

# Math with Python

E. Krishnan  
e.krshnana@gmail.com

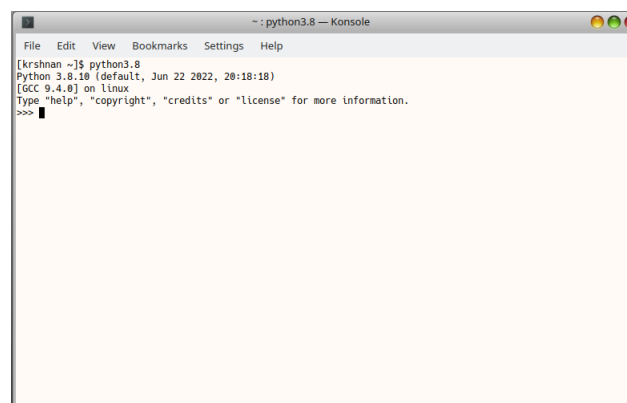
This is a short introduction to the Python programming language emphasizing its uses in basic mathematics. We use Python3.8 in a Linux machine.

## 1 Interactive Mode

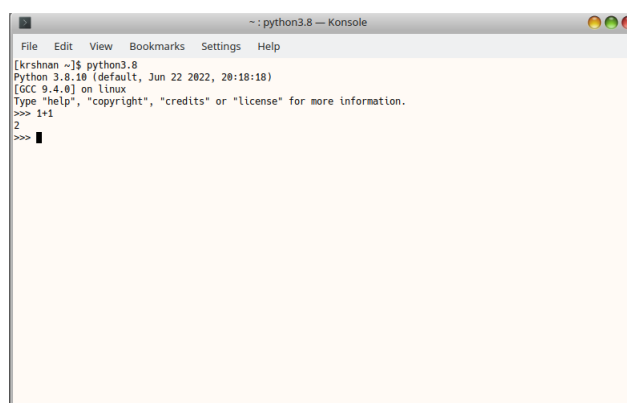
In a Linux system, a *terminal* is a command line interface where we type commands. In the default state, the commands we type are instructions to the operating system, and these are interpreted by what is called a *shell*. To start Python, we type in the terminal

`python3.8`

and press the **Enter** key. This command asks the shell to instruct the operating system to start the Python *interpreter*, which in turn prints a message and changes the prompt from shell prompt `$` to the Python prompt `>>>`:

A screenshot of a terminal window titled "python3.8 — Konsole". The window has a menu bar with "File", "Edit", "View", "Bookmarks", "Settings", and "Help". The terminal output shows the command `[krshnan ~]$ python3.8` being executed. The response is `Python 3.8.10 (default, Jun 22 2022, 20:18:18)`, followed by `[GCC 9.4.0] on linux` and `Type "help", "copyright", "credits" or "license" for more information.`. The prompt has changed from `$` to `>>>`, and a cursor is visible on the line following the prompt.

Now whatever we type goes to the Python interpreter. For example if we type `1+1` and press enter, we get this:

A screenshot of the same terminal window. The output now includes the line `>>> 1+1` followed by the result `2` on the next line. The prompt `>>>` is now on the line following the result, with a cursor.

In the following, we won't show anymore screen-shots, but simply give our **inputs** and the python **outputs** using typewriter font. For example:

```
>>> 3-1
2
```

## 1.1 Arithmetic operations

We can do other basic operations in Python, using `*` for multiplication, `/` for division and `**` for exponentiation:

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

Python supports arbitrarily large integers. As an example, see how Python computes  $2^{1000}$ :

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

(Now try `2**1000000` and see what happens!)

Note that Python answers the division problem `14/5` as the decimal `2.8`. We can get the integer quotient and remainder using the operations `//` and `%`:

```
>>> 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
```

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
```

This is how we use Python as a calculator. To quit the session, either type `quit()` at the prompt, or press the `Ctrl` and the `d` keys together.

## 1.2 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

$$a = p \left(1 + \frac{r}{100}\right)^n$$

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.

From what we have seen so far, we can write the right side of the above formula as `p*(1+r/100)**n`. To round it to the nearest integer, we use the `round` operator. Thus we write:

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

The first three lines assign the numbers 1000, 6, 3 to the three *variables* `p`, `n`, `r` and the fourth assigns the number `1000*(1+6/100)**3`, rounded to the nearest integer, to the variable `a`.

Now change the values of `p`, `n`, `r` and ask to compute `a` again:

```
>>> p = 15000
>>> r = 5
>>> n = 4
>>> a
11910
```

Why doesn't `a` change? Well, our first computation assigned the number 11910 to `a` and our reassignments of `n`, `p`, `r` does not change this automatically. We must explicitly ask to recompute `a` using these new values of `n`, `p`, `r` as below:

```
>>> p = 15000
>>> r = 5
>>> n = 4
>>> a
>>> a = round(p*(1+r/100)**n)
18233
```

The so called *history mechanism* in a Linux system, which means we can cycle through previously typed commands by pressing the up and down arrow keys is of some help here: after entering the new values of `p`, `n`, `r`, we can repeatedly press up arrow key till we get the command defining `a` and press the Enter key, to insert this line. But still, it's a bit cumbersome.

One way to get over this is by using *functions*. This we consider next

## 1.3 Functions

In mathematics, a function specifies a rule of computation. So is the case with python; only the syntax of *defining* a function is different. As an example, consider the computation of compound interest. Mathematically it involves the computation of the total amount `a` in terms of the principal `p`, interest rate `r` and the number of years `n` using the formula

$$a(p, r, n) = p \left(1 + \frac{r}{100}\right)^n$$

and to be practical we round it (half up) to the nearest integer. We define this (in fact any function) in Python in two lines of code: in the first line, we *define* the name of the function and the name of the variables it acts upon, and in the second line we specify the value it should *return*, by describing its operation on the variables. Thus in our compound interest example, we start Python as before and at its prompt, first type:

```
>>> def a(p,r,n):
```

Note that the line should end with a colon (:) which essentially says that we want to continue the instruction. On pressing Enter, we get the secondary prompt ..., since the definition is not complete. To complete it, we must state what we want the function to *return* (that is, what it should compute and give us):

```
>>> def a(p,r,n):
...     return(round(p*(1+r/100)**n))
```

Note carefully that the second line is *indented* from the left by (at least) one space. That is, we must hit the spacebar at least once before typing in this second line.

After typing this and pressing Enter, we again get the secondary prompt ... again and hitting Enter once more, we get the primary prompt >>> back. Now we can compute **a** for various values of **p**, **r**, **n** as follows:

