**Module 5. Strings**

**Learning Objectives**

* Understand the concept of sequences and their significance in real-world and programming contexts.
* Identify and manipulate individual characters within a string using indexing and slicing.
* Demonstrate string concatenation, joining, splitting, and the use of escape sequences.
* Utilize string methods to modify and process strings effectively.
* Apply regular expressions for pattern matching in strings.
* Implement string formatting techniques for clear and readable output in Python programs.

# **String: A sequence of Characters**

Sequences are a big part of our everyday life. For instance, you might remember that your car is the fourth one in a line (a sequence) in a parking lot. Or you might have a to-do list (a sequence) of tasks for the afternoon. Teaching a child the letters in a word is also dealing with a sequence.

In simple terms, a sequence is just an ordered list of items. Here are some real-world examples of sequences:

* Stock prices listed one after another on a stock ticker.
* Exam scores listed from the first test to the final exam.
* Scientific measurements, like electrical current readings taken over time.
* A long list of names.
* A series of text messages from a friend stored on your phone.

No matter what data each item in the sequence contains, you can process the sequence in the same way: by referring to individual items or subsequences, and by looping through it. Programs often store sequences of data because this data typically appears together in real life, and it needs to be processed together to make sense of it.

Some reasons sequences are useful in software include:

* The sequence data often describes the same thing, event, or process.
* Software solves real-world problems by processing real-world information.
* Real-world information often comes in sequences, so we need to capture and process it in a program to get new, useful information.

In Python, a string is simply a sequence of characters. This means that every character in a string is also a string.

For example, if you type the following in the Python interactive shell:

|  |
| --- |
| type("help"[0]) |

This command does two things:

1. Selects the first character ('h') in the string "help".
2. Passes it into the built-in ***type*** function to print its data type.

The result <class 'str'> shows that each character in a string is also considered a string in Python.

This means processing a string is like processing any other sequence. The same techniques for accessing and modifying elements apply to strings, but the way you format the contents is unique to strings.

# **Strings in Programs**

When working with strings in your software program, you usually:

1. **Work with the string as a whole**: For example, printing someone's name.
2. **Pick out specific characters within the string**: For example, remembering the third character on your to-do list.

In Python, you can process a string using operators with the sequence literal or variable.

**Example of Accessing Characters**

For example, ***myname[0]*** accesses the first letter in a string stored in the variable ***myname*** using the variable name and some symbols.

|  |
| --- |
| myname = "John"  print(myname[0]) # Output: 'J' |

An operator is a symbol that performs a specific operation on one or more operands. For instance, the statement ***employment\_income + retirement\_income*** uses the addition (+) operator to add the values of the two variables.

|  |
| --- |
| employment\_income = 50000  retirement\_income = 150000  print(employment\_income + retirement\_income) # Output: 200000 |

**Example of Calling Methods**

You can also process a string by calling methods on the object holding the sequence. For example, ***myname.upper()*** is called on the string object in ***myname*** to produce the uppercase version of the string.

|  |
| --- |
| myname = "John"  print(myname.upper()) # Output: 'JOHN' |

This works because the Python string type (***str***) has the upper method defined in its class. You can see more about str by typing help(***str***) in the Python interactive mode.

**Working with String Literals**

String literals are the actual string content visible in the source code. This is useful when the string data is assigned initially or used directly.

For example:

|  |
| --- |
| print("Take everything " + "literally") |

This code concatenates two string literals using the + operator before printing the result. Python interprets the + operator between two strings as a concatenation operator.

**Joining Strings**

If you want to include more content than just the existing strings, you can use the join method of the str class. This method allows you to concatenate strings with a fixed string in between each of them.

|  |
| --- |
| >>>".".join(["192", "168", "1", "1"])  '192.168.1.1' |

This joins the strings in the list ***["192", "168", "1", "1"]*** with a ***.*** in between, resulting in ***'192.168.1.1***'.

**Splitting Strings**

You can also split a string into separate strings based on a fixed substring.

For example:

|  |
| --- |
| >>>'192.168.1.1'.split(".")  ['192', '168', '1', '1'] |

This splits the string at each ***.*** and results in the list ***['192', '168', '1', '1']***.

