Introduction to Python Language

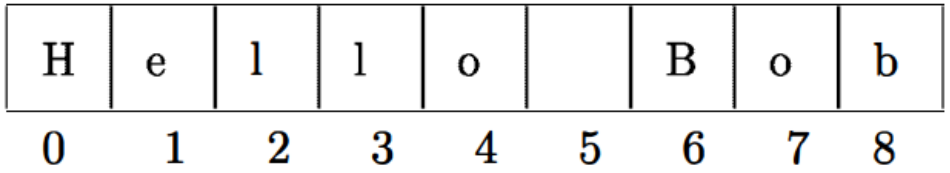
INPUT- OUTPUT (STRING) MANIPULATION

OBJECTIVES

- To become familiar with various operations that can be performed on strings through built-in functions and string methods.
- To understand the basic idea of sequences and indexing as they apply to Python strings and lists.
- To be able to apply string formatting to produce attractive, informative program output.
- To understand basic file-processing concepts and techniques for reading and writing text files in Python.
- To understand basic concepts of cryptography.
- To understand and write programs that process textual information.

Basic String Operations

- We might want to access the individual characters that make up the string. In Python, this can be done through the operation of *indexing*. We can think of the positions in a string as being numbered, starting from the left with 0.
- The general form for indexing is `<string> [<expr>]` .



- Here are some interactive indexing examples:

```
>>> greet = "Hello Bob"
>>> greet[0]
'H'
>>> print(greet[0], greet[2], greet[4])
H l o
>>> x = 8
>>> print(greet[x-2])
B
```

- By the way, Python also allows indexing from the right end of a string using negative indexes.

```
>>> greet[-1]
'b'

>>> greet[-3]
'B'
```

- Indexing returns a string containing a single character from a larger string. It is also possible to access a sequence of characters or substring from a string.
- In Python, this is accomplished through an operation called *slicing*.
- Slicing takes the form `<string> [<start> : <end>]`.

```
>>> greet[0:3]
'Hel'
>>> greet[5:9]
' Bob'
>>> greet[:5]
'Hello'
>>> greet[5:]
' Bob'
>>> greet[:]
'Hello Bob'
```

A slice produces the substring starting at the position given by **start** and running up to, but *not including*, **position end**.

- Since strings are sequences of characters, you can iterate through the characters using a Python for loop.

```
>>> name="gantep"
>>> for i in name:
...     print (i,end=' ')
...
g a n t e p
```

- Basic string operations are summarized in table below.

Table 5.1 Basic string operations

operator	meaning
+	concatenation
*	repetition
<string>[ ]	indexing
<string>[:]	slicing
len(<string>)	length
for <var> in <string>	iteration through characters

Simple String Processing

- We want to write a program that reads a person's name and computes the corresponding username. (Use first initial and seven characters of last name as a username) Using this method, the username for John Smith would just be "Jsmith."

```
print ( "This program generates computer usernames . \n" )

# get user's first and last names
first = input ( "Please enter your first name (all lowercase) : " )
last = input ( "Please enter your last name (all lowercase) : " )

# concatenate first initial with 7 chars of the last name .
uname = first [0] + last [: 7]

# output the username
print ( "Your username is: " , uname)
```

- Here's an example run:

```
This program generates computer usernames .

Please enter your first name (all lowercase) : john
Please enter your last name (all lowercase) : smith
Your username is:  jsmith
```

- Here is another problem that we can solve with string operations. Suppose we want to print the abbreviation of the month that corresponds to a given month number. The input to the program is an int that represents a month number (1-12), and the output is the abbreviation for the corresponding month. For example, if the input is 3, then the output should be Mar, for March.
- We recognize that this is a decision problem. On the other hand, we can accomplish it by simply slicing operation.

- If the input is 3, then the output should be Mar, for March.
- The basic idea is to store all the month names in a big string:

```
months = "JanFebMarAprMayJunJulAugSepOctNovDec"
```

- Since each month is represented by three letters, if we knew where a given month started in the string, we could easily extract the abbreviation:

```
monthAbbrev = months[pos:pos+3]
```

This would get us the substring of length 3 that starts in the position indicated by `pos`.

- Let's try a few examples and see what we find. Remember that string indexing starts at 0.

month	number	position
Jan	1	0
Feb	2	3
Mar	3	6
Apr	4	9

To get the correct multiple, we just subtract 1 from the month number and then multiply by 3.

