factors` to the empty list `[ ]`. The main part of the code is the `while` loop. So long as `n > 1`, certain actions are performed. First, if `n` is divisible by `d` the trial divisor `d` is appended to the list of factors. Then `n` is divided by `d` and the quotient stored in `n`. If `n` is not divisible by `d`, the trial divisor is increased by `1`.

Note the consistent use of indentation (two spaces per level) to exhibit the structure of the code, and note the use of colons after statements that begin a new level.

The algorithm used to design `factor1` can be improved considerably. First, we can divide `n` by as many `2`'s as possible, then do trial division by successive odd numbers beginning with `3`. This should speed up factorization by a factor of two. The code below illustrates this improvement.

```
def factor2(n):
    d = 2
    factors = [ ]
    while n % d == 0:
      factors.append(d)
      n = n/d
    d = 3
    while n > 1:
      if n % d == 0:
        factors.append(d)
        n = n/d
      else:
        d = d + 2
    return factors
```

But one can do far better still. If n has a factor, it has a factor $d \leq \sqrt{n}$. You should prove this for yourself. As a result, trial division can stop as soon as $d^2 > n$. The code below illustrates this improvement.

```
  def factor3(n):
    d = 2
    factors = [ ]
    while n % d == 0:
      factors.append(d)
      n = n/d
    d = 3
    while n > 1 and d*d <= n:
      if n % d == 0:
        factors.append(d)
        n = n/d
      else:
        d = d + 2
    if n > 1:
      factors.append(n)
    return factors
```

You should think about why the code fragment

```
    if n > 1:
      factors.append(n)
```

is necessary.

Let us digress for a moment on the running time of algorithms, which we measure in seconds or some proportional quantity such as years. Inputs are measured by their information content, that is, by the number of *bits* needed to specify them. The number of bits needed to specify a positive integer n is just the number of binary digits needed to express it. Thus 123 in binary is 111011, so it is a 6-bit integer. In general the number of bits in n is about $B = \log_2 n$.

Most of the time in running the various versions of the factoring code above is spent in trial division. Thus a rough estimate of the running time T_1 for `factor1` is

$$T_1 \sim \text{ number of trial divisions} \times \text{ time per division.}$$

The amount of time needed to divide (or multiply) B-bit integers is proportional to B^2. The number of divisions required in `factor1` is about n in the worst case. Thus

$$T_1 \sim CB^2 n,$$

where "$\sim$" means "roughly equal to." Since $n = e^{kB}$ with $k = \log 2$, where the logarithm is the natural one this can be written as

$$T_1 \sim CB^2 e^{kB},$$

where $C > 0$ is a constant. In the second version we must do about half as many trial divisions, so

$$T_2 \sim CB^2(n/2) = (CB^2/2)e^{kB}$$

The third version is much better: at most $\sqrt{n}$ trial divisions are required, so

$$T_3 \sim CB^2\sqrt{n} = CB^2 e^{(k/2)B}.$$

To better interpret these running time estimates, use the inequality $x^2 < e^x$ for $x \geq 0$ to obtain

$$T_1 \sim Ce^{(k+1)B} \tag{1}$$
$$T_2 \sim (C/2)e^{(k+1)B} \tag{2}$$
$$T_3 \sim Ce^{((k+1)/2)B} \tag{3}$$

Thus all versions of our factoring algorithms have so-called *exponential running times.*

Reducing the coefficient of B in the exponential is much more significant than reducing the constant C. This we know by pure thought, and our theory is confirmed by the data below. Times are in seconds, run on a 2002-vintage laptop.

N	factor1	factor2	factor3	factorization
12345678	0.137	0.056	0.00066	[2, 3, 3, 47, 14593]
123456789	0.031	0.014	0.0176	[3, 3, 3607, 3803]
123456787	1.448	0.628	0.00178	[31, 31, 128467]
1234567898	123.9	53.9	0.0156	[2, 61, 10119409]

We will show in section 5 how to produce timing data like that displayed in the above table. In any case, there is clearly much to think about. In particular, can we factor numbers with hundreds of digits?