**Escape Sequences**

Escape sequences are symbols in a string that represent special characters, such as a new line (***\n***) or a tab (***\t***). These sequences are used to include special content that isn't visible in the string but needs to be printed.

|  |
| --- |
| print("I need many lines to express my\n\n\nbrilliance")  print("Tabs ...\taren’t enough to do it") |

The first statement prints multiple new lines, and the second statement prints a tab space.

A black screen with white text

Description automatically generated

**Object-Oriented Programming with Strings**

In object-oriented programming, an object is like a small machine with valuable data inside. We call methods on the object to make it do useful work with its data. In Python, when you use a string, you're working with a str object, and all the capabilities of the str data type are available for you to use.

This code example demonstrates storing a string in a variable, indexing it to access its parts, and calling a function from its object.

Listing 6-1. strings.py

# Store two strings in variables

myname = "Monty"

mylastname = "Python"

# Include the strings in expressions to use their values

print("Yeah, I’m " + myname + " " + mylastname + ". Who’s asking?")

# Index the strings to access particular parts of them

print("Shorten my name to " + myname[0] + mylastname[0] + " if you want, but that’s rude")

print("My username? " + myname[0] + mylastname + ". Why are you asking for this??")

# join method joins the two strings 'around' the contents of the string object

#it is called from

print("I was told to add strange characters to my password, so I made it " + "✩!@".join([myname, mylastname]) + ". Arggh.. I just said that out loud, didn’t I...")

# Call a function from the variable to process it, then print it

print(myname.lower())

Here’s the output of running the program:

|  |
| --- |
| Yeah, I’m Monty Python. Who’s asking?  Shorten my name to MP if you want, but that’s rude  My username? MPython. Why are you asking for this??  I was told to add strange characters to my password, so I made it Monty✩!@Python. Arggh.. I just said that out loud, didn’t I...  monty |

# **Strings are Immutable**

In Python, strings are **immutable**, meaning that once a string is created, it cannot be changed. Let's break this down with some simple examples.

**What Does Immutable Mean?**

When we say that strings are immutable, it means:

* You cannot change the characters in a string directly.
* If you want to change a string, you have to create a new string.

**Example 1: Trying to Change a Character**

Let's say we have a string:

|  |
| --- |
| greeting = "Hello"  greeting[0] = 'Y' |

If we try to change the first character 'H' to 'Y', it will give an error because strings are immutable:

|  |
| --- |
| >>> TypeError: 'str' object does not support item assignment |

**Example 2: Creating a New String**

Instead of changing a string directly, we create a new string. For example, if we want to change "Hello" to "Yello", we can do this:

|  |
| --- |
| greeting = "Hello"  new\_greeting = "Y" + greeting[1:]  print(new\_greeting) |

This code does the following:

1. Takes the original string "Hello".
2. Creates a new string "Yello" by combining "Y" with the substring "ello" (everything except the first character).
3. Prints "Yello".

**Example 3: Reassigning a Variable**

If we reassign a string variable, we are not changing the original string; we are just making the variable reference a new string:

|  |
| --- |
| message = "Good morning"  message = "Good evening"  print(message) #Output: "Good evening" |

Here, we change message from "Good morning" to "Good evening". The original string "Good morning" remains unchanged in memory.

**Example 4: Using String Methods**

String methods like ***upper()***, ***lower()***, ***replace()***, etc., do not change the original string but return a new string:

|  |
| --- |
| text = "hello world"  new\_text = text.upper()  print(new\_text) #Output: "HELLO WORLD"  print(text) #Output: "hello world" |

This code:

1. Converts "hello world" to "HELLO WORLD" using the ***upper()*** method.
2. Stores the result in ***new\_text***.

However, the original text variable is still "hello world".

In Python:

* Strings cannot be changed after they are created.
* To modify a string, you need to create a new string.
* String methods return new strings and do not alter the original.

# **Indexing the String**

**Picking Out Characters in a String**

Imagine you are on a game show with three luxury cars lined up. If you want to choose the second car, you'd say "I want the second car." Similarly, in programming, you use numbers to pick specific elements in a string or sequence.

In Python, you can select a particular element from a sequence by typing an integer inside square brackets ***[ ]***. This is called **indexing**. To select a range of elements, you use **slicing**, which involves specifying a starting and ending index separated by a colon ***:***.