```
>>> def a(p,r,n):
...     return(round(p*(1+r/100)**n))
...
>>> a(1000,5,3)
1158
>>> a(15000,6.5,4)
19297
>>> a(2000,6.5,3)
2416
```

As an another example, consider the formula for the area of triangle of sides  $a$ ,  $b$ ,  $c$ :

$$\text{Area} = \sqrt{s(s-a)(s-b)(s-c)} \text{ where } s = \frac{1}{2}(a+b+c)$$

To define this in Python, we rewrite the above formula explicitly in terms of  $a$ ,  $b$ ,  $c$  as

$$\text{Area} = \frac{1}{4}\sqrt{(a+b+c)(b+c-a)(c+a-b)(a+b-c)}$$

and use it in Python:

```
>>> def trarea(a,b,c):
...     return((1/4)*((a+b+c)*(b+c-a)*(c+a-b)*(a+b-c))**(1/2))
...
>>>
```

(Note that in Python, we can use any word (or just any combination of letters as the name of a variable or function, instead of a single letter.) Then we can compute the area of any triangle by continuing as below:

```
>>> def trarea(a,b,c):
...     return((1/4)*((a+b+c)*(b+c-a)*(c+a-b)*(a+b-c))**(1/2))
...
>>> trarea(3,4,5)
>>> 6.0
>>> trarea(3.5,10.7,12.2)
17.891338686638292
>>>
```

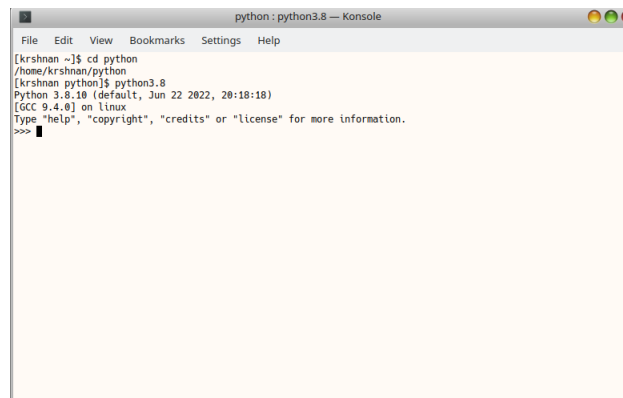
We can also define the above function by defining the variable `s` before `return`:

```
def trarea(a,b,c):  
    s=(a+b+c)/2  
    return((s*(s-a)*(s-b)*(s-c))**(1/2))
```

One drawback of using functions this way is that once we quit the current Python session, the definition of the functions vanish, so that the next time we start Python and want to use these functions, we will have to define them all over again! One way to overcome this is to make a file `myfunctions.py` (the name can be anything, but the extension must be `.py`) containing this function. The best way to do this is to make a directory (folder) named `python` (or any other name), where we keep all our python files. Open a text editor (such as `Gedit` or `Emacs` and *not* in a word processor such as `LibreOffice Writer`) and type these lines:

```
# This is a function to compute the compound interest  
  
def compound(p,r,n):  
    return(round(p*(1+r/100)**n))  
  
# This is a function to compute the area of a triangle  
  
def trarea(a,b,c):  
    return((1/4)*((a+b+c)*(b+c-a)*(c+a-b)*(a+b-c))**(1/2))
```

A line after a `#` will not be read by Python and is just a comment for our own reference. Save this in the `python` directory as `myfunctions.py`. We can give any name to this file, but the extension must be `.py`. The next time we want to use any of these functions, say, `compound`, start Python from this directory, by first issuing the command `cd python` and then `python3.8`:



```
python: python3.8 — Konsole  
File Edit View Bookmarks Settings Help  
[krshnan ~]$ cd python  
/home/krshnan/python  
[krshnan python]$ python3.8  
Python 3.8.10 (default, Jun 22 2022, 20:18:18)  
[GCC 9.4.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>>
```

Now if we want to use the `compound` function, first type the command shown below at the Python prompt:

```
>>> from myfiles import compound  
>>>
```

Then we can continue to give commands like

```
>>> from myfiles import compound  
>>> compound(2000,6,4)  
2531  
>>>
```

We can use the `triarea` also in the same session by typing

```
>>> from myfiles import triarea
```

at any stage.

We have been using only basic mathematical operations till now. To use further mathematical operations like the trigonometric functions, we will have to give some special instructions. We will talk about this in the next section.

## 1.4 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* them. So, if we want to compute the sine value, we do

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

Recall how we imported our own functions, as explained in the last section. In fact the file `myfunctions.py` mentioned there is a module.

Note that the example above gives the value of the sine of 1 *radian* and not 1 degree. The `sin` function in Python is designed to interpret the argument as radians. To convert degrees to radians, the **math** module has the function `radians`. Thus

```
>>> from math import radians
>>> radians(30)
0.5235987755982988
>>> from math import pi
>>> pi/6
0.5235987755982988
```

So if we want  $\sin 1^\circ$ , we go

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

To see other functions available in the **math** module, see <https://docs.python.org/3/library/math.html>

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

```
>>> from fractions import Fraction
>>> Fraction(3,8)+Fraction(9,17)
Fraction(123, 136)
>>> Fraction(23.375)
Fraction(187, 8)
```

More on this module can be found at <https://docs.python.org/3/library/fractions.html>

## 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.

For example, suppose we want to find the quotient and remainder on dividing a number by another. We can define this as Python function:

```
def qr(a,b):  
    return(a//b,a%b)
```

We can either type this in a running python session and get quotient and remainder for various pairs of dividend and divisor as in

```
>>> def qr(a,b):  
...     return(a//b,a%b)  
...  
>>> qr(18,7)  
(2, 4)  
>>> qr(2317,198)  
(11, 139)  
>>>
```

which shows  $18 = 2 \times 7 + 4$  and  $2317 = 11 \times 198 + 139$ . Or we can save this definition in a file and import it as explained earlier.

There's yet another way. We type the function definition and the dividend-divisor pairs in a single file and run Python on it, to get the quotient-remainder pair displayed in the terminal. To do this, open a *text editor* and type the following lines:

```
def qr(a,b):  
    return(a//b,a%b)  
  
print(qr(18,7))  
print(qr(2317,198))  
print(qr(26751,3456))
```

The only new thing here is the **print** command. It simply asks Python to display something (in this case, the quotient-remainder pair) in the terminal, when the program is run. Save this file in the python directory as `division.py` (or any other name, but the extension must be `.py`). Now open a terminal, `cd` to the python directory and type the command

```
python3.8 division.py
```

We get the three pairs of quotient and remainder displayed in the terminal:

```
(2, 4)  
(11, 139)  
(7, 2559)
```

Instead of typing all the dividend-divisor pairs in the program file itself, we can instruct Python to ask for the divisor and dividend one by one and then print the quotient-remainder pairs.

We do this using the **input** command. To see this in action, first start python in the terminal and type

```
input('Hello')
```

On pressing Enter we get

```
input('Hello')  
>>> Hello
```

Note that we don't get the prompt back. Actually, Python is waiting for some *input* from us. For the time being, press **Enter** again to get the prompt back. Note carefully the quotes around the word Hello. To see why they are necessary, try without the quotes next: On pressing **Enter** we get

```
>>> input(Hello)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Hello' is not defined
>>>
```

This is a typical Python error message. The last line tells what the error is. Recall that any combination of letters can be used to define a Python variable. Here Python treats Hello as the name of a variable and flags an error, since such a variable is not defined. If instead, we enclose Hello within quotes, then Python treats it just as a combination of characters. (Technically, a *string*; we will talk about strings in detail later.)

Now let's try another experiment. At the Python prompt, type

```
input('Number')
```

On pressing **Enter**, Python displays **Number** and waits, as seen before. But instead of pressing **Enter**, type a number, say, 314:

```
>>> input('Number')
Number314
'314'
>>>
```

Note that what is displayed is 314 *within* quotes; that is, the string of characters 3, 1 and 4 and not the number 314. To see this, do this experiment:

```
>>> a=input('Number')
Number314
>>> a
'314'
>>> a+7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>>
```

What happened? Our first line assigns to the Python variable **a**, whatever we input as a response to 'Number'. Our input is 314, which Python treats as the string. So when we ask Python to add 7 to the value of **a**, Python responds with the error message that an integer cannot be added to a string.

So, how do we make Python accept our input as a number? We use the **int** operator to convert a string of digits to the corresponding integer:

```
>>> int('314')
314
>>> int('-345')
-345
```

So now we can write a program which asks us for a dividend and divisor and gives us the quotient and remainder:



```
#Program to compute quotient and remainder

a=int(input('Dividend : '))
b=int(input('Divisor  : '))

q=a//b
r=a%b

print('Quotient  : ',q)
print('Remainder : ',r)
```

Running Python on this file first displays **Dividend :** and when we type an integer and press **Enter**, responds with **Divisor :** . Typing another integer and pressing **Enter** gives the quotient and remainder:

```
Dividend : 37891
Divisor  : 789
Quotient  : 48
Remainder : 19
```

Note that the first **print** command instructs Python to display the string '**Quotient :**' followed by the value of the variable **q**. Thus we can make Python print various entities, by separating them with commas.

We can write a similar program to compute the area of a triangle, but here the lengths need not be integers. So, we use the operator **float** (for *floating point*, a term used in computer science to denote approximate decimal representations of real numbers.) to convert strings with a dot inside into corresponding decimal numbers. Thus the program would be like this:

```
#Program to compute area of triangle

a=float(input('First side  : '))
b=float(input('Second side : '))
c=float(input('Third side  : '))

s=(a+b+c)/2
A=round((s*(s-a)*(s-b)*(s-c))**(1/2),2)

print('Area : ',A)
```

Here the number 2 after the formula to compute the area instructs Python to round the answer to two decimal places. A typical run of this program is like this:

```
First side  : 4.6
Second side : 2.7
Third side  : 3.1
Area : 4.05
```

We have been so far using Python as a sort of fancy calculator, with no actual programming. We next consider some examples involving some programming constructs.

### 3 Loops

In computer programming, a loop is a sequence of instructions that is repeated subject a specified condition. As in other programming languages, Python also has two different kinds of loops. We look at them one by one.

### 3.1 while loops

We illustrate the idea of a **while** loop using an earlier example. Consider our program to compute the quotient and remainder of dividing a number by another. On being run, the program asks us for the dividend and divisor, computes the quotient and remainder, and then quits. If we want to do this for another pair of numbers, we have to run Python again. It would be nice to have the program ask us whether to continue or quit after each such computation. See the program below:

```
#Program to compute quotient and remainder till asked to stop

option='yes'

while option=='yes':
    a=int(input('Dividend : '))
    b=int(input('Divisor  : '))
    q=a//b
    r=a%b
    print('Quotient   : ',q)
    print('Remainder  : ',r)
    print('Do you wish to continue?')
    option=input('Type yes or no : ')
```

There are only a few changes from the program we wrote earlier for this computation. The first line defines a new variable **option** and assigns the *string yes* as its value. The second line

```
while option=='yes'
```

asks Python to check the value of the variable **option** and so long as it is the string **yes**, carry out the next instructions. (Note carefully that **=** is used to *assign* a specified value to a variable and **==** is used to *check* whether a variable has a specified value). The remaining lines instructs Python to do the following, *as long as the value of option remains the string yes*:

- Ask for dividend and divisor
- Compute quotient and remainder and display them
- Display the message **Do you want to continue?**
- Display the message **Type yes or no** and assign our response as the new value of the variable **option**

So for the final message, if we type **yes**, then the whole sequence of operations is repeated. If we type **no** (actually anything other than **yes**), the program stops. For example:

```
Dividend : 389
Divisor  : 17
Quotient  : 22
Remainder : 15
Do you wish to continue?
Type yes or no : yes
Dividend : 3987
Divisor   : 234
Quotient  : 17
Remainder : 9
Do you wish to continue?
Type yes or no : no
```

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 indentations indicate *blocks*, instead of parantheses or braces as in other programming languages.

As another example of the **while** loop, let's consider the problem of computing the factorial of a number. In ordinary language, we can simply say "multiply all numbers from 1 to  $n$ ", for a specified number  $n$ . But in a program, we'll have to break this up as

- (i) Start with 1
- (ii) Multiply by 2
- (iii) Multiply the product by 3
- (iv) Repeat till  $n$

Let's denote the multipliers  $2, 3, \dots, n$  by a variable **i** and the value of the product at each stage of multiplication by the variable **f**. Then we can write the procedure above as

- (i) Start with **f=1** and **i=2**
- (ii) Compute **f\*i**
- (iii) Change the value of **f** to **f\*i** and the value of **i** to **i+1**
- (iv) Repeat till the value of **i** becomes **n**

This is what happens at every stage of this procedure:

Stage 1	<b>f=1</b>	<b>i=2</b>
Stage 2	<b>f=1*2</b>	<b>i=3</b>
Stage 3	<b>f=1*2*3</b>	<b>i=4</b>
...	...	...
Stage n	<b>f=n!</b>	<b>i=n</b>

Now it's not very difficult to convert this into actual code:

```
#Program to compute factorial

n=int(input('Type the number : '))

f=1
i=2
while 2<=i<=n:
    f=f*i
    i=i+1
print(f)
```

Here, the symbol  $\leq$  stands for  $\leq$  (less than or equal to). The two lines

```
f=f*i
i=i+1
```

mean to change the value of `f` to that of `f*i` and the value of `i` to `i+1`. Note that these lines are typed with an indentation from the beginning of the line `while 2<=i<=n`, that is they are within the scope of the `while` loop, which means they are repeated (till `i` reaches the value `n`). But the `print` command is not indented, so that it is not within the scope of the loop, and so the value of `f` is displayed *only after all computations are done*. Thus only the final value of `f`, which is the factorial of `n`, is displayed. It is an interesting exercise to see what happens if we include the `print` statement also within the scope of the `while` loop, that is, change the code to

```
n=int(input('Type number : '))

f=1
i=2
while 2<=i<=n:
    f=f*i
    i=i+1
print(f)
```

We can also write a function to compute factorials:

```
# Function to compute factorials using while loop

def factorial(n):
    f=1
    i=2
    while 2<=i<=n:
        f=f*i
        i=i+1
    return(f)
```

One use of this is that it can be included in other programs which require factorials. For example, the number `e` is defined as the sum of an infinite series (that is, the limit of successive finite sums):

$$e = \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

We can write a program to approximate `e`:

```
# Program to approximate e

def factorial(n):
    f=1
    i=2
    while 2<=i<=n:
        f=f*i
        i=i+1
    return(f)

n=int(input('Number of terms : '))

s=1
i=1
while i<=n:
    s=s+1/factorial(i)
    i=i+1
print(s)
```

If we want the sum of the first 10 terms we give 10 as the number of terms when the program asks for it:

```
Number of terms : 10
2.7182818011463845
```

If we want to see how the digits in the decimal places stabilize, include the `print` statement within the scope of `while` in the above program, by pushing the last `print` a space to the right: `while`:

```
n=int(input('Number of terms : '))

s=1
i=1
while i<=n:
    s=s+1/factorial(i)
    i=i+1
    print(s)
```

Now when we give 10 as the number of terms, we get

```
Number of terms : 10
2.0
2.5
2.6666666666666665
2.7083333333333333
2.7166666666666663
2.7180555555555554
2.7182539682539684
2.71827876984127
2.7182815255731922
2.7182818011463845
```

which shows the approximation 2.718281 is correct upto six decimal places.

### 3.2 for loops

We have seen a program to compute factorials, using the `while` loop. There's another way of doing this, using a different kind of looping. For this, let's look at how we compute the factorial of a specified number  $n$ : we start with 1 and then multiply 1 successively by the natural numbers from 2 to  $n$ . In other words, we create the list of natural numbers from 2 to  $n$  and multiply 1 by the numbers in this list. In Python, this list can be created by `range(2,n+1)`. Then we use the `for` loop to iterate over this list:

```
# Program to compute factorial using for

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

f=1
for i in range(2,n+1):
    f=f*i
print(f)
```

Note carefully that `range(2,n+1)` is the list of natural numbers from 2 to  $n$ ; that is all integers  $k$  with  $2 < k < n + 1$ .

Comparing this with the earlier program done with the `while` loop, we note the following differences

- There is no prior assignment of the variable `i=2`, as in the first program
- Instead of `while 1<=i<=n` for the iteration, here we have `for i in range(2, n)`

- There is no need to reassign the value of the variable, like  $i=i+1$ , as in the first program

As another example of **for** loop, let's compute the sum of reciprocals of natural numbers; that is,

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

We can do it like this:

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

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

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

This is what we get if we sum upto  $\frac{1}{1000}$ :

```
Number of terms : 1000
7.485470860550343
```

And if we go upto  $\frac{1}{1000000}$ , we get

```
Number of terms : 1000000
14.392726722864989
```

This means

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{1000} \approx 7.485470860550343$$

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{1000000} \approx 14.392726722864989$$

From these, it may seem as if the sums approach a definite number as we sum more and more terms, but it is not actually so. In fact, by taking *sufficiently* many terms, the sum can be made *as large as we please*. For example, it can be proved that the sum up to  $\frac{1}{2^{1998}}$  is larger than 1000. Even Python cannot do this sum (at least in our computers). To see how big a number is  $2^{1998}$ , just type `2**1998` at a Python prompt.

Let's do one more example. In the 14<sup>th</sup> century, the Kerala mathematician Madhavan discovered that if we alternatively add and subtract the reciprocals of odd numbers, we get better and better approximations for  $\frac{1}{4}\pi$ . That is, the sums

$$1, 1 - \frac{1}{3}, 1 - \frac{1}{3} + \frac{1}{5}, 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7}, \dots$$

get closer and closer to  $\frac{1}{4}\pi$ . (In modern terminology, we write

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots = \frac{1}{4}\pi$$

and say that the series converges to  $\frac{1}{4}\pi$ .)

So to approximate  $\pi$ , we need only multiply the sums at each stage by 4. Here's the Python program to do it:

```
#Program to approximate pi using Madhavan series

n=int(input("Number of terms : "))
s=0
for i in range(1,n+1):
    s=s+(-1)**(i+1)/(2*i-1)

print(4*s)
```

But this approximates  $\pi$  very slowly. For example, here's what we get by summing 1000 terms:

```
Number of terms : 1000
3.140592653839794
```

and this is correct only up to two decimal places. But Madhavan himself seems to have realized this, for he says the correction terms

$$\frac{n^2 + 1}{4n^3 + 5}$$

is to be added or subtracted, depending on whether we take an even or odd number of terms. Let's incorporate this correction also into our program:

```
# Program to approximate pi using Madhavan series with correction

n=int(input("Number of terms : "))
s=0
for i in range(1, n+1):
    s=s+(-1)**(i+1)/(2*i-1)
    s=s+(-1)**n*(n**2+1)/(4*(n**3)+5*n)

print(4*s)
```

Now see what happens when we sum just ten terms and add the correction:

```
Number of terms : 10
3.141592705349156
```

And this is correct up to six decimal places!

## 4 Conditional statements

How do we find the factors of a number? We divide the number by all numbers less than (or equal to) it and choose those for which the remainder is zero. Let's write this in detail, so that we can turn it into a Python program:

- (i) We divide the number, say  $n$ , by each of the numbers  $1, 2, 3, \dots, n$
- (ii) If the remainder is zero for any one of these numbers, we note it as a factor. Otherwise we discard it

the first step can be done in Python by doing `n%i for i in range(1,n+1)`. the second step requires checking `n%i==0` and if so, print it. This is done as follows:

```
#Program to find the factors of a number

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

for i in range(1,n+1):
    if n%i==0:
        print(i)
```

A sample output is like this:

```
Type number : 6541
1
31
211
6541
```

Not that the `if` condition is in a line by itself, terminated by a colon and the `print` statement is indented.

Very often, we want a program to do something if a specified condition is satisfied and some thing else if the condition is not satisfied. In such cases, the `if` is coupled with an `else`. In the example above, this is not necessary, since if `n%i==0` (that is if `i` is not a factor of `n`, then there is not anything else to do than skipping to the next number `i+1`, and this is taken care of by `for i in range(1,n+1)`.

Just to illustrate the use of the `if...else` construct, let's change the above code to this:

```
n=int(input('Type number : '))

for i in range(1,n+1):
    if n%i==0:
        print(i, 'is a factor of', n)
    else:
        print(i, 'is not a factor of', n)
```

A sample output is shown below:

```
Type number : 10
1 is a factor of 10
2 is a factor of 10
3 is not a factor of 10
4 is not a factor of 10
5 is a factor of 10
6 is not a factor of 10
7 is not a factor of 10
8 is not a factor of 10
9 is not a factor of 10
10 is a factor of 10
```

Now consider the problem of finding the *prime* factors of a number  $n$ , repeating each how many times it divides the number. Let's think how we would do it manually:

1. Check if  $n$  is divisible by 2
2. If divisisible by 2, include 2 as a factor and
  - (a) check if the quotient  $\frac{n}{2}$  is divisible by 2
  - (b) continue till the quotient is not divisible by 2
  - (c) move onto 3
3. If not divisble by 2, move on to 3 and repeat the above steps
4. Repeat this for all primes less than  $n$

Now this procedure checks divisiblity by only primes and so we need a list of consecutive primes to use it. And since there is no formula to generate such a list, it is not easy to implement a computer program.

But there's a way out. Note that in this scheme, after exhausting the 2's and 3's, we need not check for 4, since even if it were a factor, we would've removed it when we divided by 2 twice. Similarly we need not check 6, since even if it were present as a factor, it would've been removed when we divided by 2 and 3. But there's no harm in (unnecessarily) checking 4 and 6, since we would only find that they are not factors of the quotient. This gives an idea of how to write a program to do it:



1. Start with  $n$  as the given number and  $i$  as 2
2. Check whether  $n \% i == 0$ 
  - (a) If so, `print(i)` and change the value of  $n$  to  $n // i$
  - (b) If not, change  $i$  to  $i + 1$  and repeat
3. Continue till  $n$  is 1

Now we can write the program as below:

```
# Program to compute prime factors

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

i=2
while n>1:
    if n%i==0:
        print(i)
        n=n//i
    else:
        i=i+1
```

Running this program gives output like this:

```
Type number : 360
2
2
2
3
3
5
```

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:

- (i) 1 and 2 are not primes
- (ii) Divide the given number successively by 3, 4, ... upto (but not including) the number in turn and note the remainder
- (iii) If the remainder at any stage is 0, we stop and conclude that the number is not a prime
- (iv) If not we continue, till we have tried all numbers less than the given number
- (v) 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 of the numbers 1 and 2 (Step (i) above) and concentrate on other numbers. The procedure given in (ii) and (iii) above can be coded:

```
n=int(input("Type the number to check : "))
for i in range(2,n):
    if (n%i)==0:
        print(n,'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:

```

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

```

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

What happened? Look at the code again. Note that both the `print` instruction are *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 want is to stop the `for` loop once the remainder is zero for some value of `i` and hence printing **not a prime**. The `break` statement in Python does just that. So we write

```

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

```

But then if no zero remainder is found at any stage, we want the program to print **is a prime**. For this, we can give an `else` clause to the `for` loop. Whatever instruction we give under this `else` will be executed only if the `for` loop goes through without a `break`; here this means, only if no zero remainder is found for any value of `i`. Thus we write

```

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

```

Now it's only a question of handling 1 and 2. Note that the number 2 is already taken care of in the above code, for if `n=1`, then `range(1,n)` is `range(2,2)`; and this list does not contain any integer since there is no integer  $k$  with  $2 < k < 2$ . So, the `for` loop is not executed at all and the `else` clause takes over.

So, we can put together all pieces and write the final program:

```

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

n=int(input("Type the number to check : "))

if n==1:
    print(n, 'is not a prime')
if n>1:
    for i in range(2,n):
        if n%i==0:
            print(n, 'is not a prime')
            break
    else:
        print(n,'is a prime')

```

Note carefully the indentation level of the final `else` clause. It is attached to the `for` loop (so that it has the same indentation as `for`) and not to the `if n>1` condition.

Actually this program is a bit inefficient, since to check a large number, quite a number of possible factors are to be checked. We can reduce the number of divisions by noting that if  $n$  is not a prime, then it has a factor less than or equal to  $\sqrt{n}$  (Can you see why?). So, we can modify our program like this:

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

n=int(input("Type the number to check : "))

if n==1:
    print(n, 'is not a prime')
if n>1:
    for i in range(2,int(n**0.5)+1):
        if n%i==0:
            print(n, 'is not a prime')
            break
    else:
        print(n,'is a prime')
```

With a slight change in the code, we can find all primes below a specified number: Suppose we want all primes less than a specified number  $n$ . What we do is take all natural numbers from 1 to  $n$  and check whether *each* of these is a prime or not. In Python terminology,

- (i) `for m in range(1,n)` check whether  $m$  is a prime, (using our earlier code)
- (ii) `n` print those  $m$  which are primes

The program below will do the trick:

```
# Program to compute all primes below a given number

n=int(input("List primes upto : "))

for m in range(2,n+1):
    for i in range(2,int(m**(0.5)+1)):
        if m%i==0:
            break
    else:
        print(m)
```

Here's a sample run:

```
List primes upto : 25
2
3
5
7
11
13
17
19
23
```

But if we do it for 100 or 1000, the output scrolls off the screen. It'd be nice if we can give the output as a horizontal list. It is one (trivial) use of a Python *list*, which we consider next.

## 5 Lists

We have seen that in Python we can use *numbers* such as integers, decimals, complex and also *strings*. In computer parlance, these are the some of *data types* Python can handle. Lists form another type of data in Python. A list is specified by enclosing its entries separated by commas, within square brackets [ ]. Try these at a Python prompt:

```
>>> l=[1,2,3,4]
>>> l
[1, 2, 3, 4]
>>> L=['a','b','c']
>>> L
['a', 'b', 'c']
```

We create a list by writing something like `l=[]` and we can put anything in this list by the instruction `l.append`. Thus we can modify our earlier program for computing primes below a specified number to output these as a horizontal list list, instead of a vertical stack:

```
# Program to list all primes below a given number

n=int(input("List primes below : "))

P=[]

for m in range(2,n+1):
    for i in range(2,int(m**(0.5)+1)):
        if m%i == 0:
            break
    else:
        P.append(m)
print(P)
```

Try this to get a list a primes below 1000 or even 1000000.

We can also define a function which returns the list of primes below it, and put in our `myfunctions.py` file:

```
# Function to compute the list of primes below a specified number

def primes_below(n):
    P=[]
    for m in range(2,n+1):
        for i in range(2,int(m**(0.5)+1)):
            if m%i == 0:
                break
        else:
            P.append(m)
    return(P)
```

Note the name of the function: it's named for quick recollection, and the underscore between `primes` and `below` is just for readability. Now we can use this at a Python prompt as below:

```
>>> from myfunctions import primes_below
>>> primes_below(50)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Now suppose we want the primes between two natural numbers. This we can get using our `{prime_list}` functions itself. For example suppose we want the primes between 50 and 100.

What we do is create a list of primes below 100 as above and then create another list consisting only of those numbers in the first list which are greater than 50, as shown below:

```
>>> from myfunctions import primes_below
>>> P=primes_below(100)
>>> L=[x for x in P if x>50]
>>> L
[53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Incidentally this shows another way of creating a list, by specifying a condition, instead of explicitly specifying each entry. (Recall the roster and rule method of specifying sets in math.)

We can have some more fun with this. Suppose we are only interested in the *number* of primes below a specified number. That is, just the number of entries in our `{primes_below(n)}` for some `n`. In Python, the number of entries in a list is called its length and is given by the `len` function. Thus to find just the *number* of primes below 1000, we do this:

```
>>> from myfunctions import primes_below
>>> P=primes_below(1000)
>>> len(P)
168
```

and find that there are 168 primes below 1000. Again, the number of primes between 100 and 1000 can be computed as

```
>>> from myfunctions import primes_below
>>> P=primes_below(1000)
>>> L=[x for x in P if x>100]
>>> len(L)
143
```

which shows there are 143 primes between 100 and 1000.

Next note that by the very process of appending numbers to `P`, in our `primes_below` function, the numbers are in ascending order. This means if we want the 100<sup>th</sup> prime, we need only locate the hundredth member of `P`. Here, a note of caution must be added. Python names the entries of `P` as `P[0]`, `P[1]`, `P[2]`, ..., so that  $n^{\text{th}}$  entry of the list is `P[n-1]` (and not `P[n]`.) So to get the 100<sup>th</sup> prime, we do this:

```
>>> from myfunctions import primes_below
>>> P=primes_below(1000)
>>> P[99]
541
```

which shows, the hundredth prime is 541.

Not only can we extract the single entry of a list at a specific position, we can also extract a part of a list between specified positions. For example, suppose (for whatever reason) we want the list of primes from the 10<sup>th</sup> to the 20<sup>th</sup>. This means, we want the list of `P[9]` to `P[19]`. This is written `P[9:19]`. Thus

```
>>> from myfunctions import primes_below
>>> P=primes_below(1000)
>>> P[9:19]
[29, 31, 37, 41, 43, 47, 53, 59, 61, 67]
```

Now there's another question. We could compute the 100<sup>th</sup> prime as above, since we knew that there are more than 100 primes below 1000. How do we directly compute the  $n^{\text{th}}$  prime for a specified  $n$ ? The trick is to keep on adding consecutive primes to a list, till we have  $n$  of them and then take the  $n^{\text{th}}$  entry. of the list. And this process can be defined as a Python function as below:

```
# Function to compute the nth prime

def prime(n):
    prime_list=[2]
    test_number=3
    while len(prime_list)<n:
        for i in range(2,int(test_number**(0.5))+1):
            if test_number%i==0:
                break
        else:
            prime_list.append(test_number)
            test_number=test_number+1
    return(prime_list[n-1])
```

Saving this in myfunctions.py, we can get the prime at any position:

```
>>> from myfunctions import prime
>>> prime(100)
541
>>> prime(1000)
7919
>>> prime(10000)
104729
```

As another example of lists, recall how we wrote a program to display the prime factors of a specified number. We can modify this to display the factors as a list, rather than one below the other as in the earlier case:

```
# List of prime factors of a number

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

prime_factors=[]

i=2
while n>1:
    if n%i==0:
        prime_factors.append(i)
        n=n//i
    else:
        i=i+1
print(prime_factors)
```

Running this gives, for example

```
Type number : 360
[2, 2, 2, 3, 3, 5]
```

As another example of lists in a different context, consider the computation of the arithmetic mean of some numbers. This is done by creating a list of numbers and dividing the sum of these numbers by how many numbers there are. We know how to create a list in Python and also how to find the number of entries in it. Finally, we have `sum` function for lists in Python, which computes the sum of the entries:

```
>>> def am(L):
...     return(sum(L)/len(L))
...
>>> am([3,2.5,1.62,9.34,6.732])
4.6384
```

There are some interesting things to note here. First is that we can use this to compute the arithmetic mean of *any set* of numbers, regardless of how many there are. This is not available in most other programming language, where we'll have to first fix the size of an array (the generic term for list-like objects in programming) to use it.

Secondly, note that we didn't explicitly say in the definition that `L` is to be a list. The expressions `sum(L)` and `len(L)` implies it is to be some object for which such constructs are valid, and when we ask to compute `am([3,2.5,1.62,9.34,6.732])`, then the square brackets in `[3,2.5,1.62,9.34,6.732]` indicates a list. To understand this, type the function argument without the square brackets `[]` and see what happens:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: am() takes 1 positional argument but 5 were given
```

It's the last line

```
TypeError: am() takes 1 positional argument but 5 were given
```

that tells us what's wrong. The definition of the `am` function involves only one variable `L`, and with square brackets, Python interprets `[3,2.5,1.62,9.34,6.732]` as a *single list* to be used as `L`. But without the square brackets, it means the function has to operate on *five numbers*.

In the next section, we'll see how we can write a function to compute the arithmetic mean, which does not require us to type the square brackets.

## 5.1 Strings

Another data type in Python is *string*. We had talked about strings in some of the earlier section. Here we take a detailed look at them.

