Factoring with Python – continued
Spring 2015

Our goal is write a python program to factor integers. This document and the first few classes model the software development process. We build a complex program one step at a time, documenting the process and keeping track of the steps along the way.

2 Arranging user input

One problem with `factor1.py` is that I have to edit it in order to factor a different number. `factor2.py` asks user for input, after suggesting an interesting number to test:

```
> python factor2.py
test with 10967535067
number, please: 10967535067
smallest factor of 10967535067 is 104723
computation took 0.11700701713562012 seconds
```

I also added some code to *time* the algorithm. It took just over a tenth of a second on my PC for python to find that the smallest factor of the 11 digit number 10967535067 is the six digit prime 104723. In this case that number is nearly the square root, so the **while** loop had to execute more than 100,000 times. A tenth of a second is a lot of computer time. Soon I will want to tinker with the algorithm to see if I can get it to run faster.

Here's how I asked for input in `factor2.py`

```
11 import math
12 import time
13
14 print("test with 10967535067")
15 # The input function writes a prompt and reads a string from the terminal
16 # The int function converts that string to an integer
17 number = int(input('number, please: '))
```

and here's the timing code surrounding the real work. Only lines 21-22 and 35-36 are new.

```
21 # record the current time
22 start = time.time()
23 possible_factor = 2
24 while (possible_factor <= sqrt):
25     remainder = number % possible_factor
26     if remainder == 0:
27         break
28     possible_factor = possible_factor + 1
29
30 if (possible_factor > sqrt):
31     smallest_factor = number # we have a prime number
32 else:
33     smallest_factor = possible_factor
34
35 # done, so calculate elapsed time
36 elapsed = time.time() - start
```

3 Functions – making code modular

In this next version of the factor program I will separate the algorithm from the input and output code, so I can experiment with different factoring algorithms to find a faster one.

We know about some of the builtin python functions – **print()**, **str()**, **input()**, **int()** and `exit()`. Now we'll see how to create a function of your own. The python keyword is **def**.

I tested repeatedly to check that this program behaves just the way the last one did. That was easy since I'd arranged for prompted input, so that I didn't have to edit the program each time.

I made some small improvements in the loop body, too, so that it begins to look more like idiomatic python than beginner's python:

- I start out assuming that the number I want to factor is a prime. Then I don't need to test after the loop to see whether I've gone as far as the square root.
- I no longer need the variable `remainder` since I test `(number % possible_factor == 0)` directly.
- I use the neat *increment* operator `+= 1` to add 1 to `possible_factor`.

```
# factor3.py
#
# get an integer from the user,
# call find_smallest_factor to find its smallest factor
# time how long the program takes
#
# no error handling if the user doesn't enter a number
#
# Ethan Bolker
# January 15, 2015

import math
import time

# def defines a function( with arguments )
# indent the function body
# stuff in quotes at start of body is built in documentation
def find_smallest_factor(number):
    """ naive search for smallest factor of input

     loop over possible factors 2,3,4,..., sqrt
        if you find a 0 remainder you have a factor
     if you haven't found a factor by now, you have a prime
    """
    sqrt = math.sqrt(number)
    answer = number # start out assuming prime

    # loop body a little more idiomatic than in last version
    possible_factor = 2
    while (possible_factor <= sqrt):
        if (number % possible_factor == 0):
            answer = possible_factor # not prime!
            break
        possible_factor += 1 # add 1 to a variable
    # the return keyword tells the caller what this function calculated
    return answer

# code to execute begins here
number = int(input('number, please: '))

start = time.time()
# give smallest_factor the value returned by  the function
smallest_factor = find_smallest_factor(number)
elapsed = time.time() - start

print ("smallest factor of " + str(number) + " is " + str(smallest_factor))
print( "computation took " + str(elapsed)+ " seconds" )
```

4 Modules – making code even more modular

In my algorithm's **while** loop I increment the possible factor by 1 each time. But I know that's inefficient. If I just test first to see if the number is even, then I can loop just over the odd numbers 3, 5, 7, ... That should make the algorithm twice as fast for big numbers. If I also test for divisibility by 3 then I can loop over 5, 7, 11, 13, 17, 19, 23, 25, ... (alternately adding 2 and 4). That should make the algorithm three times as fast as the naive one (for big numbers) since I only look at two out of every six possible factors.

I will put the naive algorithm in a function called `fsf0` and the faster one in function `fsf1`. Then I will put those two functions in a file by themselves – a python `module` I'll call `fsf.py`.