The estimates just given are *upper bound* on the running times. Any algorithm can be analyzed to give an upper bound on the computational complexity (number of bit operations) needed to compute a quantity. A better algorithm gives a better upper bound. It is much harder to get lower bounds on the computational complexity. However, one lower bound is clear. The number of bit operations must be at least equal to the sum of the number of bits in the input and the output. This is the case of instantaneous computation. Thus a lower bound on the running time is CB. In this case we say that the running time is *linear* in the input size. Many algorithms, e.g., long multiplication of positive integers, have polynomial running times: multiplication of two B-bit integers runs in time CB^2. Thus there is room for improvement, and in fact there are faster, though very complicated algorithms.

Exercise 5 *In light of the above discussion, estimate how long it would take to factor numbers with one or two hundred digits.*

Exercise 6 *Devise a function* `isprime(n)` *which returns* `1` *if* `n` *is prime,* `0` *if it is composite.*

Exercise 7 *Devise a function* `gcd(a,b)` *which computes the greatest common divisor of positive integers* `a` *and* `b`*. Use the Euclidean algorithm. Estimate the running time of* `gcd`*.*

Note. *Try both iterative and recursive implementations of* `gcd`*.*

4 Loops and conditionals

As noted earlier, loops and conditional statements are the core of all machine computations. Let us therefore study these constructs more closely. To begin, we consider the problem of computing partial sums of infinite series, e.g., the harmonic series. Thus let

$$h(n) = 1 + 1/2 + \cdots + 1/n$$

and consider the corresponding Python code.

```
>>> def h(n):
...   sum = 0
...   for k in range(1,n+1):
...     sum = sum + 1.0/k
...   return sum
```

```
...
>>> h(10)
2.9289682539682538
>>> h(100)
5.1873775176396206
```

The expression `range(n+1)` creates the list [1, 2, 3, .., n] — the list of integers k such that $1 \leq k < n+1$. To compute $h(n)$ we first set the variable `sum` to 0. Then, for each `k` in the given range, we compute `sum + 1.0/k`, and store the result in `sum`. Finally, we `return` the result as the value of `h`.

The computation in the preceding example is driven by a `for-in` loop. The *loop body* — the set of actions to be repeated `n` times — is indented by a consistent amount, in this case two spaces. The loop body, which in this case is the single statement `sum = sum + 1.0/k` can be a whole paragraph, or block, possibly with many levels of indentation. The example below, which computes partial sums of the alternating harmonic series

$$g(n) = 1 - 1/2 + 1/3 - 1/4 \pm \cdots$$

illustrates this.

```
>>> def g(n):
...   sum = 0
...   for k in range(1,n+1):
...     if k % 2 == 1:
...       sum = sum + 1.0/k
...     else:
...       sum = sum - 1.0/k
...   return sum
>>> g(10)
0.64563492063492067
>>> g(100)
0.68817217931019503
```

Note again the consistent, linguistically significant use of indentation, of which there are now three levels. The `if-then-else` statement determines whether the current term is to be added to or subtracted from the running sum. Test for equality is always expressed by `==` . Assignment of a value to a variable uses `=` .

The `else` clause in a conditional statement is optional, and one can also have optional `elif` clauses to express more than two choices, as in the example below.

```
if <condition 1>:
  <action 1>
```

```
elif <condition 2>:
  <action 2>
else:
  <action 3>
```

Constructions like this can be as long as you please.

In addition to the `for-in` loop, there is the `while` loop, already used in the code for `factor`. We could have just as easily used it for the previous problem:

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

Here the variable `k` acts as the loop counter. If you forget the statement `k = k + 1` which increments the counter, then the loop is infinite and the code will run forever. To terminate a loop which has run too long, type control-C.

