6 Lists

To illustrate the next idea, let's consider the program to test whether a natural number is a prime or not, which we did earlier. Suppose now want a list of all primes below a certain natural number. This can be easily done as follows:

```
# A program to produce all primes below a specified number

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

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

If we run this program for a small limit such as 20, you get an output like this

```
List primes upto : 20
2
3
5
7
11
13
17
19
```

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) instance of a Python *list*. Look at this program:

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

print()

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

print()

P = []

for n in range(2,l+1):
 for x in range(2,int(n**(0.5)+1)):
  if n % x == 0:
   break
 else:
  P.append(n)

print()

print(P)

print()
```

As mentioned before, the empty `print()` commands are just to make some blank lines in the output to make it more readable. The new concept is the command

```
P = []
```

This creates an empty *list* in Python and names it P. And after the code for choosing the primes which we have seen earlier, the line

```
P.append(n)
```

asks the program to include in the list (*append* to the list) P all those selected numbers. Finally once all numbers are tested and the primes selected and included in the list, we give the command to *print* the list. Run it with the limit set to 1000 and see.

We can have some more fun with this. Suppose we are only interested in the *number* of primes below a specified number. All we need to do is replace the line `print(P)` with

```
print(len(P))
```

to get the *length* of the list P, that is the number of entries in the list. For example if we run this with limit 1000, the output would be `168`, showing there are 168 primes less than 1000.

Again, note that by the very process of appending numbers to P, the numbers are in ascending order. This means if we want the 100^{th} 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 $P(0), P(1), P(2), \dots$, which means the n^{th} entry of the list is $P[n-1]$ (and not $P[n]$.) So to get the 100^{th} prime, we write

```
print(P[99])
```

and this outputs `541`, which is the hundredth prime.

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^{th} to the 20^{th}. This can be done by changing the `print(P)` in the code to

```
print(P[9:19])
```

and this gives the output

```
[29, 31, 37, 41, 43, 47, 53, 59, 61, 67]
```

Now there's another question. We could compute the 100^{th} prime as above, since we knew that there are more than 100 primes below 1000. How do we directly compute the n^{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 maximum 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(max(prime_list))
```

The only new thing in this code is `max(prime_list)` which, as the name implies, gives the largest number in the list. Now saving this in a file and importing it, we can find the various terms of the prime sequence

```
>>> from myfunctions import prime
>>> prime(1000)
7919
>>> prime(10000)
104729
>>> prime(100000)
1299709
```

the last one taking almost half a minute in my system.

A better idea is to first define the prime-check as a function and then use it in the definition of the n^{th} prime:

```
# Function to check for primality

def is_prime(n):
 if n==1:
  conclusion=False
 elif n==2:
  conclusion=True
 elif n>2:
  for i in range(2, int(n**0.5) + 2) :
   if (n%i)==0:
    conclusion=False
    break
  else:
   conclusion=True
 return(conclusion)

# Function to find the nth prime

def prime(n):
 prime_list=[2]
 test_number=3
 while len(prime_list)<n:
  if is_prime(test_number)==True:
   prime_list.append(test_number)
  test_number=test_number+1
 return(max(prime_list))
```

With this we have

```
>>> from myfunctions import is_prime, prime
>>> prime(1000)
7919
>>> prime(10000)
104729
>>> prime(100000)
1299709
>>>
```

Note that in the definition of the first function, the lines

```
coclusion=False

conclusion=True
```

are not assignments of *strings* ‘False’ or ‘True’, to the variable conclusion (no quotes in the code!) but assignments of what are known as Boolean variables which are built in. For example consider these examples:

```
>>> 2+2==5
False
>>> 2+2==4
True
>>> 2+2==1+3
True
>>> 2+2>3
True
```

As another example, see this:

```
>>> A=True
>>> B=True
>>> C=False
>>> (A and B) or C
True
>>> (A or B) and C
False
```

Coming back to lists, here's a program to list all prime factors of a specified number, each repeated as many times as it occurs:

```
# Prime factorization of a given number

print()

n = int(input("Enter Number : "))

L = []

while n > 1:
 i = 2
 while i <= n:
  if n % i == 0:
   L.append(i)
   n = n / i
  else:
   i = i + 1

print()

print(L)

print()
```

Here's an example output:

```
Enter Number : 360

[2, 2, 2, 3, 3, 5]
```

Do you see why it gives only *prime factors*?

Another point about lists is that we can specify a list by a condition, rather than giving the actual entries (as we do for sets in math). For example, see this:

```
>>> L=[x**2+x for x in range(1,11)]
>>> L
[2, 6, 12, 20, 30, 42, 56, 72, 90, 110]
```

Also instead of iterating over a `range`, we can iterate over any list. For example, suppose we have a function (in our myfunctions.py) to create the list of primes up to a number as below:

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

Using this, we can create a list of primes up to a number as in:

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

Now every odd number is of the form either $4n + 1$ or $4n - 1$ and suppose we want to extract from the list P above, those of the form $4n + 1$. We can do this as:

```
>>> L=[x for x in P if x%4==1]
>>> L
[5, 13, 17, 29, 37, 41]
```

(A curious reader may have a doubt about the above code. Why create a list `P=primes_below(50)`? Why not directly use the `prime_list` defined in the function? The point is that the *variable* `prime_list` is defined only *locally* within the function `primes_below` and is not *globally* available outside definition of the function. Try typing `prime_list(50)` and see!)

As yet another example of using lists, consider the problem of finding the arithmetic mean of some numbers. This can be done using lists using the definition below:

```
# Function to calculate the arithmetic mean of a list of numbers

def am(L):
 return(sum(L)/len(L))

```

The only new thing here is the expression `sum(L)`, which as the name implies gives the sum of the numbers in the list L. A typical example is like this:

```
>>> from myfunctions import am
>>> 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 form 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 this without the square brackets `[]` and see what happens:

```
>>> from myfunctions import am
>>> am(3,2.5,1.62,9.34,6.732)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: am() takes 1 positional argument but 5 were given
```

The definition of the `am` function involves only one variable `L` and with 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*.

Here are some exercises.

1. We have seen that we can find the sum of numbers in a list by the operator `sum`. But Python has no such built-in operator to find the product of numbers in a list. Write a program to compute the product of numbers in a given list

2. Start with any natural number. If it is even divide by 2 and if it is odd, multiply by 3 and add 1. Repeat these operations, depending on whether the result is even or odd at every stage. The so called Collatz conjecture states that for any natural number, this process will eventually terminate at 1. Write a program which accepts a natural number as input and apply these procedures repeatedly and outputs the following:

   - The list of all numbers obtained at each stage, in the order of occurrence
   - How many numbers this list contains
   - The largest number reached during this process

   Check this for 27.

3. It is well known any natural number other than a power of 2 can be written as a sum of consecutive of natural numbers. For example

$$\begin{aligned}
9 &= 4 + 5 \\
10 &= 1 + 2 + 3 + 4 \\
11 &= 5 + 6 \\
12 &= 3 + 4 + 5 \\
13 &= 6 + 7 \\
14 &= 2 + 3 + 4 + 5 \\
15 &= 1 + 2 + 3 + 4 + 5
\end{aligned}$$

   Write a program which accepts a natural number as input, and gives the output

   - `The number cannot be written as a sum of consecutive numbers`, if the number input is a power of 2
   - The list of consecutive numbers which gives this number as sum, if the number is not a power of 2