**Indexing and Slicing Examples**

* **Indexing**: Picking a Single Character

You can select a single character from a string by its position. Positions start at 0, not 1.

|  |
| --- |
| mystring = 'Monty'  print(mystring[0]) # Output: 'M'  print(mystring[1]) # Output: 'o'  print(mystring[2]) # Output: 'n' |

* **Combining Characters**: Using Multiple Indexes

You can combine characters from different positions to create new strings.

|  |
| --- |
| print(mystring[0] + mystring[4]) # Output: 'My'  print(mystring[0] + 'onda' + mystring[4]) # Output: 'Monday' |

* **Slicing**: Picking a Substring

To pick a range of characters, specify a starting index and an ending index. The ending index is not included in the result.

|  |
| --- |
| print(mystring[0:3]) # Output: 'Mon'  print(mystring[3:]) # Output: 'ty'  print(mystring[3:5]) # Output: 'ty' |

To summarize:

* **Indexing**: Use square brackets ***[ ]*** with a number to pick a specific character from a string. For example, ***mystring[0]*** gives the first character.
* **Slicing**: Use a colon ***:*** between two numbers inside square brackets to pick a range of characters. For example, ***mystring[1:3]*** gives the characters from index 1 to index 2.
* **Combining Characters**: You can combine different characters to form new strings.
* **Splitting**: You can split a string into a list of parts based on a specific character or substring.

# **Matching**

Regular expressions are a powerful way to specify patterns for matching strings. In Python, the built-in re module provides methods for working with regular expressions. Let's break this down with an example and some explanations.

**Example in Python Interactive Mode**

Here's an example of how to use the re.match method to find matches in strings:

|  |
| --- |
| >>>import re  # Get help information about re.match  >>>help(re.match)  # Use re.match to find a match for the pattern ".\*bob.\*" in the string "bob and alice"  >>>match = re.match(".\*bob.\*", "bob and alice")  >>>print(match) |

Output will be:

<re.Match object; span=(0, 13), match='bob and alice'>

(Explanation)

* **Importing the Module**: First, we import the ***re*** module, which contains all the functions needed for regular expressions.
* **Getting Help**: Using ***help(re.match)***, we can see detailed information about the ***re.match*** method.
* **Matching**: The ***re.match*** method checks if the pattern in the first parameter matches the string in the second parameter. If it does, it returns a ***Match*** object.

In our example, the pattern "***.\*bob.\****" matches the string "***bob and alice***". This is because the pattern specifies that the string can contain "***bob***" with any characters before and after it.

**Special Characters in Regular Expressions**

Regular expressions use special characters to define patterns efficiently:

* ***.\****: This means "0 or more instances of any single character."
* ***.***: This means "any single character."

These special characters allow us to create complex patterns without writing lengthy logic.

**Practical Use**

Regular expressions are widely used in programming for tasks like:

* Searching for specific patterns in text.
* Validating input data (like email addresses or phone numbers).
* Replacing parts of strings.

# **The *split* Method**

The ***split*** method in Python is used to divide a string into a list of substrings based on a specified delimiter. It is a powerful tool for breaking down strings into manageable pieces, which is particularly useful in data processing and text manipulation tasks. Here’s a detailed explanation of how to use the ***split*** method with various delimiters.

**Basic Usage of split**

The split method takes a string and splits it into parts based on a specified delimiter. The syntax is as follows:

*string.split(delimiter, maxsplit)*

* **delimiter**: The character or substring on which to split the string. If not specified, the default delimiter is any whitespace.
* **maxsplit**: An optional argument that specifies the maximum number of splits. If not specified, the string is split at every occurrence of the delimiter.

**Examples**

* **Splitting by Whitespace (Default Behavior)**

If no delimiter is provided, the split method splits the string at any whitespace (spaces, tabs, or newlines).

text = "This is a sample string"

words = text.split()

print(words)  # Output: ['This', 'is', 'a', 'sample', 'string']

* **Splitting by a Specific Character**

You can split a string using a specific character as the delimiter. For example, splitting by a comma:

csv\_line = "apple,banana,cherry"

fruits = csv\_line.split(',')

print(fruits)  # Output: ['apple', 'banana', 'cherry']

* **Splitting by a Substring**

The delimiter can also be a substring.

text = "Hello---world---Python"

parts = text.split('---')

print(parts)  # Output: ['Hello', 'world', 'Python']