Exercise 8 *Define a function* `sum(f,a,b)` *which returns the sum*

```
f(a) + f(a+1) + ... + f(b)
```

where `a <= b` *and where* `f` *is an arbitrary function of one variable. Test your definition of* `sum` *using*

```
>>> sum( lambda n: n, 1, 10 )
```

and related expressions. Then calculate the sum

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

to an accuracy which you find satisfactory. Is it possible to find an exact answer for this infinite sum? Consider also the alternating sum

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

Exercise 9 *Define a function* `search(f, x1, x2, y1, y2)` *which prints a list of all solutions to* `f(x,y) == 0` *for integer vectors* `(x,y)` *where* `x1 <= x <= y`, `y1 <= y <= y2`. *Here* `f` *is an arbitray function from pairs of integers* `(x,y)` *to the integers, e.g.,* `pell2 = lambda x,y:x*x - 2*x*y - 1`. *This is the function we would use to study Pell's equation*

$$x^2 - 2y^2 = 1.$$

Use your code to find solutions to Pell's equation. A typical call to the function `f` *would be* `search( pell2, 0, 1000, 0, 1000)`. *What is the running time of this algorithm? Can one do better?*

Exercise 10 *Let f be a function from a set S to itself. Let x_0 be an element of S. Define a sequence $\{x_n\}$ by the rule $x_{n+1} = f(x_n)$. Define the* orbit *of x_0 under f to be the shortest non-repeating sequence $\{x_0, x_1, \ldots, x_n\}$.*

Devise a function `orbit(f,a)` *to return a list representing the orbit of* `a` *under* `f`. *Use it to study orbits of $f(x) = a * x$ modulo 101 for various a. What are their lengths? What other questions do your findings suggest? Pursue these questions and report on them. Be alert to ways of efficiently automating the computations your questions suggest.*

Hints and possible ingredients: *Use a* list *to build up the orbit, beginning with the empty list* `[ ]`. *To check whether an element is already in the list, use an expression of the form* `if x not in L:`

Exercise 11 *Devise an iterative definition of the function* `factorial(n)`. *In other words, use a loop, not recursion.*

5 Files

It is convenient to separate the functions of running and editing Python code. To do this, use a text editor such as emacs to store and modify the code. As an example, we create a file `factor.py` for factoring positive integers.

```
# file: factor.py

def factor3(n):
  """factor3(n) returns a list of the prime factors of n"""
  d = 2
  factors = [ ]
  while n % d == 0:
    factors.append(d)
    n = n/d
  d = 3
  while n > 1 and d*d <= n:
    if n % d == 0:
      factors.append(d)
      n = n/d    else:
      d = d + 2
  if n > 1:    factors.append(n)
  return factors
```

Then we can do the following:

```
student% python
>>> from factor import factor3
>>> f = factor3
>>> f(1234567)
[127, 9721]
>>> f(123)
[3, 41]
>>> f(123123123)
[3, 3, 41, 333667]
>>> f(123123123123123)
[3, 31, 41, 41, 271, 2906161L]
```

The `import` command reads in all of the definitions made in the file `factor.py` — just one in this case — so that they can be used as if the user had typed them in. To cut down on typing we have assigned `factor3` to the variable `f`. To see what the line after `def` containing the triple quotes does, try this:

```
>>> print factor3.__doc__
factor3(n) returns a list of the prime factors of n
```

It is good practice to document functions at they same time they are written. Documentation strings can contain more than one line.

Let us look at some other ways to use files. Consider the test file below, which is used to print factorizations of the twenty numbers 1000 through 1019:

```
# file = test.py
from factor import factor3

def test(n):
  print "   ", n, "==>", factor3(n)

print
for k in range(1000,1020):
  test(k)
print
```

We can then try this at the command line:

```
student% python test.py
```

The result is the output below:

```
1000 ==> [2, 2, 2, 5, 5, 5]
1001 ==> [7, 11, 13]
1002 ==> [2, 3, 167]
1003 ==> [17, 59]
1004 ==> [2, 2, 251]
1005 ==> [3, 5, 67]
1006 ==> [2, 503]
1007 ==> [19, 53]
1008 ==> [2, 2, 2, 2, 3, 3, 7]
1009 ==> [1009]
1010 ==> [2, 5, 101]
1011 ==> [3, 337]
1012 ==> [2, 2, 11, 23]
1013 ==> [1013]
1014 ==> [2, 3, 13, 13]
1015 ==> [5, 7, 29]
1016 ==> [2, 2, 2, 127]
1017 ==> [3, 3, 113]
1018 ==> [2, 509]
1019 ==> [1019]
```