- Now, we're ready to write the program.

```
# month.py
#  A program to print the abbreviation of a month, given its number

# months is used as a lookup table
months = "JanFebMarAprMayJunJulAugSepOctNovDec"

n = int(input("Enter a month number (1-12): "))

# compute starting position of month n in months
pos = (n-1) * 3

# Grab the appropriate slice from months
monthAbbrev = months[pos:pos+3]

# print the result
print("The month abbreviation is", monthAbbrev + ".")
```

Here is a sample of program output:

```
Enter a month number (1-12): 4
The month abbreviation is Apr.
```

Python Lists as Sequences

- As a matter of fact, the operations given in Table 5.1 are not only string operations. They are operations that apply to sequences.
- Python **lists** are also a kind of *sequence*. That means we can also index, slice, and concatenate lists, as the following session illustrates:

```
>>> [1,2] + [3,4]
[1, 2, 3, 4]
>>> [1,2]*3
[1, 2, 1, 2, 1, 2]
>>> grades = ['A','B','C','D','F']
>>> grades[0]
'A'
>>> grades[2:4]
['C', 'D']
>>> len(grades)
5
```

One of the nice things about lists is that they are more general than strings.
Strings are always sequences of characters, whereas lists can be sequences of arbitrary objects. We can mix the types up.

```
myList = [1, "Spam", 4, "U"]
```

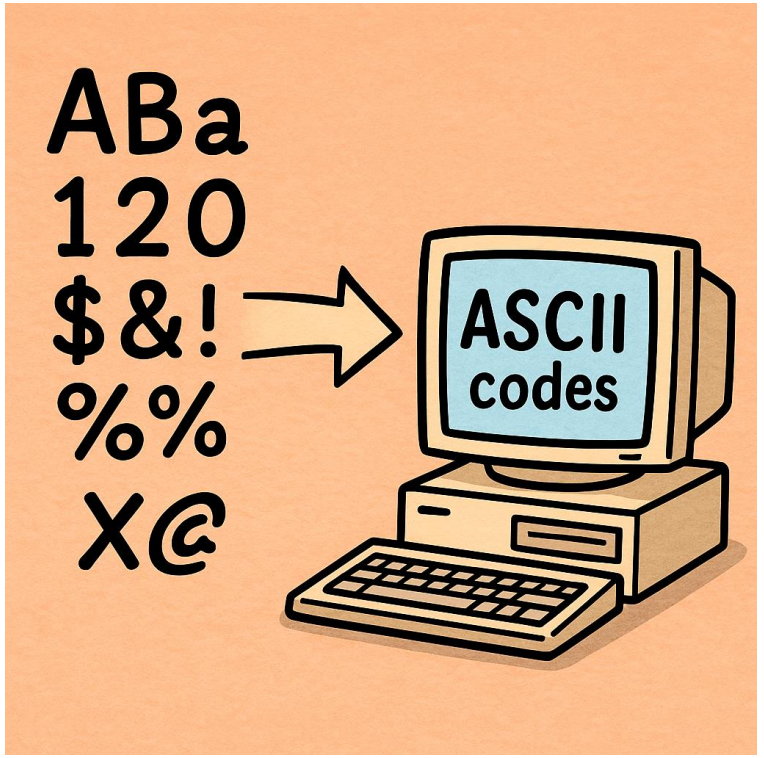
- Using a list of strings, we can rewrite our month abbreviation program from the previous section and make it even simpler:

```
# month2.py
#  A program to print the month abbreviation, given its number.
# months is a list used as a lookup table
months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

n = int(input("Enter a month number (1-12): "))

print("The month abbreviation is", months[n-1] + ".")
```

String Representation and Message Encoding

- A computer represents strings in this way: Each character is translated into a number, and the entire string is stored as a sequence of (binary) numbers in computer memory.
- It doesn't really matter what number is used to represent any given character as long as the computer is consistent about the encoding/decoding process.
- Today, computer systems use industry standard encodings.

- One important standard is called ASCII (American Standard Code for Information Interchange).
- ASCII uses the numbers 0 through 127 to represent the characters typically found on an (American) computer keyboard, as well as certain special values known as control codes that are used to coordinate the sending and receiving of information.



- Most modern systems are moving to **Unicode**, a much larger standard that aims to include the characters of nearly all written languages.