As you may recall, a string just a finite sequence of *characters*, meaning the various letters and symbols we can type in our keyboards. And to indicate that we are referring to a string, rather than the name (or value) of a variable, we must enclose it within quotes. We have also noticed that a string of digits can be converted to the natural number formed by these digits in order, using the function `int` and to a decimal number using `float`, as in

```
>>> int('478')
478
>>> int('00478')
478
>>> float('478')
478.0
>>> float('00478.23')
478.23
```

Note that in this conversion, any leading zeros are stripped off to get a meaningful number.

On the other hand, we can convert a natural number to just the string of its digits using the `str` function:

```
>>> n=2500
>>> s=str(n)
>>> s
'2500'
```

Another interesting thing is the operator `+`, used to add numbers can be applied to strings to join them end to end (the technical term for this operation being *concatenation*):

```

>>> s='abc'
>>> t='pqr'
>>> s+t
'abcpqr'
>>> s='123'
>>> t='456'
>>> s+t
'123456'
>>> int(s)+int(t)
579

```

Let's have a math problem. First look at these:

$$\begin{aligned}
 1 + 2 + 3 + 4 + 5 &= 15 \\
 2 + 3 + 4 + 5 + 6 + 7 &= 27 \\
 4 + 5 + 6 + \dots + 28 + 29 &= 429
 \end{aligned}$$

Question is whether we can find some more pairs of natural numbers like these, that is, pairs for which the sum of all the natural numbers from the smaller to the larger is just those two numbers pasted together.

Let's consider the steps through which we find such pairs:

1. First we fix a number **b** as the bound for our search. That is we will test only pairs of numbers below **b**
2. We assign variables **s** for the smaller number of the pair and **l** for the larger, so that **s** has to be stepped through 1, 2, 3, ..., **b** and for each value of **s**, the variable **l** has to be stepped through 1, **l**+1, ..., **b**.
3. We will have to find the sum of all natural numbers from **s** to **l**. This can be done by the Python function `sum(range(s,l+1))`
4. To find the number got by pasting **l** to the end of **s**, we do the following:
  - (a) Convert **s** and **l** to strings using `str` function
  - (b) Form their concatenation using `+`
  - (c) Turn this into a number by the `int` function
5. Finally we check whether the sum is the concatenated number using `==` and if so, print **s** and **l**

Putting these ideas into practice, we write the program as below:

```

# Program to list a special kind of number pairs

def catnum(m,n):
    return int(str(m)+str(n))

b = int(input('Check up to : '))

for s in range(1,b+1):
    for l in range(s,b+1):
        if sum(range(s,l+1))==catnum(s,l):
            print("Sum of integers from", s , "to", l, "is", catnum(s,l))

```

We get the following pairs below 1000:



```

Sum of integers from 1 to 5 is 15
Sum of integers from 2 to 7 is 27
Sum of integers from 4 to 29 is 429
Sum of integers from 7 to 119 is 7119
Sum of integers from 13 to 53 is 1353
Sum of integers from 18 to 63 is 1863
Sum of integers from 33 to 88 is 3388
Sum of integers from 35 to 91 is 3591
Sum of integers from 78 to 403 is 78403
Sum of integers from 133 to 533 is 133533
Sum of integers from 178 to 623 is 178623

```

Another interesting thing is that Python strings share some properties of lists. For example, just as we can access the entries of a list `L` as `L[0]`, `L[1]`, `L[2]`, ..., so can we access the characters of a string. For example:

```

>>> s='mathematics'
>>> s[0]
'm'
>>> s[8]
'i'

```

Again, we can get a part of a string as in the case of a list:

```

>>> s='mathematics'
>>> s[2:5]
'the'

```

Apart from the two parameters specifying the start and finish of the entries (or characters), we can also add a third parameter, indicating the *step* (or jump). For example:

```

>>> L=[x**2 for x in range(1,16)]
>>> L
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225]
>>> L[2:10:3]
[9, 36, 81]
>>> s='abcdefghijklmno'
>>> s[1:14:2]
'bdfhjln'

```

Thus in `L[2:10:3]`, Python gives `L[2]`, `L[5]`, `L[8]` (that is, the entries with indices 2,  $2+3=5$ ,  $5+3=8$ ), which are the squares of multiples of 3 in `L`. Similarly, in `s[1:14:2]`, we get the characters of the string at odd number positions.

To go from the first to a certain position, we can do away with the 0 at the beginning and type for example `L[:10:3]` to get entries from the first to the ninth, stepping through every third entry. Similarly, if we want up to the last entry, we need not specify its position. Thus for `L` and `s` as above, we can do things like this:

```

>>> L[:10:3]
[1, 16, 49, 100]
>>> s[:12:2]
'acegik'
>>> L[3::2]
[16, 36, 64, 100, 144, 196]
>>> s[4::5]
'ejo'

```

We can also get entries or characters in the reverse order by making the step negative. For example, with the same `L` and `s` as above, we have

```
>>> L[10:2:-3]
[121, 64, 25]
>>> s[14:3:-2]
'omkige'
```

So, what happens if we give `s[::-1]`?

```
>>> s[::-1]
'onmlkjihgfedcba'
```

This gives a neat trick to reverse the digits of a number:

```
>>> n=1357
>>> s=str(n)
>>> t=s[::-1]
>>> m=int(t)
>>> m
7531
```

These ideas can be used to verify an amusing fact in number theory. Take a three-digit number with not all digits the same and form the number got by reversing the digits. Subtract the larger from the former. For example, starting with 183, this gives

$$381 - 183 = 198$$

Reverse the digits again and this time add:

$$198 + 981 = 1089$$

Do this for some other numbers. Do you get 1089 always? Sometimes you get another number (what number?) But this and 1089 are the only possibilities. How do we write a program to check this for several numbers using Python?

We have seen how we have seen just now seen how we can reverse the digits of a number in Python. We can write this as a function:

```
def rev_number(n):
    s=str(n)
    t=s[::-1]
    m=int(t)
    return(m)
```

Next we have to subtract the smaller of `n` from the larger. We can use the `max` and `min` functions in Python (which as their names imply finds the maximum or minimum of specified numbers) to do this:

```
l=max(n,rev_number(n))
m=min(n,rev_number(n))
```

Now we can write the program to check the result of the operations above, displaying all intermediate steps. And since we want to check these for several numbers, we use our earlier trick of including an `option` string, which asks us every time whether to continue or quit:

```
# A program to verify a number property

def rev_number(n):
    s=str(n)
    t=s[::-1]
    m=int(t)
    return(m)

option='yes'

while option=='yes':
    n=int(input("Enter a three digit number with not all digits same :"))
    print("Number reversed is", rev_number(n))
    l=max(n,rev_number(n))
    s=min(n,rev_number(n))
    d=l-s
    print("Differnce is", l, '-', s, '=', d)
    print("This reversed is", rev_number(d))
    l=max(d,rev_number(d))
    s=min(d,rev_number(d))
    print("Their sum is", d, '+', rev_number(d), '=', l+s)
    print('Do you wish to continue?')
    option=input('Type yes or no : ')

```

Using this to check a couple of numbers, we get something like this:

```
Enter a three digit number with not all digits same :387
Number reversed is 783
Differnce is 783 - 387 = 396
This reversed is 693
Their sum is 396 + 693 = 1089
Do you wish to continue?
Type yes or no : yes
Enter a three digit number with not all digits same :280
Number reversed is 82
Differnce is 280 - 82 = 198
This reversed is 891
Their sum is 198 + 891 = 1089
Do you wish to continue?
Type yes or no : n

```

Try it with other numbers see if you get a number other than 1089. Then you can think about the conditions under which we get one answer or the other. And finally think about why this happens!