It would be still more convenient to be able to issue commands like `python test.py 1000 20` to construct tables like the one above: a table of factorizations of twenty numbers starting with 1000. That way we don't have to edit the code to do a new example. The version of `test.py` displayed below shows how to do this. Briefly, we import the `sys` module which gives access to arguments like 1000 and 20 that are typed on the command line. This is done through the list of arguments `sys.argv`. The arguments — the list elements — are strings, and so have to be converted to long integers using the `long` function.

```
# file = test.py
import sys
from factor import factor3

def test(n):
  print  "  ", n, "==>", factor3(n)
```

```
a = long( sys.argv[1] )
b = long( sys.argv[2] )
print
k = a
while k < a + b:
  test(k)
  k = k + 1
print
```

The name `sys.argv` consists of two parts: the module `sys` named in the `import` command and the name `argv` defined in that module. Contrast this with the names from the module `factor`: we imported `factor3` but no others using `from factor import factor3`. Note that it is the file `factor.py` from which `factor3` is imported.

As a final illustration, we show how to create a system-level command `factor` which prints a table of factorizations. The idea is to be able to say

```
student% factor 1000 20
```

at the command line. To do this, first add the line

```
#! /usr/bin/env python
```

to the head of the file. (It must be the first line). Then type the following:

```
mv test.py factor            # change name of file to "factor"
chmod u+x factor             # make the file executable
mv factor ~/bin/scripts      # put it in standard directory
mv factor.py ~/bin/scripts   # put factor.py there too
rehash                       # update list of commands
```

You have set up command called `factor` in your `bin/scripts` directory. Here we assume that this directory exists and is in your search path.

Timings

Let us conclude this section with a discussion of how to time the execution of Python code. To this end, let us imagine that the definitions of the three factoring methods discussed in section 3 reside in a file `factormethods.py` under the names `factor1`, `factor2`, and `factor3`. Then the code in file `file = timings.py` below.

```
# file = timings.py
```

```
import sys, time
from factormethods import factor, factor2, factor3

n = long( sys.argv[1] )

a = time.time()
result = factor1(n)
b = time.time()
print "1: ", b - a

a = time.time()
result = factor2(n)
b = time.time()
print "2: ", b - a

a = time.time()
result = factor3(n)
b = time.time()
print "3: ", b - a

print n, "==>", result
```

The function `time` imported from module `time` returns the time in seconds of a system clock.

Exercise 12 *Put one of your previous programs into a file and try the various ways of executing it described above.*

Exercise 13 *Improve the command* `factor` *so that (1) if it takes no arguments, a message describing its usage is printed; (2) if it takes one argument, e.g.* `factor 1234567` *it prints the factorization of that number, (3) if it takes two arguments, e.g.,* `factor 1234567 10`*, it prints a table, as before.*

Exercise 14 *Devise a function* `doc` *such that* `doc(f)` *displays the documentation for* `f`*.*

6 Case study: the Fermat test

In section 3 we saw how trial division could be implemented in Python. The resulting code for factoring an integer runs in exponential time relative to the size in bits of the number to be factored. This code also gives an exponential time algorithm for determining whether a number is prime:

```
isprime = lambda n: len(factor3(n)) == 1
```

The function `isprime` returns `1` if the argument is prime, `0` if it is composite. This is because the value of an expression of the form `<a> == <b>` is `1` if `<a>` and `<b>` are equal, and is `0` otherwise. Such expressions are called *Boolean*. Here is a short test of `isprime`:

```
>>> for n in range(2,8):
...   print n, isprime(n)
...
2 1
3 1
4 0
5 1
6 0
7 1
```

We can also use `isprime` to make lists of primes:

```
>>> filter( isprime, range(2,100) )
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
>>> len(_)
25
# there are 25 primes < 100
```

The value of an expression of the form `filter(f, L)` is a new list whose elements are the elements `x` of `L` such that `f(x) == 1`. You should take the above example as an illustration of Python's `filter` function. A much better way of creating lists of primes is to use the sieve of Eratosthenes (c. 276 - c. 194 BC).