* **Using *maxsplit* Argument**

The ***maxsplit*** argument specifies the maximum number of splits to be done. If provided, the string is split at the first ***maxsplit*** occurrences of the delimiter.

text = "one, two, three, four, five"

parts = text.split(',', 2)

print(parts)  # Output: ['one', ' two', ' three, four, five']

* **Splitting by Multiple Characters (Regular Expressions)**

To split by multiple different characters, you can use the re module's split function. Here’s how you can split by both comma and space:

import re

text = "apple, banana, cherry; orange"

parts = re.split('[,; ]+', text)

print(parts)  # Output: ['apple', 'banana', 'cherry', 'orange']

* **Splitting by Newlines**

To split a string by newlines, use the ***split*** method with ***\n*** as the delimiter.

text = "Line 1\nLine 2\nLine 3"

lines = text.split('\n')

print(lines)  # Output: ['Line 1', 'Line 2', 'Line 3']

# **Formatting the String**

When working with strings in Python, you often need to format them to make the data more readable and presentable. Formatting helps in arranging the data in a way that looks good and is easy to understand.

**Basic Formatting Using f*ormat()***

The ***format()*** method in Python allows you to insert variables into a string in specific places, which makes it easy to create well-formatted strings.

**Example of Basic Formatting**

Here's an example:

|  |
| --- |
| username = "Monty"  password = "123456"  print('You entered "{}" "{}"'.format(username, password)) |

Output will be:

You entered "Monty" "123456"

In this example:

* The ***format()*** method is called on the string ***'You entered "{}" "{}"'***.
* The curly braces ***{}*** act as placeholders for the variables ***username*** and ***password***.
* The ***format()*** method fills these placeholders with the values of ***username*** and ***password***.

**Formatting Details**

* **Field Width and Alignment**: You can specify how wide the space for each variable should be and how the content should be aligned.

|  |
| --- |
| print('{:10} {:10}'.format(username, password)) |

This will print ***username*** and ***password*** each in a field of 10 characters wide.

Monty 123456

* **Right-Justification**: You can also align the text to the right within the specified width.

|  |
| --- |
| print('{:>10} {:>10}'.format(username, password)) |

This will right-align the ***username*** and ***password***.

Monty 123456

**Example 1: Change Counter**

The following program reads the number of each type of coin from the user and calculates the total amount. It uses string formatting to present the total value in a user-friendly way.

# Function defined to do the work

def main():

    print("Change Counter\n")

    print("Please enter the count of each coin type.")

    quarters = eval(input("Quarters: "))

    dimes = eval(input("Dimes: "))

    nickels = eval(input("Nickels: "))

    pennies = eval(input("Pennies: "))

    total = quarters \* 25 + dimes \* 10 + nickels \* 5 + pennies

    # Each {} contains an index number and symbols control appearance.

    # Total is divided by 100 with integer division for the first parameter (dollars)

    # and modulus with 100 for the second parameter (cents)

    # See https://docs.python.org/3.6/library/stdtypes.html#str.format

    print("The total value of your change is ${0}.{1:02}".format(total // 100, total % 100))

main()

The Output of Running the Program

*Change Counter*

*Please enter the count of each coin type.*

*Quarters: 1*

*Dimes: 1*

*Nickels: 1*

*Pennies: 1*

*The total value of your change is $0.41*

**Example 2: Encoding**

This example demonstrates encoding, where each character in a string is converted to its numeric representation using Unicode. The script then prints both the numeric representation and the original character.

# Sequences: encoding

# A string message to encode.

mymessage = 'Monty'

# Try out indexing and concatenation

print(mymessage[0] + mymessage[1] + mymessage[2])

# Encode: in this case we can encode and decode a character from a string.

for char in mymessage:

    print('Numeric version:', ord(char))

    print('Recover the original:', chr(ord(char)))

(Explanation)

* **String Message**: ***mymessage*** is the string to be encoded.
* **Indexing and Concatenation**: Demonstrates how to access and combine characters from the string.
* **Encoding**: Uses ***ord()*** to convert each character to its numeric (Unicode) representation.
* **Decoding**: Uses ***chr()*** to convert the numeric representation back to the original character.

