7 Strings

We've talked about *strings* in Python in passing on several occasions. Here we'll take a more detailed look on this data type and see how it can be used in some mathematical problems.

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, double or single. (Recall that unlike the usual practice in mathematics, names of python variables can have more than one letter.) For example:

```
>>> s=abc
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'abc' is not defined
>>> s='abc'
>>> s
'abc'
>>> t="xyz"
>>> t
'xyz'
```

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`, as in:

```
>>> s='125'
>>> int(s)
125
>>> s='00134'
>>> int(s)
134
```

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'
>>> s[1]
'5'
>>> s[3]
'0'
```

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
```

(Thus in Python, data types can be changed within a program. Technically this is described as Python being *dynamically typed.* Also, the same operation can have different meanings depending on the type of objects it acts upon. This is technically called *operator overloading.* Just a couple of technical jargon, to impress your programmer friends!)

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

$$1 + 2 + 3 + 4 + 5 = 15$$
$$2 + 3 + 4 + 5 + 6 + 7 = 27$$
$$4 + 5 + 6 + \cdots + 28 + 29 = 429$$

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.

From what we have said about strings above, the way to do this in Python must be obvious. For each pair of numbers, we must find the sum of numbers between them and compare it with their concatenation:

```
# Program to list a special kind of number pairs

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

print()

l = int(input("Type how far to check : "))

print()

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

print()
```

A typical output is like this:

```
Type how far to check : 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 $L[0], L[1], L[2], \ldots$, so can we access the characters of a string. For example:

```
>>> s='mathematics'
>>> s[0]
'm'
>>> s[5]
'm'
```

Again, in lists as well as strings, we get a part of them as follows:

```
>>> L=[2*x for x in range(1,11)]
>>> L
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>>> L[2:5]
[6, 8, 10]
>>> s='mathematics'
>>> s[5:8]
'mat'
```

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'
```

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 L and s as in the above example, 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
```

Time for another math idea. First do it by hand. 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. Suppose we want to check this for several numbers using Python. A program to do this can be something like this:

```
# A program to verify a number property

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

print()

n=int(input("Enter a three digit number with not all digits same :"))

print()

print("Number reversed is", rev_num(n))

large_num=max(n,rev_num(n))
small_num=min(n,rev_num(n))

print()

print("Differnce is", large_num, '-', small_num, '=', large_num-small_num)

print()

d=large_num-small_num

print("This reversed is", rev_num(d))

print()

large_num=max(d,rev_num(d))
small_num=min(d,rev_num(d))


print("Their sum is", d, '+', rev_num(d), '=', large_num+small_num)

print()
```

The only new things are the min and max functions, which as the names imply, compute the maximum and minimum of specified numbers. An example run is given below:

```
Enter a three digit number with not all digits same :368

Number reversed is 863

Differnce is 863 - 368 = 495

This reversed is 594

Their sum is 495 + 594 = 1089
```

(You can also try to give a mathematical proof of this fact!)

Now just as we can convert an integer to a string using the int function, we can form a list of the characters in a string using the function list:

```
>>> s='2901'
>>> list(s)
['2', '9', '0', '1']
```

But if we try to do the reverse, that of converting a list to string formed by it entries using the str function, we are in for a surprise:

```
>>> L=[3,9,1,0]
>>> str(L)
'[3, 9, 1, 0]'
```

(Can you see what happened here?) The workaround here is to write an iterative loop adding the entries of the list one by one to form a string:

```
>>> L=[3,9,1,0]
>>> s=''
>>> for x in L:
...  s=s+str(x)
...
>>> s
'3910'
```

Again, there is a function called sorted which applied to a list, arranges the entries in ascending order. Applied to a string it gives a *list* of characters in ascending order:

```
>>> L=['cat','tiger','lion','bear']
>>> sorted(L)
['bear', 'cat', 'lion', 'tiger']
>>> M=[3,0,8,4,1]
>>> sorted(M)
[0, 1, 3, 4, 8]
>>> s='xpayhs'
>>> sorted(s)
['a', 'h', 'p', 's', 'x', 'y']
>>> t='87312'
>>> sorted(t)
['1', '2', '3', '7', '8']
```

With all this under our belt, let's look at an apparently simple problem—to find the smallest and largest integer that can be formed using some specified digits. Before taking a peek at the code below, try to do it yourselves.