The `ord( )` and `chr( )` Functions

- Python provides a couple of built-in functions that allow us to switch back and forth between characters and the numeric values used to represent them in strings.
- The `ord` function returns the numeric ("ordinal") code of a single character string, while `chr` goes the other direction. Here are some interactive examples:

```
>>> ord("a")
97
>>> ord("A")
65
>>> chr(97)
'a'
>>> chr(90)
'Z'
```

Programming an Encoder

- Using the Python ord and chr functions, we can write some simple programs that automate the process of turning messages into sequences of numbers and back again.
- The algorithm for encoding the message is simple:

```
get the message to encode
for each character in the message:
    print the letter number of the character
```

- Here the Python codes:

```
print("This program converts a textual message into a sequence")
print("of numbers representing the Unicode encoding of the message.\n")

# Get the message to encode
message = input("Please enter the message to encode: ")

print("\nHere are the Unicode codes:")

# Loop through the message and print out the Unicode values
for ch in message:
    print(ord(ch), end=" ")

print() # blank line before prompt
```

- Here an example run for the code:

```
This program converts a textual message into a sequence
 of numbers representing the Unicode encoding of the message .

Please enter the message to encode : This is a code

Here are the Unicode codes :
84 104 105 115 32 105 115 32 97 32 99 111 100 101
```

One thing to notice about this result is that even the space character has a corresponding Unicode number. It is represented by the value 32.

Programming a Decoder

- Now that we have a program to turn a message into a sequence of numbers, it would be nice if our friend on the other end had a similar program to turn the numbers back into a readable message.
- Our decoder program will prompt the user for a sequence of Unicode numbers and then print out the text message with the corresponding characters.
- Here is the decoding algorithm:

```
get the sequence of numbers to decode
message = ""
for each number in the input:
    convert the number to the corresponding Unicode character
    add the character to the end of message
print message
```

Before the loop, the accumulator variable **`message`** is initialized to be an empty string; that is, a string that contains no characters ‘ ’. Each time through the loop, a number from the input is converted into an appropriate character and appended to the end of the `message` constructed so far.

- How exactly do we get the sequence of numbers to decode? We don't even know how many numbers there will be.
- First, we will read the entire sequence of numbers as a single string using input. Then we will split the big string into a sequence of smaller strings, each of which represents one of the numbers.
- Here is the complete algorithm:

```
get the sequence of numbers as a string, inString
split inString into a sequence of smaller strings
message = ""
for each of the smaller strings:
    change the string of digits into the number it represents
    append the Unicode character for that number to message
print message
```

The `.split()` Function

- Strings have some built-in methods in addition to the generic sequence operations that we have used so far.
- The `split` method splits a string into a list of substrings. By default, it will split the string wherever a space occurs.
- Here an example:

```
>>> myString = "Hello, string methods ! "
>>> myString . split ()
['Hello, ', 'string' , 'methods ! ']
```

- Split can be used to split a string at places other than spaces by supplying the character to split on as a parameter. Here an example for that:

```
>>> myString . split(",")
['Hello', ' string methods ! `]

>>> "32, 24, 25, 57" . split ( ", " )
[' 32' ' '24' ' '25' ' '57']
```

- Here another example, we could get the x and y values of a point in a single input string, turn it into a list using the split method, and then index into the resulting list to get the individual component strings as illustrated in the following interaction:

```
>>> coords = input ( "Enter the point coordinates (x, y) : " ) . split(",")
Enter the point coordinates (x, y) : 3.4, 6.25
>>> coords
['3.4','6.25']
>>> coords [0]
'3.4'
>> coords [1]
'6.25'
```

- Of course, we still need to convert those strings into the corresponding numbers.

```
coords = input ( "Enter the point coordinates (x, y) : " ).split(",")
x, y = float(coords[0]), float (coords[1])
```

- Returning to our decoder, we can use a similar technique. Since, the sequence Unicode numbers produced by Encoder program has space between each numbers, the default split command will work.

```
>>> "87 104 97 1 16 32 97 32 83 1 1 1 1 17 1 14 1 12 1 17 1 15 1 15 33".split()
['87','104','97','116','32','97','32','83','111','117','114', '112', '117',
'115', '115', '33']
```

- Of course, we still need to convert those strings into the corresponding numbers.

```
print ( "This program converts a sequence of Unicode numbers into " )
print ( "the string of text that it represents . \n" )

# Get the message to encode
inString = input ( "Please enter the Unicode-encoded message : " )

# Loop through each substring and build Unicode message
message = " "
for numStr in inString . split () :
        codeNum = int (numStr) # convert digits to a number
        message = message + chr (codeNum) # concatentate character to message
print ( " \nThe decoded message is : ", message)
```

