

Math with Python—An Introduction

E. Krishnan
e.krshnana@gmail.com

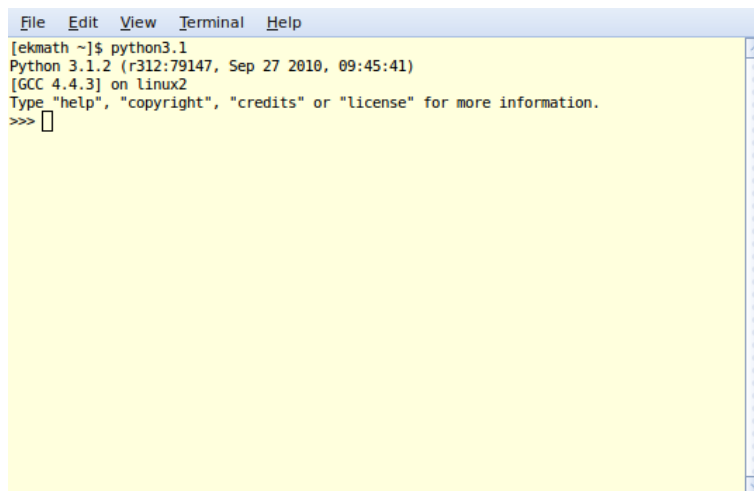
This is a very short introduction to the Python programming language, using Python 3.1. We will just explain how we invoke the Python interpreter in a Linux machine and how we can write and execute some simple programs in Python, and that too only those related to numbers.

1 Interactive Mode

The basic way to start using Python is to open a terminal (assuming we are using a GUI interface) and type

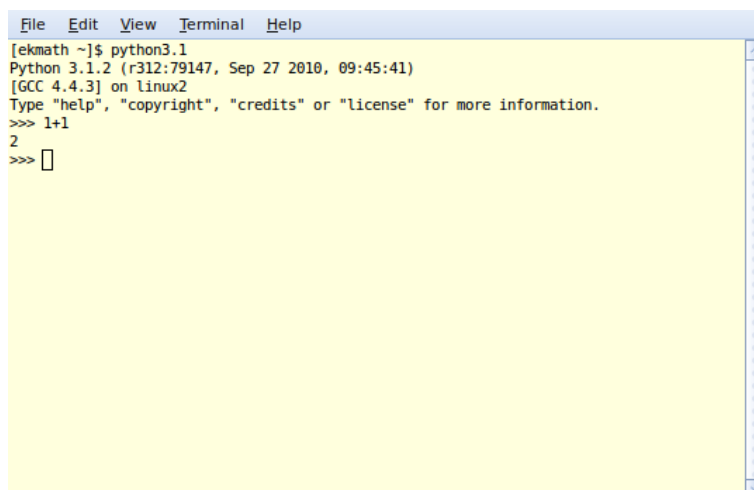
`python3`

The Python interpreter prints a message and gives us a *prompt*, where we are to type our first line of code:



```
File Edit View Terminal Help
[ekmath ~]$ python3.1
Python 3.1.2 (r312:79147, Sep 27 2010, 09:45:41)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

If we now type `1+1` and press enter, we get this:



```
File Edit View Terminal Help
[ekmath ~]$ python3.1
Python 3.1.2 (r312:79147, Sep 27 2010, 09:45:41)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 1+1
2
>>> 
```

In the following, we won't show anymore screen-shots, but simply give our **inputs** and the python **outputs**, distinguishing between typographically, as in this senetence.

1.1 Basic calculations

We can use Python as a simple calculator, for basic calculations involving the usual operataions of addition, subtraction, division and exponentiation:

```
>>> 2.5*2
5.0
>>> 14/5
2.8
>>> 1.5**2
2.25
```

Note that exponentiation is indicated as `1.5**2`, instead of `1.5^2`, as in many other programming languages. Something about division must also be mentioned. The operation `14/5` returns the *decimal* 2.8. We can get the quotient and remainder separately as integers like this:

```
>>> 14//5
2
>>> 14%5
4
```

We can compute roots using fractional exponents. For example to compute $\sqrt{3}$ and $\sqrt[3]{2}$, we proceed like this:

```
>>> 3**(1/2)
1.7320508075688772
>>> 2**(1/3)
1.2599210498948732
```

We can also use the function `pow` to do these computations:

```
>>> pow(3,1/2)
1.7320508075688772
>>> pow(2,1/3)
1.2599210498948732
```

There's also a `round` function to round numbers upto a desired decimal place

```
>>> 100000*(1.065)**3
120794.9625
>>> round(100000*(1.065)**3)
120795
>>> round(100000*(1.065)**3,2)
120794.96
```

Python also supports complex numbers: we write. for example, `2+3j` to denote the complex number $2 + 3i$. Thus:

```
>>> (2+3j)*(1-4j)
(14-5j)
>>> (0+1j)**(0+1j)
(0.20787957635076193+0j)
>>> abs(3+4j)
5.0
```

Note that complex exponentiation returns the principal value, so that

$$i^i = \sqrt{e^{-\pi}} \approx 0.20787957635076193$$

We also note in passing that Python has *unlimited integer precision*, so that we can get

```
>>> 2**1000
1071508607186267320948425049060001810561404811705533607
4437503883703510511249361224931983788156958581275946729
1755314682518714528569231404359845775746985748039345677
7482423098542107460506237114187795418215304647498358194
1267398767559165543946077062914571196477686542167660429
831652624386837205668069376
```

This ends our first Python session. To quit, either type `quit()` at the prompt or press the Ctrl and the d keys together.

1.2 Extension modules

When we invoke Python, it starts with a collection of commands and functions built into its core (called, surprisingly enough, *built-ins*). Thus the various math operations we have tried out are all built-ins. To have further functions, we will have to incorporate additional files kept in the Python *library*. These files, called *modules*, contain additional functionalities. For example, there is a **math** module, which contains the definitions for other math functions such as trigonometric and exponential functions. To use these, we will have to *import* the module:

```
>>> import math
>>> math.sin(1)
0.8414709848078965
```

Note that this is the value of the sine of 1 *radian* and not 1 degree. To convert degrees to radians, the **math** module has the function **radians**. Thus

```
>>> math.radians(1)
0.017453292519943295
>>> math.pi/180
0.017453292519943295
>>> math.sin(math.radians(1))
0.01745240643728351
```

Instead of typing `math.sin` and `math.radians` everytime, we can do

```
>>> from math import sin,radians
>>> sin(radians(1))
0.01745240643728351
```

To know about the other functions available in the **math** module, look up *The Python Library Reference*. If we want to use all the functions in this module without the **math** prefix, we do

```
>>> from math import *
```

and then we can find $\sqrt{e^{-\pi}}$, for example, by simply typing `exp(x)` the function **radians**. Thus

```
>>> sqrt(exp(-pi))
0.20787957635076193
```

The **fractions** modules allows us to work with fractions, instead of decimals:

```
>>> from fractions import Fraction
>>> Fraction(1,2)+Fraction(1,3)
Fraction(5,6)
>>> Fraction('1.25')
Fraction(5,4)
```

1.3 Variables

In math, when we need same computations on different numbers, we encode these in a formula, using letters instead of numbers. For example, to compute the money accrued by compound interest, we use the formula

$$a = p(1 + r)^n$$

and in actual practice, we will have to round this to the nearest rupee. For a specific amount, rate of interest and period, we can use a calculator to do the computation for us. If we want to do this for a large number of different values, even using a calculator may be tedious. Then we can turn to a computer. Let's see how we do this in Python:

```
>>> p = 10000
>>> r = 6
>>> n = 3
>>> a = round(p*(1+r/100)**n)
>>> a
11910
```

The first three lines assign the values 1000, 6, 3 to the three *variables* `p`, `n`, `r` and the fourth assigns the number $1000 \cdot (1 + 6/100)^3$ to the variable `a`.

To compute `a` for other values of `p`, `r`, `n`, we will have to go through this *entire* set of commands. (Note that simply changing the values of these, the value of `a` *does not* change. We will have to run the definition of `a` again.) Though the *history mechanism* (cycling through previous inputs using the up and down arrow-keys) is of some help here, it is somewhat clumsy.

There are at least two ways to get over this: one is to define a function, (which can be done interactively) or to use a file. Functions we will consider later. In the next section, we look at the second alternative.

2 Using files

In the interactive Python sessions, we type the instructions line by line directly to the Python interpreter. Instead, we can type the various instructions in a *text* file and then call Python to read it and act according to instructions in it. To do our last example this way, open a *text editor* (Emacs being my personal choice) and type the following lines:

```
# This program computes the total amount, given the principal,
# interest rate and number of years

p = int(input("Principal : "))
r = int(input("Rate of interest : "))
n = int(input("Years : "))
print(round(p*(1 + r/100)**n))
```

Save the file as `interest.py` (or any other name, but the extension must be `.py`). Now in a terminal type

```
python3.1 interest.py
```

The program prompts for Principal, Rate of Interest, Years and, upon receiving the input at each stage, outputs the amount as below:

```
Principal : 10000
Rate of interest : 6
Years : 3
11910
```

Now let's have a line-by-line look at this program. The first two lines of the program are not absolutely necessary; in fact Python does not read any line starting with the `#` (hash) symbol. They are just *comments* about the program. The third line does several things:

- (i) Display the `Principal :` in the terminal
- (ii) Accept whatever is typed and store it as an *integer*
- (iii) Assign the name `p` to this integer

To understand this better, call Python in interactive mode and run these commands:

```
>>> input("Principal : ")
Principal : 10000
'10000'
>>> int(input("Principal : "))
Principal : 10000
10000
```

This is an instance where we come across two of the different *data types* in Python. By default, what we type as a response to the `input` command is stored as a *string* in Python ('10000'); and the command `int` changes its type to an integer (10000). (we'll talk more about strings later).

The fourth and fifth lines similarly define the variables `r`, `n` and assign the input values to them. The last line computes the value of the specified expression and the `print` command displays it on the screen

In a Linux system, there is another way to run this program. Type this line at the very beginning of the program:

```
#!/usr/bin/python3.1
```

and then make the file executable by running the command

```
chmod u+x interest.py
```

Then you can run Python on it by simply typing

```
./interest
```

For this to work, the final command should be run from the directory containing the file. Also in this method, the file can be named simply `interest`, without any extension.

The first line, since it starts with `#`, is ignored by Python, but it is a special instruction to the Linux *shell* to find the Python interpreter `python3.1` residing in the directory `/usr/bin`

For the rest of the discussion, we assume that Python is used in this mode, so that we only describe programs, assuming it to be typed into a file and Python run on it, in either of the two ways mentioned above. In the program above, everytime a computation is finished, the program automatically quits. Wouldn't it be nice, if we can continue with other set of values, till we are ready to quit? This is what we attempt in the next section.

3 Loops

As in other computing languages, Python also has different kinds of *loops*, which mean programming constructs to repeat the execution of certain operations.

3.1 while loops

At the end of the last section, we noted that we can make a program to continue, until asked to quit. What we need is a mechanism to read user inputs, evaluate something using these and then print the result, this cycle repeating till asked to quit. Let's see how we can modify our compound-interest program to behave like this. (In modifying programs, it is always safer to copy it under another name, and then make changes in the copy. In this way, we will have something to fall back on, if something goes wrong.)

```

# This program computes the total amount, given the principal,
# interest rate and number of years and continues till asked to quit

