Lesson 6: Strings, Lists and `for` loop

A string is a **sequence of characters.**

To create a string in Python you can assign a string to a variable as in:

```
primary_colours = 'red orange yellow green blue indigo violet'

colours = 'pink, white, black, brown, grey'

s = 'hello’
```

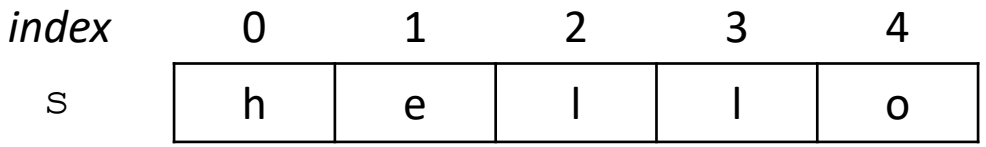
We can access any element of a string, by using its position in the string.

This is called its **index** or **subscript**.

We put the index in [] brackets after the name of the string.

A string can be visualized as shown below for a string `s`:

```
s = 'hello '
```



We can access any character in the string `s` using its **index**:

```
S[0] is 'h'

S[1] is 'e'

S[4] is 'o'
```

The first character of a string is always at index 0

We can also use `input` to create a string:

```
address = input('\nEnter your address on 1 line')
```

String Length - `len`

The **len** function gives us the length of a string or list e.g.

```
l =  len('abcd')
```

gives `l` the value 4.

```
s = 'abcdef'
lens =  len( s )        # lens = 6 in this case
print (f'length of s: {lens}\n’)
```

outputs

```
length of s: 6
```

Note: Because we start strings at **index 0**, the last character in any string is **always at** index **(`length_of_string - 1`)**.

Example L6.1

Write a program to output the characters in a string on separate lines.

```
# str.py: Output each characters on a newline

string = 'abc'

length = len(string) # 3 in this case

i = 0
while ( i < length ):
  print( string[i] ) # displays elements 0, 1 and 2
  i = i + 1
```

Output

```
a
b
c
```

Concatenating Strings

We use the **+** operator to add one string on to the end of another string – this is called **concatenation**.

```
s1 = 'abc'
d1 = '456'

s2 = s1 + d1                # s2 is 'abc456'
print ( s2 )

print ( s1 + ' ' + d1 )
```

Output

```
abc456
abc 456
```

Example L6.2

Write a program to read a name and 3 lines of address. The program displays the name and 3 lines of address on a **single** line.

```
name  = input('Enter name: ')
addr1  = input('Enter Address line 1:')
addr2  = input('Enter Address line 2:')
addr3  = input('Enter Address line 3:')

print (f'\n' + name + ' ' + addr1 + ' ' + addr2 + ' ' +
addr3)
```

Output

```
Enter name: Super Man
Enter Address line 1: Time Square
Enter Address line 2: New York
Enter Address line 3: USA

Super Man Time Square New York USA
```

Other operations on Strings: `upper` and `lower`

Python allows you perform many other operations on strings and we only look at a few of them in this Handbook. We use a different mechanism to carry out these operations – it is called using **methods** from a form of programming called object-oriented programming.

In this form of programming, a string is regarded as an **object** and to carry out an operation on an object you perform a method on the object.

For example, we often want to convert all the alphabetic characters (A to Z, a to z) in a string to uppercase (A to Z) or to lowercase (a to z).

Python provides the methods **`upper`** and **`lower`** to do these conversions. For example, the code below will convert any uppercase characters in the string `s` to lowercase and assign the new string to the variable `t`:

Example L6.3

```
s = 'ABC def 123 +^*'
t = s.lower()

print (f' s is: {s} \n')
print (f' t is: {t} \n')
```

Output:

```
s is: ABC def 123 +^*
t is: abc def 123 +^*
```

The statement `t = s.lower()` converts all uppercase letters in `s` to lowercase and stores them in `t`.