- Output of the program:

```
This program converts a sequence of Unicode numbers into
the string of text that it represents .

Please enter the Unicode-encoded message : 84 104 105 115 32 105 115 32 97 32 99
 111 100 101

The decoded message is :   This is a code
```

More String Methods

- Python is a very good language for writing programs that manipulate textual data. Table 5.2 lists some other useful string methods.

function	meaning
`s.capitalize()`	Copy of `s` with only the first character capitalized.
`s.center(width)`	Copy of `s` centered in a field of given `width`.
`s.count(sub)`	Count the number of occurrences of `sub` in `s`.
`s.find(sub)`	Find the first position where `sub` occurs in `s`.
`s.join(list)`	Concatenate `list` into a string, using `s` as separator.
`s.ljust(width)`	Like `center`, but `s` is left-justified.
`s.lower()`	Copy of `s` in all lowercase characters.
`s.lstrip()`	Copy of `s` with leading white space removed.
`s.replace(oldsub,newsub)`	Replace all occurrences of `oldsub` in `s` with `newsub`.
`s.rfind(sub)`	Like `find`, but returns the rightmost position.
`s.rjust(width)`	Like `center`, but `s` is right-justified.
`s.rstrip()`	Copy of `s` with trailing white space removed.
`s.split()`	Split `s` into a list of substrings (see text).
`s.title()`	Copy of `s` with first character of each word capitalized.
`s.upper()`	Copy of `s` with all characters converted to uppercase.

Table 5.2: Some string methods

- Let's use these methods.

```
>>> s="hello, I came here for an argument"
>>> s.capitalize()
'Hello, i came here for an argument'
>>> s.title ()
'Hello, I Came Here For An Argument'
>>> s.lower()
'hello, i came here for an argument'
>>> s.upper ()
'HELLO, I CAME HERE FOR AN ARGUMENT'
>>> s.replace ( "I" , "you")
'hello, you came here for an argument'
>>> s.center (30)
'hello, I came here for an argument'
>>> s.center (50)
'     hello , I came here for an argument'
```

```
>>> s.count('e')
5
>>> s.find( ',')
5
>>>" ".join(["Number" , "one ," , "the" ,
"Larch"] )
'Number one , the Larch'
>>> "spam" . join(["Number" , "one ","the "
, "Larch"] )
'Numberspamone , spamthespamLarch'
```

- Like strings, lists are also objects and come with their own set of "extra" operations.
- The `append` method can be used to add an item at the end of a list. This is often used to build a list one item at a time.
- Here's a fragment of code that creates a list of the squares of the first 100 natural numbers:

```
squares = []
for x in range ( 1 , 101) :
squares . append(x*x)
```

- With the append method in hand, we can go back and look at an alternative approach to our little decoder program.

```
def main() :
        print ( "This program converts a sequence of Unicode numbers into " )
        print ( "the string of text that it represents . \n" )

# Get the message to encode
        inString = input ( "Please enter the Unicode-encoded message : " )

# Loop through each substring and build Unicode message
        chars = []
        for numStr in inString.split () :
                codeNum = int (numStr)
                chars.append(chr(codeNum) )

message = "".join(chars)
# convert digits to a number
# accumulate new character
        print ( " \nThe decoded message is : " , message)

main ()
```

Modular Programming: By including def main ( ), the program codes will be run when the program is called. Otherwise, the codes will always run when it is imported to another sub program.

The final message is obtained by joining these characters together using an empty string as the separator.
So, the original characters are concatenated together without any extra spaces between.

Input/Output as String Manipulation

- For example, consider a program that does financial analysis. Some of the information (e.g., dates) must be entered as strings.
- The results of the analysis will typically be a nicely formatted report including textual information that is used to label and explain numbers, charts, tables, and figures.