We will now study a way of testing whether a number is composite that is much, much faster, so long as it is implemented efficiently. The idea is to use a theorem of Fermat:

Theorem 1 *If p is a prime and a is not divisible by p, then $a^{p-1} \equiv 1 \bmod p$.*

Thus we could define a function `ft` (for Fermat test) as follows:

```
>>> ft = lambda n: 2**(n-1) % n == 1
```

One could then test `ft` as follows:

```
>>> filter( ft, range(2,100) )
[3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
>>> len(_)
24
# 24 numbers < 100 pass the Fermat test
```

It seems as though the Fermat test is pretty good: the numbers that passed the test are all prime. Also, while 2 failed the test, this failure is consistent with the theorem, which applies to primes other than 2. Nonetheless, one should also be careful: the theorem is really a test for compositeness, not for primality. (Numbers which pass the Fermat test have been called "industrial-grade primes.") In light of these comments, you should do the next exercise before proceeding:

Exercise 15 *Are there any numbers* $n < 1000$ *which pass the Fermat test, but which are composite? Said in other words: what is the defect rate for production of industrial grade primes?*

Let us now turn to the question of implementing the Fermat test in an efficient way — one so efficient that we can use it to demonstrate that very large numbers are composite. For example using a good implementation, we find, essentially instantaneously, that

```
>>> ft(367129485367127143774839380041017394964020200854210369)
0
```

The given number is therefore composite. Yet the factorization methods studied so far fail to find prime factors in a time bounded by this author's patience.

For an efficient implementation, we use successive squaring with reduction modulo n to compute expressions of the form $a^k \bmod n$. We illustrate the method for 3^{100} modulo 101. First compute the binary expansion of 100 by repeated division by two. The work is recorded in the table below:

```
quotient  remainder
--------  ---------
     100          0
      50          0
      25          1
      12          0
       6          0
       3          1
       1          1
       0
```

Thus the binary expansion of 100 is 1100100. Thererefore

$$100 = 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

Thus

$$3^{100} = 3^{2^6} \times 3^{2^5} \times 3^{2^2}.$$

Observe that

$$3^{2^{n+1}} = (3^{2^n})^2$$

Thus the powers of three that we need for the computation can be obtained by repeatedly squaring the number 3. If we reduce modulo 101 at each stage, the numbers stay under control:

```
  q  r    s
-----------
100  0    3
 50  0    9
 25  1   81
 12  0   97
  6  0   16
  3  1   54
  1  1   88
```

The columns `q, r, s` stand for `quotient`, `remainder`, and (repeated) `square`, where the squares are taken mod 101. Now we multiply together the repeated squares associated with binary digit 1:

$$3^{100} = 3^{2^6} \times 3^{2^5} \times 3^{2^2} \equiv 88 \times 54 \times 81 \text{ modulo } 101.$$

The result is

$$88 \times 54 \times 81 = 384912,$$

which taken modulo 101 is 1. The table, by the way, was generated using Python:

```
>>> a = 100
>>> s = 3
>>> for k in range(0,7):
...   print a, a % 2, k, 2**k, s
...   a = a/2
...   s = s*s % 101
...
```

Note that no numbers larger than 100^2 appear in the course of the computation.