Note: The string `s` is unchanged. You cannot change the elements in a string in Python we say that strings are **immutable** in Python e.g. you **cannot** use `s[0] = 'x'` to change an element of a string.

Example L6.4: Convert to uppercase

`t = s.upper()` converts all lowercase letters in `s` to uppercase and stores them in `t`

```
s = 'ABC def 123 +^*'
t = s.upper()

print (f' s is: {s} \n')
print (f' t is: {t} \n')
```

Output:

```
s is: ABC def 123 +^*

t is: ABC DEF 123 +^*
```

Example L6.4: Write a guessing game program to ignore the case of the user guess

```
secret = 'Blue'
guess = ' '
num_chances = 1

secret = secret.lower()   # convert to lowercase
while (guess != secret) and ( num_chances <= 3 ) :
     guess = input('Guess the secret word:  ')
     guess = guess.lower()   # convert to lowercase
     if guess != secret:
        print('\nWrong guess: ', guess)
        num_chances = num_chances + 1
    else:
        print('Well done !')
if num_chances > 3:
    print('Sorry you have used all of your guesses')
    print('The secret word was: ', secret)
```

Output:

```
Guess the secret word:  BLUE
Well done !
```

Other String operations: `in`

The **`in`** operator allows us check if a string (1 or more characters) is part of another string. The condition **`e in str`** is `True` if `e` is contained in the `str` and `False` otherwise:

```
str  = 'bread gums blue black'
if 'gum' in str:
       print(f'Yes gum is in {str}')
if 'k' in str:
       print(f'k  is in {str}')
if 'car' in str:
       print('Yes car in string')
else:
       print('Car not in string')
```

Output

```
Yes gum is in bread gums blue black
Yes k  is in bread gums blue black
Car not in bread gums blue black
```

More String operations: `isupper, islower, isdigit`

The method `isupper` returns True if the string is all uppercase (A to Z).

The method `islower` returns True if the string is all lowercase (a to z).

Numbers, symbols and spaces are **ignored** by `isupper` and `islower` - only alphabet characters are checked.

The method `isdigit` returns true if the string is **all** digits (0 to 9).

The 3 methods above can be applied to **single** character or multi character strings.

```
a ='Hello World!'
b = 'MAN. UTD’
c = '456’
d = ‘hello world!'

print( a.isupper())
print( b.isupper())
print( c.isdigit())
print( d.lower())
```

Output:

```
False
True
True
True
```

Write a program to read text from the user and display whether there is an uppercase character in the text.

```
# text.py: Check for an uppercase character in the input

text = input('Enter any string: [Q/q to quit] ')

while text != 'q' and text != 'Q':
    found = False
    i = 0
    while ( i < len (text) ) :
        if text[i].isupper() :      # check for uppercase character
            found = True            # found an uppercase character
            break                   # leave loop if found uppercase
        else:
            i = i + 1

    if found == True:
        print (f'Uppercase found in: {text} ')
    else:
        print(f'No uppercase found in {text}’)

    text = input('Enter any string: [Q/q to quit] ')
```

Output:

```
Enter any string: [Q/q to quit] asdf
No uppercase found in asdf
Enter any string: [Q/q to quit] Abc123
Uppercase found in: Abc123
Enter any string: [Q/q to quit] q
```

Lists

We encounter examples of lists in our daily lives:

shopping list of things to buy

list of students in a class

list of employees in a company

Python provides us with a ***list* data type** to handle lists.

A data type refers to the type of value a variable has. We have already used the data types *integers*, *floats* and *strings* in our programs.

It is easy to create and use lists in Python. We give the list a **name** and we access the items in the list using an **index** (subscript) in the same way that we used an index in accessing the elements of a string.