**Summary**

1. A sequence is an ordered list of items, such as stock prices or exam scores.
2. Python strings are sequences of characters, where each character is also a string.
3. You can access individual characters in a string using indexing, e.g., ***"help"[0]*** returns ***'h'***.
4. Strings can be processed as whole entities or by accessing specific characters.
5. Operators like ***+*** can concatenate strings, e.g., ***"Hello" + " World"*** gives ***"Hello World"***.
6. Methods like ***.upper()*** can be called on string objects to transform them, e.g., ***"hello".upper()*** returns ***"HELLO"***.
7. String literals are explicit string values in the source code, useful for initial assignments.
8. The join method can concatenate strings with a separator, e.g., ***".".join(["192", "168"])*** returns "192.168".
9. The split method divides a string into a list based on a delimiter, e.g., ***"192.168.1.1".split(".")*** gives ***['192', '168', '1', '1']***.
10. Escape sequences like ***\n*** for new lines and ***\t*** for tabs add special characters to strings.
11. In Python, strings are objects, and their methods can be used for various operations.
12. Strings are immutable, meaning their content cannot be changed once created.
13. To modify a string, you create a new string, not alter the original.
14. Indexing retrieves single characters from a string, e.g., ***"Monty"[0]*** gives ***'M'***.
15. Slicing retrieves substrings, e.g., ***"Monty"[0:3]*** gives ***'Mon'***.
16. Regular expressions specify patterns for matching strings using special characters.
17. The ***re.match*** method checks if a pattern matches a string, returning a Match object.
18. Formatting strings using the ***format()*** method inserts variables into strings at specified positions.
19. String methods like ***.upper()*** return new strings without modifying the original.
20. Encoding converts characters to their numeric Unicode representations, and decoding reverses this.

**Programming Exercises**

1. **Reverse a String**

Write a program that asks the user to enter a string and then displays the string in reverse order. For example, if the user enters "hello", the program should display "olleh".

1. **Palindrome Checker**

Write a program that asks the user to enter a string and checks whether the string is a palindrome. A palindrome is a string that reads the same forwards and backwards, ignoring spaces, punctuation, and capitalization. For example, "A man, a plan, a canal, Panama" is a palindrome.

1. **Word Counter**

Write a program that asks the user to enter a sentence and then counts the number of words in the sentence. For example, if the user enters "The quick brown fox jumps over the lazy dog", the program should return 9.

1. **Character Frequency Counter**

Write a program that asks the user to enter a string and then counts the frequency of each character in the string. The program should display each character along with its frequency.

1. **Anagram Checker**

Write a program that asks the user to enter two strings and checks whether they are anagrams. Anagrams are words or phrases that contain the same characters but in a different order. For example, "listen" and "silent" are anagrams.

1. **Title Case Converter**

Write a program that asks the user to enter a sentence and then converts the first character of each word to uppercase. For example, if the user enters "the quick brown fox", the program should display "The Quick Brown Fox".

1. **Phone Number Formatter**

Write a program that asks the user to enter a 10-digit phone number as a single string of digits and formats it as (XXX) XXX-XXXX. For example, if the user enters "1234567890", the program should display "(123) 456-7890".

1. **Remove Vowels**

Write a program that asks the user to enter a string and then removes all vowels (a, e, i, o, u) from the string. For example, if the user enters "beautiful", the program should display "btfl".

1. **Caesar Cipher**

Write a program that asks the user to enter a string and an integer shift value, and then encodes the string using a Caesar cipher. A Caesar cipher shifts each character in the string by the given number of positions in the alphabet. For example, with a shift of 3, "a" becomes "d", "b" becomes "e", and so on.

1. **Initials Formatter**

Write a program that asks the user to enter a string containing multiple names separated by spaces, and then displays the initials of each name. For example, if the user enters "John William Smith Alice Mary Brown", the program should display "J. W. S. A. M. B.".

1. **Word Lengths Calculator**

Write a program that asks the user to enter a sentence. The program should split the sentence into words and then display the length of each word. For example, if the user enters "Hello world from Python", the program should display:

Word lengths:

Hello: 5

world: 5

from: 4

Python: 6

1. **Domain Extractor**

Write a program that asks the user to enter a series of email addresses separated by spaces. The program should split the input string into individual email addresses and then extract and display the domain of each email address. For example, if the user enters "user1@example.com user2@test.org user3@domain.net", the program should display:

Domains:

example.com

test.org

domain.net