```
# Program to find the smallest and largest number using specified digits

print()

digit_string=input("Type digits without any separation:")

ascend_list=sorted(digit_string)
ascend_string=''

for digit in ascend_list:
 ascend_string=ascend_string+digit
min_number=int(ascend_string)

descend_string=ascend_string[::-1]

max_number=int(descend_string)

print()

print("Smallest number is", min_number)
print("Largest number is", max_number)

print()
```

An example run of the program is given below:

```
Type digits without any separation:81006523

Smallest number is 123568
Largest number is 86532100
```

To illustrate another feature of strings, recall our earlier function to compute the arithmetic mean of a list of numbers. We did this at the Python console. Suppose we want a program which accepts some numbers separated by commas as input and gives their arithmetic mean as output. Remembering that the input function makes anything we type at the prompt as a *string* and that list converts a string to a list, our first attempt at this would be something like

```
s=input("Type numbers separated by commas :")
L=list(s)
M = [float(L[i]) for i in range(0,len(L))]
a = sum(M) / len(M)
print("The arithmetic mean is", a)
```

But if we run this and input some numbers separated by commas, we get an error:

```
Type numbers separated by commas :2,4,9
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    M = [float(L[i]) for i in range(0,len(L))]
NameError: name 'L' is not defined
[krshnan programs]$ python3 test.py
Type numbers separated by commas :1,2,3
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    M = [float(L[i]) for i in range(0,len(L))]
  File "test.py", line 5, in <listcomp>
    M = [float(L[i]) for i in range(0,len(L))]
ValueError: could not convert string to float: ','
```

The clue to debugging (a programming jargon finding errors in a program) is the last line:

ValueError: could not convert string to float: ','

To understand this, let's see what exactly is stored as the string s and the list L in the above code, by commenting out some lines (don't *delete* these lines, we may need them later!) and just printing s and L:

```
s=input("Type numbers separated by commas :")

L=list(s)
#M = [float(L[i]) for i in range(0,len(L))]
#a = sum(M) / len(M)

#print("The arithmetic mean is", a)

print(s)
print(L)
```

This gives:

```
Type numbers separated by commas :1,2,3
1,2,3
['1', ',', '2', ',', '3']
```

Do you see what is happening? The characters of the string are not only the three numbers in it but also the two commas separating them. So L includes these three commas also, and when we try to convert the entries of L to float, Python flashes the error that this cannot be done for the first comma it encounters.

The way out is to use the split method. Try this at the console:

```
>>> s='1,2,3'
>>> list(s)
['1', ',', '2', ',', '3']
>>> s.split(",")
['1', '2', '3']
>>> s='1+2+3'
>>> list(s)
['1', '+', '2', '+', '3']
>>> s.split("+")
['1', '2', '3']
>>> s='malayalam'
>>> s.split('a')
['m', 'l', 'y', 'l', 'm']
>>> s='This is a sentence'
>>> s.split(" ")
['This', 'is', 'a', 'sentence']
```

Thus in any string with some character repeated, we can get a *list* of characters (or substrings) got from the string by *splitting* it at the repeated character (called a *separator*). Thus in the examples the separators are

- The comma in the first
- The plus sign in the second
- The letter a in the third
- The white space in the fourth

Now the way to correct our code for computing arithmetic mean is clear. This is it:

```
# This is a program to calculate the arithmetic mean of given list of numbers

print()

s = input("Type numbers separated by commas : ")
L = s.split(",")
M = [float(L[i]) for i in range(0,len(L))]
a = sum(M)/len(M)

print()

print("The arithmetic mean is", a)

print()
```

And this is a sample run:

```
Type numbers separated by commas : 2.5,1,3.2,4

The arithmetic mean is 2.675
```

Somewhat opposite to split is the join method. Acting on a string, this joins the *characters* of the string with a chosen character in between each:

```
>>> s='123'
>>> '+'.join(s)
'1+2+3'
```

This also works on a list of strings:

```
>>> L=['1','2','3']
>>> '+'.join(L)
'1+2+3'
>>> L=['bat','cat','eat']
>>> '+'.join(L)
'bat+cat+eat'
```

This gives a way to get a nicely formatted output from our program to list the prime factors, repeated according to their occurrence. Recall that in the last section, we wrote the code:

```
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(L)
```

to get a list of prime factors like

```
Enter Number : 360

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

Wouldn't it be nicer if the output is written as a multiplication, instead of a mere list? Here's the way to go!

```
# Prime factorization of a given number

print()

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

store_number = number
factor_list = []

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

print()

factor_list_strings = [str(i) for i in factor_list]

print(store_number, "=", " x ".join(factor_list_strings))

print()
```

Running this gives for example,

```
Enter Number : 360

360 = 2 x 2 x 2 x 3 x 3 x 5
```

Note that the joining character is `" x "` and not `"x"`. (What difference does it make?)

That's all about strings (for the time being!) Now here are some exercises you can try

1. Write a program to list all palindromic primes below a specified number
2. D. R. Kaprekar, an Indian school teacher discovered the following fact:
   (a) Take any four-digit number with at least two digits different
   (b) Rearrange the digits in ascending order and then in descending order to get two other numbers
   (c) Subtract the smaller number from the larger
   (d) Repeat the process with difference
   (e) The process terminates with the number 6174 in at most 8 iterations

   (If we apply this process to 6174, we get the same number!) Write a program which accepts a four digit numbr with not all digits the same and gives as output each stage of the Kaprekar routine, consisting of

   - The larger number
   - The smaller number
   - The difference

   and also the number of iterations
3. Write a program which accepts as input the components of two vectors as comma separated numbers and gives the angle between them in degrees as output