List Examples

A list of items we wish to buy in the shops.

```
shop_list = ['bread', 'milk', 'coffee', 'sugar']
```

A list of student names in a class.

```
student = ['Bat Man', 'Super Man', 'Wonder Woman', 'Green Hulk']
```

A list of student names with their grades in three subjects (Maths, Science and History).

```
grades = ['Bat Man', 'Maths', 60, 'Science', 70, 'History', 55,
'Super Man', 'Maths', 90, 'Science', 950, 'History', 80]
```

A list of employee names with their rate of pay per hour and the number of hours they worked in a week.

```
employee = ['Harry Potter', 12, 40, 'Wonder Woman', 15, 35, 'Hulk',
10, 38]
```

Accessing the elements of a list:

```
shop_list[0], shop_list[3]          # 1st and 4th elements

student[1], student[n]              # 2nd and (n+1)th elements

employee[22], employee[i]           # 23rd and (i+1)th elements
```

The first element in a list is always at index **0**, just as for strings.

The last element in a `list` with **`n`** items is always at **`list[n-1]`**

index	*shop_list*
0	bread
1	milk
2	coffee
3	sugar

```
shop_list = ['bread', 'milk', 'coffee', 'sugar']
```

index	*employee*
0	Harry Potter
1	12
2	40
3	Wonder Woman
4	15
5	35
6	Hulk
7	10
8	38

```
employee = ['Harry Potter', 12, 40, 'Wonder Woman', 15, 35, 'Hulk', 10, 38]
```

We can use a loop to process all of the items in a list as follows:

```
shop_list = ['bread', 'milk', 'coffee', 'sugar']

print (f'Weekly Shopping List')

i = 0
while i < 4:
    print( shop_list[i] )
    i = i+ 1
```

Output

```
Weekly Shopping List
bread
milk
coffee
sugar
```

The following list stores the name of a student and their marks in Maths, Science and History. We use it to display the student's grades on separate lines.:

```
grades = ['Joe Carthy', 'Maths', 60, 'Science', 70, 'History', 55]

print(f'Grades for: {grades[0]} are')
i = 1
while i < 6:
    print(f'{grades[i]} {grades[i+1]}' )
    i = i+ 2
```

Output

```
Grades for: Joe Carthy are
Maths 60
Science 70
History 55
```

Why do we increment `i` by 2 in the above loop?

Empty List []

An empty list (list with no items in it) is denoted by [] e.g.

```
List = []
```

You can add an entry to any list by using the `append` method e.g.

```
List.append('hello')
```

adds the string `'hello'` to `List` which now is `['hello']`

```
shop_list = ['bread', 'milk', 'coffee', 'sugar']
```

```
shop_list.append('jam')
```

adds `'jam'` to `shop_list` which now becomes:

```
['bread', 'milk', 'coffee', 'sugar', 'jam']
```

```
# L6.7 grade2.py: Read names and marks from user
# to compute the average class mark. The program then displays the list of students, their mark and
# the deviation (difference)between their mark and the class average.


grades = []                 # empty list to start
sum = 0.0
n = 0                       # number of students

name = input('\nEnter name: [quit]: ')

while ( name != 'quit' ):
    grades.append(name)                                 # Add name to list

    mark = float(input(f'Enter mark for {name}: '))
    grades.append( mark )                               # Add mark to list

    sum = sum + mark
    n = n + 1
    name = input('\nEnter name: [quit]: ')

average = sum / n   # there are n marks in the list

print (f'\n\nClass average  {average:.2f}\n')
print(f'Name      Mark    Deviation from Class average\n')

nm = len (grades)           # number of elements in grades
j = 0

# process list in pairs (0,1), (2,3), (4,5) and so on
while j < nm :
    diff = grades[j+1] - average
    print(f'{grades[j]}      {grades[j+1]}      {diff:.2f}' )
    j = j + 2                       # 2 elements per student

print(f'\nFinished \n')
```

Output

```
Enter name: [quit]: Joe
Enter mark for Joe: 55

Enter name: [quit]: Tom
Enter mark for Tom: 62

Enter name: [quit]: Jane
Enter mark for Jane: 75

Enter name: [quit]: quit


Class average  64.00

Name      Mark    Deviation from Class average
Joe      55.0      -9.00
Tom      62.0      -2.00
Jane      75.0      11.00
Finished
```

`for` Loop

The **`for` loop** is used when we know the number of times we wish to repeat the loop body. We frequently know how often we wish to repeat a statement(s). For example, we often use the **`for`** loop to process a list of items.