Example Application: Date Conversion

- As a concrete example, let's extend our month abbreviation program to do date conversions. The user will input a date such as "24/02/2025," and the program will display the date as "February 24, 2025."

```
Input the date in dd/mm/yyyy format (dateStr)
Split dateStr into day, month and year strings
Convert the month string into a month number
Use the month number to look up the month name
Create a new date string in form Month Day , Year
Output the new date string
```

- Here is the program:

```
dateStr = input ( "Enter a date (dd/mm/yyyy) : " )
dayStr,monthStr,yearStr = dateStr.split ("/")

months=["January","February","March","April","May","June",
    "July","August","September","October","November","December"]
monthStr = months [int (monthStr)-1]
print("The converted date is:" , monthStr, dayStr + ",",yearStr)
```

- When run, the output looks like this:

```
Enter a date (dd/mm/yyyy) : 24/02/1988
The converted date is: February 24, 1988
```

- We now have a complete set of operations for converting values among various Python data types. Table 5.3 summarizes these four Python type conversion functions:

function	meaning
`float(<expr>)`	Convert `expr` to a floating-point value.
`int(<expr>)`	Convert `expr` to an integer value.
`str(<expr>)`	Return a string representation of `expr`.
`eval(<string>)`	Evaluate `string` as an expression.

Table 5.3: Type conversion functions

String Formatting (f-string method)

- The reason to converting numbers into string is to output the results nicely. Python provides a powerful string formatting operation that makes the job much easier.
- Let's start with a simple example.

```
>>>name = "Ali"
>>>print(f"Hello, {name}!")
Hello, Ali!
```

- f shows the string given includes string formating. Additionaly, variables can be used in the curly braces {}.

```
>>>name = "Ali"
>>>age= 25
>>>print(f"{name} is {age} years old.")
Ali is 25 years old.
```

```
>>> pi = 3.14159
>>>print(f"Pi number is {pi:.2f}")
Pi number is 3.14
```

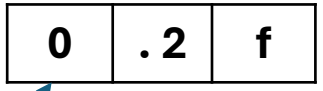
String Formatting (.format( ) method)

- This method uses curly braces ({})as placeholders (slots).

```
{<index>:<format-specifier>}.format(<values>)
```

```
>>> total = 12.3456
>>>print("The total value of your change is ${0:0.2f}".}".format(total))
The total value of your change is $12.35
```

```
<width>.<precision><type>
```



The precision is 2, which tells Python to round the value to two decimal places.

The type character f says the value should be displayed as a fixed-point number.

The width specifies how many "spaces" the value should take up. Putting a 0 here essentially says "use as much space as you need."

```
>>> "Hello {0} {1}, you may have won ${2}".format("Mr.", "Smith", 10000)
'Hello Mr. Smith, you may have won $10000'
>>>item = "notebook"
>>>quantity = 3
>>>price = 12.5

>>>print("You bought {0} {1}(s) for ${2:.2f} each.".format(quantity, item, price))
You bought 3 notebook(s) for $12.50 each.
```

- Often, you'll want to control the width and/or precision of a numeric value.

```
>>> "This int, {0:5}, was placed in a field of width 5".format(7)
'This int,    7, was placed in a field of width 5'

>>> "This int, {0:10}, was placed in a field of width 10".format(7)
'This int,          7, was placed in a field of width 10'
```

```
>>> "This float, {0:10.5}, has width 10 and precision 5".format(3.1415926)
'This float, 3.1416, has width 10 and precision 5'

>>> "This float, {0:10.5f}, is fixed at 5 decimal places".format(3.1415926)
'This float, 3.14159, is fixed at 5 decimal places'

>>> "This float, {0:0.5}, has width 0 and precision 5".format(3.1415926)
'This float, 3.1416, has width 0 and precision 5'

>>> "Compare {0} and {0:0.20}".format(3.14)
'Compare 3.14 and 3.1400000000000001243'
```

Notice that for normal (not fixed-point) floating-point numbers, the precision specifies the number of significant digits to print.

For fixed-point (indicated by the f at the end of the specifier) the precision gives the number of decimal places.

- You may notice that, by default, numeric values are right -justified. This is helpful for lining up numbers in columns.
- Strings, on the other hand, are left -justified in their fields.
- You can change the default behaviors by including an explicit justification character at the beginning of the format specifier.

```
>>>"{:<10}".format("hi")      # right aligned
>>>"{:^10}".format("hi")      # centered
>>>"{:>10}".format("hi")      # left aligned
```

```
>>> "{:<10}".format("Hi")
'Hi        '
>>> "{:>10}".format("Hi")
'        Hi'
>>> "{:^10}".format("Hi")
'    Hi    '
```

```
>>>text = "Python"

>>>print(f"{text:<10}")    # left aligned
>>>print(f"{text:^10}")    # centered
>>>print(f"{text:>10}")    # right aligned
```

```
>>> text="Hi"
>>> print(f"{text:<10}")
Hi
>>> print(f"{text:>10}")
        Hi
>>> print(f"{text:^10}")
    Hi
```

Next Lecture

File Processing