When you read the body of `fsf1` you will see that it uses **return** statements in the middle of the function, not just at the end. Some people think this is not a good thing, since it may make it harder for a reader to figure out what the function is doing.

Here is the module:

```
# fsf.py
#
# functions to find the smallest factor of an integer
# Ethan Bolker
# January 13, 2015

import math

def fsf0(number):
    """ naive search for smallest factor of input

     loop over possible factors 2,3,4,..., sqrt
        if you find a 0 remainder you have a factor
     if you haven't found a factor by now, you have a prime
    """
    sqrt = math.sqrt(number)
    answer = number # start out assuming prime

    possible_factor = 2
    while (possible_factor <= sqrt):
        if (number % possible_factor == 0):
            answer = possible_factor # not prime!
            break
        possible_factor += 1 # add 1 to a variable
    return answer

# This algorithm should be 3 times as fast as the naive one,
# since it loops only over possible divisors congruent to
# 1 or 5 mod 6.
def fsf1(number):
    """ better than naive search for smallest factor of input

     if number is even, found factor 2
     if 3 divides number, found factor 3
     loop over possible factors 5,7,11,13,17,19,..., sqrt
        if you find a 0 remainder you have a factor
     if you haven't found a factor by now, you have a prime
    """
    if (number%2 == 0):
        return 2
    if (number%3 == 0):
        return 3
    sqrt = math.sqrt(number)

    possible_factor = 5
    while (possible_factor <= sqrt):
```

```
        if (number % possible_factor == 0):
            return possible_factor # all done — don't bother with break
        possible_factor += 2 # next value mod 6
        if (number % possible_factor == 0):
            return possible_factor
        possible_factor += 4 # next value mod 6
    return number # started with a prime
```

The program that *calls* the functions in that module is

```
# factor4.py
#
# separate the mathematics from input/output interface
#
# get an integer from the user,
# call imported functions fsf0, fsf1 to find its smallest factor
# compare timings
#
# Ethan Bolker
# January 15, 2015

import math
import time
import fsf # module with several functions to find smallest factor

print("test with 10967535067")
number = int(input('number, please: '))

start = time.time()
smallest_factor = fsf.fsf0(number)
elapsed = time.time() - start

print ("smallest factor of " + str(number) + " is " + str(smallest_factor))
print( "fsf0 computation took " + str(elapsed)+ " seconds" )

start = time.time()
smallest_factor = fsf.fsf1(number)
elapsed = time.time() - start

print ("smallest factor of " + str(number) + " is " + str(smallest_factor))
print( "fsf1 computation took " + str(elapsed)+ " seconds" )
```

Here is the output:

```
$ python factor4.py
test with 10967535067
number, please: 10967535067
smallest factor of 10967535067 is 104723
fsf0 computation took 0.10400605201721191 seconds
smallest factor of 10967535067 is 104723
fsf1 computation took 0.029001951217651367 seconds
```

I can see that `fsf1` is indeed about three times as fast as `fsf0`, as I predicted.

Here is the LaTeX source for this document. You can cut it from the pdf and use it to start your answers. I used the `\jobname` *macro for the source file name, so you can call your file by any name you like.*

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
\documentclass[10pt]{article}
\usepackage[textheight=10in]{geometry}

\usepackage{verbatim}
\usepackage{amsmath}
\usepackage{amsfonts} % to get \mathbb letters

\usepackage[utf8]{inputenc}
\DeclareFixedFont{\ttb}{T1}{txtt}{bx}{n}{9} % for bold
\DeclareFixedFont{\ttm}{T1}{txtt}{m}{n}{9}  % for normal
% Defining colors
\usepackage{color}
\definecolor{deepblue}{rgb}{0,0,0.5}
\definecolor{deepred}{rgb}{0.6,0,0}
\definecolor{deepgreen}{rgb}{0,0.5,0}

\usepackage{listings}

%Python style from
%http://tex.stackexchange.com/questions/199375/problem-with-listings-package-for-python-syntax-colo
\newcommand\pythonstyle{\lstset{
  language=Python,
  backgroundcolor=\color{white}, %%%%%%%
  basicstyle=\ttm,
  otherkeywords={self},
  keywordstyle=\ttb\color{deepblue},
  emph={MyClass,__init__},
  emphstyle=\ttb\color{deepred},
  stringstyle=\color{deepgreen},
  commentstyle=\color{red},  %%%%%%%%
  frame=tb,
  showstringspaces=false,
  numbers=left,numberstyle=\tiny,numbersep =5pt
}}
\usepackage{hyperref}

\begin{document}

\pythonstyle{}
\setcounter{section}{1} % start with section 2
\begin{center}
\Large{
Factoring with Python -- continued \\
Spring 2015
}
\end{center}

Our goal is write a python program to factor integers. This document
and the first few classes model the software development process. We
build a complex program one step at a time, documenting the process
and keeping track of the steps along the way.