option = (input("Press Enter to continue, q to quit " ))
while option != 'q':
    p = int(input("Principal : "))
    r = float(input("Rate of interest : "))
    n = int(input("Years : "))
    print(round(p*(1 + r/100)**n))
    option = (input("Press Enter to continue, q to quit " ))

```

Ignoring the comments, like Python does, the only changes are the two new lines at the beginning and a single line at the end. The first of these displays the text within the parantheses in the terminal and stores the input typed as a string under the name `option`. The next line introduces a new construct: the phrase

`while option!= 'q'`

is to be translated as

so lonng as the value of the variable string option is not equal to the string q

(Why quotes around `q` and none around `option`?). The remaining lines all are things to do so long as `option` \neq `q`. So, as long as we don't press the `q` key as a response to the option query, the program continues to ask for new inputs and evaluates new values; once we press `q` as the option, the `while` condition fails and the program terminates.

We must also note some important Python conventions here:

- (i) The `while` condition is on a line all by itself
- (ii) It ends in a colon (`:`)
- (iii) The remaining lines all are indented and by the same amount

The colon is to indicate the end of the loop *header* and the indentatations indicate *blocks*, instead of parantheses or braces as in other programming languages. (Intelligent text editors such as Emacs produce indentation automatically when we press the `Tab` key.)

Let's consider a more abstract example. By considering regular polygons of more and more sides within a circle, we can see that

$$\lim_{n \rightarrow \infty} n \sin \left(\frac{180^\circ}{n} \right) = \pi$$

To see how this approach takes place, we can compute this expression for larger and larger values of n :

```

# This program lists the values of n*sin(180/n)
# for n between 90 and 100

from math import sin, radians

n = 90
while 90 <= n <= 100:
    p = n*sin(radians(180/n))
    print(p)
    n = n+1

```

Running this program with Python gives

```
3.14095470323
3.14096864624
3.14098213711
3.14099519516
3.14100783872
3.14102008514
3.14103195089
3.14104345158
3.14105460202
3.14106541631
3.14107590781
```

The `while` condition here is not hard to explain: it simply means, so long as $90 \leq n \leq 100$. Why the `n=90` at the beginning? In Python, we will have to assign a value to a variable before we can use it. The most interesting part of the code is the last line,

`n = n+1`

Mathematicians will definitely frown at the apparent absurdity of this equation, but this how we say in Python (and most other programming languages)

Assign to the variable `n`, its current value plus 1

Interpreted thus as an *assignment* rather than an *equation*, the code now makes sense:

- Assign the value 90 to the variable n
 1. Compute the value of $n \sin\left(\frac{180}{n}\right)$
 2. Display this value in the screen
 3. Increase the value of n by 1
- Repeat the steps (1), (2), (3) till n becomes 100

Now change the code to read `990 <= n <= 1000` and see whether the sequence moves closer to π .

As a final example of the `while` loop, let's consider Euclidean Algorithm to find the GCD of two natural numbers (the Mother of all algorithms). First, let's recall this algorithm, using an example.

Suppose we want to find the GCD of 162 and 60. The steps are as follows:

1. Divide 162 by 60 and take the remainder 42
2. Divide 60 by 42 and take the remainder 18
3. Divide 42 by 18 and take the remainder 6
4. Divide 18 by 6 which gives remainder 0
5. The GCD of 162 and 60 is the last non-zero remainder 6

Note that in this, the same procedure

Divide and find the remainder

is repeatedly on different pairs of numbers and these numbers themselves are those got at the different stages of this procedure. In other words, the entire procedure can be summarized like this:

- *Divide and find the remainder*
- *Change the dividend to the divisor and divisor to the remainder*
- *Repeat till remainder is 0*
- *The last non-zero remainder is the GCD*

Now let's see how we implement this in Python

```
# This program computes the greatest common divisor of
# two given integers using the Euclidean Algorithm

a = int(input("Type the first number : "))
b = int(input("Type the second number : "))

while b != 0:
    r = a % b
    a = b
    b = r
print("The GCD of the numbers is", a)
```

A Python run on this gives:

```
Type the first number : 162
Type the second number : 60
The GCD of the numbers is 6
```

Let's see the actual working of the code:

- Initial state : $a = 162, b = 60$

First loop	Second loop	Third loop	Fourth loop
(i) $r = 42$	(i) $r = 18$	(i) $r = 6$	(i) $r = 0$
(ii) $a = 60$	(ii) $a = 42$	(ii) $a = 18$	(ii) $a = 6$
(iii) $b = 42$	(iii) $b = 18$	(iii) $b = 6$	(iii) $b = 0$

- Final state : $a = 6, b = 0$

Note that the last print statement is *not* indented, so that it is not part of the loop and so is executed only after the looping has stopped. (See what happens if this line is indented and aligned with the loop block.)

Another point to note is the comma in the final print statement. Here, we want to print the *string* The GCD of the numbers is and the *integer* a ; and we have to separate them with a comma.

The above code can be shortened, perhaps at the cost of clarity, as follows:

```
a = int(input("Type the first number : "))
b = int(input("Type the second number : "))

while b != 0:
    a, b = b, a % b
print("The GCD of the numbers is", a)
```

Note that the loop assignment has to be *simultaneous* as above:

$$a, b = b, a \% b$$

and not *successive* like

```
a=b
b=a%b
```

(Why not? See what happens if we code it this way.) We look at another kind of loop in the next section.

3.2 for loop

The `for` loop is mainly used to iterate over a set of values. For example, look at our earlier example of computing $n \sin\left(\frac{180}{n}\right)$ for $90 \leq n \leq 100$. This can be coded also like this:

```
from math import sin, radians

for n in range(90, 101):
    p = n*sin(radians(180/n))
    print(p)
```

And gives the same output. Comparing this with the earlier program done with the `while` loop, we note the following differences

- There is no prior assignment of the variable `n = 90`, as in the first program
- Instead of `while 90 <= n <= 100` for the iteration, here we have `for n in range(90, 101)`
- There is no need to reassign the value of the variable, like `n = n+1`, as in the first program

Here `range(90,101)` makes a list of all *integers* from 90 to 100 (note, *not* 101). In fact it can be used to generate slightly more general lists of integers by specifying a *step* also. Thus:

```
>>> for n in range(-5, 16, 3):
...     print(n)
...
-5
-2
1
4
7
10
13
```

which are the integers from -5 to 16 incremented through 3 .

As another example of using the `for` loop, look at this program:

```
# This is a program to compute the sums of reciprocals of natural numbers

n = int(input("Type number : "))

a = 1
for i in range(2, n+1):
    a = a + 1/i
print(a)
```

For inputs 100 and 1000000 , this gives outputs as follows:

```
Type number : 100
5.18737751764
Type number : 1000
14.3927267229
```

which means

$$\sum_{k=1}^{10^2} \frac{1}{k} \approx 5.18737751764 \quad \text{and} \quad \sum_{k=1}^{10^6} \frac{1}{k} \approx 14.3927267229$$

showing how slowly the series diverges.

In much the same way, we can write a program to see that the series $\sum \frac{1}{n^2}$ approximates $\frac{1}{6}\pi^2$. (Try!) Again, we can write a program to compute approximate values of the Euler constant γ :

$$\gamma = \lim_{n \rightarrow \infty} \left(\sum_{k=1}^n \frac{1}{k} - \log n \right)$$

```
# This is a program to compute Euler's constant gamma

from math import log

n = int(input("Type number : "))

a = 1
for i in range(2, n+1):
    a = a + (1/i)
    g = a - log(i)
print("The Euler constant gamma is approximately", g)
```

For the input 1000000, this gives

$$\gamma \approx 0.577216164901$$

which is correct upto 5 places. (Incidentally, it is still unknown whether γ is rational or irrational)

Now as an illustration of the facilities afforded by the **range** function, we write a program which shows

$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots = \log 2$$

```
# This program shows how the alternating harmonic series
# approximates log(2)

from math import log

n = int(input("Type number : "))

a = 1
for i in range(2, n+1, 2):
    a = a - (1/i)
for i in range(3, n+1, 2):
    a = a + (1/i)
print(log(2) - a)
```

For input 10 and 1000, this gives the outputs 0.047512259925 and 0.000499750000121, showing

$$\log 2 - \left(1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots - \frac{1}{10}\right) \approx 0.047512259925$$

and

$$\log 2 - \left(1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots - \frac{1}{1000}\right) \approx 0.000499750000121$$

Note especially the use of **else** in the above program. Both the loop constructs **for** and **while** can have this clause; in a **for** loop, it is executed when the first list is exhausted and in a **while** loop, when the first condition becomes false. Thus if 5 is input in the above program, it first subtracts $\frac{1}{2}$ and $\frac{1}{4}$ successively from 1 and then adds $\frac{1}{3}$ and $\frac{1}{5}$.

This is an instance of *branching*, where a program forks into different kinds of actions, depending on specified conditions. This is very much needed in contexts other than loops also. We see more about this in the next section.

4 Branching

How do we decide whether a specified natural number is divisible by another specified natural number or not? Divide and see whether the remainder is zero or not. This idea can be translated into Python like this:

```
# This is a program which tests whether a given integer is divisible
# by another integer

number = int(input("Type number : "))
divisor = int(input("Type divisor : "))

if number % divisor == 0 :
    print(number, "is divisible by", divisor)
else:
    print(number, "is not divisible by", divisor)
```

The output is something like this:

```
Type number : 24597
Type divisor : 23
24597 is not divisible by 23
```

The only point to note is how the `if` clause is phrased: note that we have a double equal sign `==` in the verification part. This tests whether the number `number % divisor` is equal to the number 0 and in such *tests of equality*, we must use two equal signs together. (A single `=` sign *assigns* a value to a variable.)

As another example of *conditional statements*, consider this problem. The triangle with lengths of sides 5,5,6 and the rectangle with lengths of sides 6,2 have the same perimeter 16 and also the same area 12 (recall Heron's Formula). Are there other such triangle-rectangle pairs with integral sides?

The following Python program finds all such with lengths of sides less than 50.

```
# This is a program to find the lengths of sides of
# triangle-rectangle pairs of integral sides with
# equal areas and perimeters

for a in range (1,50):
    for b in range (a,50):
        for c in range (b,50):
            if a + b > c:
                perimeter_of_triangle = a + b + c
                s = perimeter_of_triangle/2
                area_of_triangle = ((s*(s-a)*(s-b)*(s-c))**0.5)
                for w in range (1,50):
                    for h in range (w,50):
                        perimeter_of_rectangle = (w + h)*2
                        area_of_rectangle = w*h
                        if (perimeter_of_triangle == perimeter_of_rectangle
                            and area_of_triangle == area_of_rectangle):
                            print("sides of triangle ", a,b,c)
                            print("sides of rectangle ", w,h)
                            print("Perimeter ", perimeter_of_rectangle,
                                "Area " , area_of_rectangle)
                            print(".....")
```

After running this and seeing that it works as claimed, certain points must be noted.

- It is a good practice to give easily understandable names to variables, such as `area_of_triangle` in the above program so that code is readable
- Python variable names can contain any letters, digits and underscore, but no other characters or spaces
- When code to be typed in a single line (according to Python syntax) goes beyond the editor window, it can be split, but the entire line must be within parentheses, as in the line `if (perimeter_of_triangle ... area_of_rectangle)` in the code above

Another useful fact is that if the output of a Python is too large for a single screen, instead of scrolling back, we can *redirect* the output to a file. Thus if the above program is in the file `areaperimeter.py`, then its output can be written to a file `apnumbers.txt` by the command

```
python3.1 areaperimeter.py > apnumbers.txt
```

and then we can read the file at our leisure. (In linux, any screen output can be redirected to a file like this.)

As yet another example of conditional statements, let's consider the problem of checking whether a given number is prime or not. First let's consider how we would go about doing this ourselves:

1. 1 and 2 are not primes
2. Divide the given number successively by 3, 4, ... upto (but not including) the number in turn and note the remainder
3. If the remainder at any stage is 0, we stop and conclude that the number is not a prime
4. If not we continue, till we have tried all numbers less than the given number
5. If we don't get zero remainder at any stage, then we conclude that the number is prime

Now let's see how these thoughts can be translated to Python code. First let's forget the special cases 1 and 2 and concentrate on other numbers. The procedure given in (2) and (3) above can be coded

```
number=int(input("Type the number to check : "))
for i in range(2,number):
    if (number%i)==0:
        print(number,'is not a prime')
```

This gives the correct output when the number is not a prime. To include the case of prime numbers, it seems as if we need only add an `else` condition:

```
number=int(input("Type the number to check : "))
for i in range(2,number):
    if (number%i)==0:
        print(number,'is not a prime')
    else:
        print(number,'is a prime')
```

But then if we run this, we get several things wrong. (Try it! It's a learning experience). For example

```
Type the number to check : 6
6 is not a prime
6 is not a prime
6 is a prime
6 is a prime
```

What happened? Look at the code again. Note that the `print` instruction is *within* the iteration loop *for*, so that every time a division is done, Python prints `is not a prime` or `is a prime` depending on whether the remainder is zero or not.

What we need is to print the conclusion after all iterations. But then the conclusion depends on whether the remainder is zero at some point or not at all. The trick is to assign a variable *string* as our conclusion. First we initialize it as the empty string and then assign it values depending on the remainder:

```
number=int(input("Type the number to check : "))
conclusion=''
for i in range(2,number):
    if (number % i) == 0:
        conclusion='is not a prime'
    else:
        conclusion='is a prime'
print(number,conclusion)
```

Now if we run this code, then we get only one line of output, but it gives `is a prime` for *every* number! Now what? Back to the code.

The problem is that the program does not exit the iteration loop after getting a zero remainder, but continues with iteration to the end and prints the final conclusion. What we must do is to ask the program to exit the loop of division, once a zero remainder is found. This is done by the `break` instruction:

```
number=int(input("Type the number to check : "))
conclusion=''
for i in range(2,number):
    if (number%i)==0:
        conclusion='is not a prime'
        break
    else:
        conclusion='is a prime'
print(number, conclusion)
```

Now everything is alright, except for the particular cases 1 and 2. (check what happens if we input 1 or 2 in running the above code) We must have the conclusion `not a prime` for 1 and `is a prime` for 2. And then the routine above. This entails several `else's`. We can use the `elif` instruction (stands for *elseif*):

```
# This is a program to check whether a given number is a prime.

number=int(input("Type the number to check : "))
conclusion=''
if number==1:
    conclusion='is not a prime'
    print(number,conclusion)
elif number==2:
    conclusion='is a prime'
    print(number,conclusion)
else:
    for i in range(2,number):
        if number%i==0:
            conclusion='is not a prime'
            break
        else:
            conclusion='is a prime'
    print(number,conclusion)
```

Let's analyze this program line-by-line:

- The first line accepts an integer input
- The second line defines a variable named `conclusion` and assigns the *empty string* `''` as its value
- The third line checks if the number is 1 and if this be so, the fourth line assigns the string `'is not a prime'` as the value of the variable `conclusion`; and then the fifth line prints this conclusion.
- If the number input is not 1, the next lines starting with `else:` comes into action, dividing the number in succession by 2, 3,
- If the remainder at any stage is 0, the variable string `conclusion` is assigned the value, `'is not a prime'` and then the `if` loop stops, because of the `break` command, and passes control to the last line to print the conclusion
- If the remainder is not 0 for any number, then the `if` loop continues to the end without break, and passes control to the `else` clause of the `for` loop, which assigns the string `'is not a prime'` as the value of `conclusion` and then exits, which makes the last line print this string as the conclusion

A typical run of this program goes something like this:

```
Type the number to check : 24597
24597 is not a prime
```

Now everything should (hopefully) run fine. (This use of the `else` clause in a `for` loop is something peculiar to Python and is a matter of a good deal of debate on its “correctness” according to coding practices)

We know that to check primality of n , we need only test divisibly upto \sqrt{n} (Why?) So we can modify the range in the iteration loop as

```
for i in range(2,int(number**0.5+1)):
```

But then 3 will be a problem, since $\sqrt{3} < 2$. So we will modify the code as follows:

```
# This is a program to check whether a given number is a prime.

number=int(input("Type the number to check : "))
conclusion=''
if number==1:
    conclusion='is not a prime'
    print(number,conclusion)
elif number==2 or number==3:
    conclusion='is a prime'
    print(number,conclusion)
else:
    for i in range(2,int(number**0.5+1)):
        if number%i==0:
            conclusion='is not a prime'
            break
    else:
        conclusion='is a prime'
        print(number,conclusion)
```