The `for` loop is often used in combination with the **`range()`** function.

There are 3 forms of `range` :

`range (stop)` — generate list from **0** to **`stop`**, ***not*** *including* `stop`

`range (5) gives 0, 1, 2, 3, 4`

`range(start, stop)` — generate list from **`start`** to **`stop`**, **not** including `stop`

`range (4,8) gives 4, 5, 6, 7`

`range(start, stop, step)` — generate list from **`start`** to **`stop`**, **not** including `stop`, by increments of size **`step`**

`range (0, 12, 2) gives 0, 2, 4, 6, 8, 10`

`for` Loop

In a previous example we used a while loop to sum the integers 1 to 99. It can be written using a for loop as follows:

```
# sum3.py: Sum 1 + 2 + 3 + ... +99

sum = 0          # contains the sum we wish to compute
for i in range(1, 100):                # 1, 2, 3, 4, …, 99
   sum = sum + i

print("\nSummation is:", sum, "\n")
```

The variable `i` takes on the **next value in the sequence** each time you go around the loop.

In this case, variable `i` starts with value `1` which is added to `sum`.

Then `i` becomes `2` which is added to `sum` and so on until `i` becomes `99`.

Remember that `range (1, 100)` generates the list from `1` to `99` – the stop value of `100` is NOT included in the list.

Write a program to read 5 integers, sum them and calculate the average. The program should display the sum and the average.

```
# sum3.py: Sum 5 numbers entered by user and display sum and average

sum = 0                              # contains the sum we wish to compute
for i in range(1, 6):                # read 5 numbers and sum them
     n = input(f'Enter number {i}: ')
     sum = sum + n

average = sum / 5

print(f'\n\nSum is: {sum} Average is: {average}')
```

Output:

```
Enter number 1: 1
Enter number 2: 2
Enter number 3: 3
Enter number 4: 4
Enter number 5: 5


Sum is: 15.0 Average is: 3.0
```

The `for` loop is usually used when we wish to process all the elements in a string or a a list.

Example L6.9
Write a program to output the characters in a string on separate lines.

```
# str.py: Output each character on a newline

string = 'abc'
l = len (string)                # 3 in this case

for i in range( l ):
  print( string[i] )            # displays elements 0, 1 and 2
```

Output

```
a
b
c
```

`for` Loop

We usually insert the `len` function into the `range` function when writing programs to process strings or lists:

Example L6.9
Write a program to output the characters in a string on separate lines.

```
# str.py: Output each character on a newline

string = 'abc'

for i in range( len(string) ):
  print( string[i] )            # displays elements 0, 1 and 2
```

Output

```
a
b
c
```

`for` Loop

We can also use the `for` loop to process strings or lists without using an index:

```
for x in string
    process element x of string

for x in list
    process element x of list
```

In these cases, the variable `x` takes on the value of **each element** of the string or list, starting with element 0, then element 1 and so on.

String example:

```
s = 'abc'
for x in s
    print(f'{x}')
```

Output:

```
a
b
c
```

`for` Loop

List example

```
shop_list = ['bread', 'milk', 'coffee', 'sugar']

print (f'Weekly Shopping List')

for j in shop_list:
    print(f'{j}' )
```

Output:

```
Weekly Shopping List
bread
milk
coffee
sugar
```

Time to practice !

- Copy all the examples from the slides above and get them to run in your Python environment.
- Then complete the exercises from the Handbook and get them to run.
- Finally carry out the assignments from the Handbook and get them to run.