\section{Arranging user input}

One problem with
```

\lstinline!factor1.py! is that I have to edit it in order to
factor a different number.
\lstinline!factor2.py! asks user for input, after suggesting
an interesting number to test:

\begin{verbatim}
> python factor2.py
test with 10967535067
number, please: 10967535067
smallest factor of 10967535067 is 104723
computation took 0.11700701713562012 seconds
\end{verbatim}

I also added some code to \emph{time} the algorithm. It took
just over a tenth of a second on my PC for python to find that the
smallest factor of the 11 digit number 10967535067
is the six digit prime 104723. In this case that number is nearly the
square root, so the \lstinline!while! loop had to execute more than
100,000 times. A tenth of a second is a lot of computer time. Soon I
will want to tinker with the algorithm to see if I can get it to run faster.

Here's how I asked for input in \lstinline!factor2.py!

\lstinputlisting[firstnumber=11, firstline=11,lastline=17]{factor2.py}
%\lstinputlisting[firstline=11,lastline=17]{factor2.py}

and here's the timing code surrounding the real work. Only lines 21-22
and 35-36 are new.

\lstinputlisting[firstnumber=21,firstline=21,lastline=36]{factor2.py}

\section{Functions -- making code modular}

In this next version of the factor program I will separate the
algorithm from the input and output code, so I can experiment
with different factoring algorithms to find a faster one.

We know about some of the
builtin python functions -- \lstinline!print()!,
\lstinline!str()!,
\lstinline!input()!,
\lstinline!int()!
and \lstinline!exit()!.
Now we'll see how to create a function of your own.
The python keyword is \lstinline!def!.

I tested repeatedly to check that this program behaves just the way
the last one did. That was easy since I'd arranged for prompted input,
so that I didn't have to edit the program each time.

I made some small improvements in the loop body, too, so that it
begins to look more like idiomatic python than beginner's python:
\begin{itemize}

\item I start out assuming that the number I want to factor is a
  prime. Then I don't need to test after the loop to see whether I've
  gone as far as the square root.

\item I no longer need the variable \lstinline!remainder! since I test
\lstinline!(number % possible_factor == 0)! directly.

```
\item I use the neat  \emph{increment} operator \lstinline!+= 1! to
  add 1 to \lstinline!possible_factor!.
\end{itemize}

\lstinputlisting{factor3.py}

\section{Modules -- making code even more modular}

In my algorithm's \lstinline!while! loop I increment the possible
factor by 1 each time. But I know that's inefficient. If I just test
first to see if the number is even, then I can loop just over the odd
numbers 3, 5, 7, \ldots. That should make the algorithm twice as fast
for big numbers. If I also test for divisibility by 3 then I can loop
over 5, 7, 11, 13, 17, 19, 23, 25, \ldots (alternately adding 2 and
4). That should make the algorithm three times as fast as the naive
one (for big numbers) since I only look at two out of every six
possible factors.

I will put the naive algorithm in a function called
\lstinline!fsf0!
and the faster one in function
\lstinline!fsf1!.
Then I will put those two functions in a file by themselves -- a
python
\lstinline!module! I'll call
\lstinline!fsf.py!.

When you read the body of
\lstinline!fsf1! you will see that
it uses \lstinline!return! statements in the middle of the function,
not just at the end. Some people think this is not a good
thing, since it may make it harder for a reader to figure out
what the function is doing.

Here is the module:

\lstinputlisting{fsf.py}

The program that \emph{calls} the functions in that module is

\lstinputlisting{factor4.py}

Here is the output:

\begin{verbatim}
$ python factor4.py
test with 10967535067
number, please: 10967535067
smallest factor of 10967535067 is 104723
fsf0 computation took 0.10400605201721191 seconds
smallest factor of 10967535067 is 104723
fsf1 computation took 0.029001951217651367 seconds
\end{verbatim}

I can see that \lstinline!fsf1!
is indeed about three times as fast as
\lstinline!fsf0!,
as I predicted.
```

```
\newpage
\emph{
Here is the \LaTeX{} source for this document. You can cut it from the
pdf and use it to start your answers. I used the} \verb!\jobname!
\emph{macro for the source file name, so you can call your file by any
  name you like.}
\verbatiminput{\jobname}

\end{document}
```