We look at another number theoretic curiosity discovered by the Indian school teacher, D. R. Kaprekar:

- (i) Take any four-digit number with at least two digits different
- (ii) Rearrange the digits to get the largest and smallest number that can be formed with the same digits
- (iii) Subtract the smaller number from the larger
- (iv) Repeat the process with the number got as the difference
- (v) The process terminates with the number 6174 in at most 8 iterations

For example, consider the number 7138. Kaprekar routine for this is

$$8731 - 1378 = 7353$$

$$7533 - 3357 = 4176$$

$$7641 - 1467 = 6174$$

which gives 6174 in three iterations. Note that if we start with 6174 itself, then we get it back in a single iteration, given by the last of the equations above.

To write a Python program to do the verification for us, we must have a Python routine to change any number to the largest and smallest numbers that can be formed using the same digits. It is easily seen that the largest number that can be formed by any set of digits is got by writing digits in the descending order, and the smallest by writing them in the ascending order. Now Python has a `sorted` function for lists which arranges the entries in ascending order; but applied to strings, this function gives a *list* in ascending order and not a string:

```
>>> l=[7,1,9,6]
>>> sorted(l)
[1, 6, 7, 9]
>>> s='64123'
>>> sorted(s)
['1', '2', '3', '4', '6']
```

So, we must have a method to convert a list into a string. We know that the `str` function converts a number into a string, but the same command applied to a list produces something else:

```
>>> l=['1', '2', '3', '4', '6']
>>> str(l)
"['1', '2', '3', '4', '6']"
```

Thus `str(l)` is the string consisting of the characters, [, ', 1 and so so on. It is seen clearly by asking for the character at a specific place of `str(l)`:

```
>>> l=['1', '2', '3', '4', '6']
>>> s=str(l)
>>> s[0]
 '['
>>> s[1]
 "'"
>>> s[2]
 '1'
```

So, the trick to get a string of the digits in the list, the trick is to start with an empty string and concatenate the entries in the list one by one:

```
>>> l=['1', '2', '3', '4', '6']
>>> s=''
>>> for x in l:
...     s=s+x
...
>>> s
'12346'
```

Now we can write a function in Python which gives the smallest number which can be formed using the digits of a given number `n`, through the following steps:

- (i) Convert `n` to the string of its digits by `str(n)`

- (ii) Form the list of the of the digits of `n` in ascending order, as strings, by `sorted(str(n))`
- (iii) Start with an empty string and attach each entry of this list to it in succession, to get the smallest number as a string
- (iv) Convert this string into the corresponding number using `int` again

To get the largest the largest number, all that we need to do is to reverse the string got in the last but one step, using the `[::-1]` construct.

Now to verify the Kaprekar routine, we will have to repeatedly find the largest and smallest numbers for different digits, so write functions to compute them:

```
def small_num(n):
    ascend_list=sorted(str(n))
    ascend_string=''
    for digit in ascend_list:
        ascend_string=ascend_string+digit
    return(int(ascend_string))

def large_num(n):
    ascend_list=sorted(str(n))
    ascend_string=''
    for digit in ascend_list:
        ascend_string=ascend_string+digit
    return(int(ascend_string[::-1]))
```

These functions in action are shown below:

```
>>> def small_num(n):
...     ascend_list=sorted(str(n))
...     ascend_string=''
...     for digit in ascend_list:
...         ascend_string=ascend_string+digit
...     return(int(ascend_string))
...
>>> small_num(8172)
1278
>>> def large_num(n):
...     ascend_list=sorted(str(n))
...     ascend_string=''
...     for digit in ascend_list:
...         ascend_string=ascend_string+digit
...     return(int(ascend_string[::-1]))
...
>>> large_num(8172)
8721
```

We also have to find the difference of the largest and smallest numbers and change the original number to this difference at every stage, so that we also define:

```
def diff_num(n):
    return(large_num(n)-small_num(n))
```

Again, we want the program to stop when the difference reaches the number 6174; that is we want it to continue till the difference is not equal to this number. The Python symbol for  $\neq$  is `!=`. So, we wrap our iteration under a `{while diff_num(n)!=6174}`

Also, if we want the program itself to say how many iterations were done, we can assign a variable to count them, increasing it by one every time the loop completes a cycle of operations. And since we want to check this for several numbers, we include a continue or quit option as done earlier

Putting these all together, we finally have the program to verify the Kaprekar process:

```
# Program to give the Kaprekar iterations of a specified number

def small_num(n):
    ascend_list=sorted(str(n))
    ascend_string=''
    for digit in ascend_list:
        ascend_string=ascend_string+digit
    return(int(ascend_string))

def large_num(n):
    ascend_list=sorted(str(n))
    ascend_string=''
    for digit in ascend_list:
        ascend_string=ascend_string+digit
    return(int(ascend_string[::-1]))

def diff_num(n):
    return(large_num(n)-small_num(n))

option='yes'

while option=='yes':
    n=input("Type a four-digit number not all digits same: ")
    print("Larget number =", large_num(n))
    print("Smallest number =",small_num(n))
    print(large_num(n), "-", small_num(n), "=", diff_num(n))
    count=1
    while diff_num(n)!=6174:
        n=diff_num(n)
        print("Larget number =", large_num(n))
        print("Smallest number =",small_num(n))
        print(large_num(n), "-", small_num(n), "=", diff_num(n))
        count=count+1
    print("Number of iterations is", count)
    option=input('Type yes or no : ')
```

And here's a sample run:

```
Type a four-digit number not all digits same: 1234
Larget number = 4321
Smallest number = 1234
4321 - 1234 = 3087
Larget number = 8730
Smallest number = 378
8730 - 378 = 8352
Larget number = 8532
Smallest number = 2358
8532 - 2358 = 6174
Number of iterations is 3
```