It is easy to translate the procedure just described into a Python function:

```
def modpower(b,e,n):
  result = 1
  s = b
  q = e
  while q > 0:
    r = q % 2
    if r == 1:
      result = result*s % n
    # print q, r, s, result
    s = s*s % n
    q = q/2
  return result
```

With `modpower` in hand, we can redesign the Fermat test:

```
ft = lambda n: modpower(2,n-1,n) == 1
```

Let us analyze the running time of the Fermat test implemented using `modpower`. The number of multiplications and divisions required for computation of `modpower(b,e,n)` is at most four times the number of binary digits in `e`. As noted earlier, the time required to multiply two numbers is bounded by some constant times the product of the number of binary digits needed to represent the numbers. Assuming $b < n$, the running time T for `modpower` is therefore

$$T \sim C \times (\text{binary digits of } e) \times (\text{binary digits of } n)^2.$$

For the Fermat test `ft(n)`, $e < n$, so

$$T \sim C \times (\text{binary digits of } n)^3.$$

Thus the running time is bounded by a polynomial in the number of bits of `n`. Polynomial-time algorithms scale up much better than do exponential-time algorithms. To illustrate this, consider two algorithms with running times

$$f(B) = .001B^3 \qquad g(B) = e^{-1}e^{B/10}$$

The table below gives running times in seconds, where $3e6 = 3 \times 10^6$, etc.:

B	f	g
10	1	1
20	8	3
40	64	20
80	512	1097
160	4096	3e6
320	32,768	3e14

Algorithms f and g have the same running time for a 10-bit input. As the input is doubled to 20 and then 40 bits, algorithm g performs several times better than algorithm f. At 80 bits algorithm f runs in 8.5 minutes, about half the running time of g. But at the next doubling, 160 bits, the outcome is vastly different. Algorithm f takes about 1.1 hours whereas g takes 5.5 weeks. One more doubling renders g impractical: it takes a million years as opposed to just 9 hours.

Exercise 16 *Consider the 73-digit number*

```
1047110956785547244551392396724393823
3977774546307981396782501853835898079
```

Is it composite? Explain.

Exercise 17 *Find an industrial prime larger than* 10^{80}.

Exercise 18 *A composite number which passes the Fermat test is called a* pseudoprime *(or a 2-pseudoprime). Investigate the number of pseudoprimes among the industrial primes.*

Exercise 19 *Devise a function* `sieve(n)` *that returns a list of primes* $p \leq n$. *Use the sieve of Eratosthenes.*

7 Problems

Below are more problems to work on.

Exercise 20 *Develop a program for listing (printing out) all the positive integer solutions to the quadratic Diophantine equation* $x^2 + y^2 = z^2$, *where* x, y, *and* z *less than some fixed bound* C. *After doing this, modify your code so that only* primitive *triples are printed out. These are triples where* x, y, *and* z *have no common factors.*

Exercise 21 *Design a function* `isolve(a,b,c)` *which returns a solution to the linear Diophantine equation* $ax + by = c$ *if it exists.*

Exercise 22 *Design a function which returns the inverse of a modulo* p, *a prime.*

Exercise 23 *Find a way of manufacturing large (probable) primes. Can you do better than in* `ft(n)`*?*

In the problems below an elliptic curve E modulo p can be represented by a list $[a, b, p]$, where the curve is given in Weierstrass form by $y^2 = x^3 + ax + b$. A point on the curve is represented by a list $[x, y]$. Think about how to represent the point at infinity.

Exercise 24 *Develop a program for counting all the points on an elliptic curve* $y^2 = x^3 + ax + b$ *modulo a prime* p.

Exercise 25 *Design a function which takes an elliptic curve mod p as input and produces a point on the curve as output. How does your code perform when p is very, very large?*

Exercise 26 *Design a function for adding and doubling points on an elliptic curve mod p.*

Exercise 27 *Design a function for generating a random elliptic curve modulo a random prime with a known rational point.*

References

[1] Donald Knuth, Seminumerical Algorithms, *The Art of Computer Programming, vol. 2*, Addison-Wesley, 1981, pp 688.

[2] Neal Koblitz, A Course in Number Theory and Cryptography, Springer-Verlag, 1994.

[3] Guido van Rossum, Python Tutorial, python.org/dev/doc/devel/tut/tut.html

[4] Joseph Silverman and John Tate, *Rational Points on Elliptic Curves*

File = QuickPython.tex, March 22, 2002