Logo, company name

Description automatically generated

Python User Guide

Table of Contents:

[Installing Python and Visual studio Code](#Section_1_0) ………………………………… Section 1.0

[Python basic printing](#Section_2_0)…………………………………………………………………… Section 2.0

[Creating your own variables](#Section_3_0)……………………………………………………… Section 3.0

[Printing in Python with variables](#Section_3_1)………………………………………………… Section 3.1

[Printing in Python formatted f string](#Section_3_2) ……………………………………… Section 3.2

[Printing in Python continued (yes again) format method](#Section_3_3) ………… Section 3.3

[Input function and convert data type](#Section_3_4) …………………………………………… Section 3.4

[Manipulating Strings in Python](#Section_4_0) ………………………………………………. Section 4.0

[Examples of using Python String Methods](#Section_4_1) …………………. Section 4.1

Count within a string

Convert to String to lowercase

[Validate String with Methods](#Section_4_2) ………………………………………….. Section 4.2

Length Validation

Empty or Non-Empty Validation

Character Validation

Isalpha

Isnumeric

Isalnum

Isdigit

Islower

Isupper

Prefix and Suffix Validation

Startswith

Endswith

Pattern Matching (Regular Expressions)

Membership Validation

In Keyword

WhiteSpace Validation

Isspace

Custom Validation

[Splitting Strings](#Section_4_3)  …………………………………………………………… Section 4.3

[Using the Join Method](#Section_4_4) ……………………………………………………. Section 4.4

[Using the Find Method](#Section_4_5) ………………………………………………. Section 4.5

[Repeating string characters](#Section_5_0)  ………………………………………………………. Section 5.0

[Justify text](#Section_5_1) ……………………………………………………………………………………. Section 5.1

[Strip white space](#Section_5_2) ………………………………………………………………………… Section 5.2

[Replace Method](#Section_5_3) …………………………………………………………………………… Section 5.3

[Slice String and indexes](#Section_6_0) ……………………………………………………………….. Section 6.0

[Finding indexes based on Character](#Section_6_1) ………………………………………………. Section 6.1

[Numbers and math](#Section_7_0) …………………………………………………………………… Section 7.0

Order of Operations

Concatenate strings vs numbers

Casting Numbers

Division by Zero

Modulo Operator

Floor Division

Ceiling Division

Python Math Library

Common shape formulas

Square

Triangle

Circle

Cylinder

Speed of a falling object

[Comments](#Section_8_0) …………………………………………………………………………………… Section 8.0

Single Line Comment

Multi-Line Comment

[Python add ons (Time delay)](#Section_8_1) ………………………………………………………… Section 8.1

[Printing with colors](#Section_9_0)  ……………………………………………………………… Section 9.0

[Comparison operators](#Section_10_0) ………………………………………………………… Section 10.0

Equal

Not Equal

Greater Than

Less Than

Greater than of equal to

Less than or equal to

[Logical operators](#Section_11_0) ………………………………………………………………… Section 11.0

And

Or

Not

[Conditional statements](#Section_12_0) ………………………………………………………… Section 12.0

If statements

If-else statement

If elif else statements

Nested if statements

Ternary Operator

[Special or Escape Characters](#Section_13_0) ………………………………………………… Section 13.0

New Line

Tab

Backslash

Single Quote

Double Quote

Backspace

Carriage Return

Form Feed

[Try/Except/Finally](#Section_14_0) ……………………..……………………………………… Section 14.0

Try Except

Catching Multiple Exceptions

Finally Block

[While Loops](#Section_15_0) ……………………..……………………………………… Section 15.0

Simple While Loop

Using Input to Terminate the Loop

[Data validation with While Loops](#Section_15_1) ……………………..…………………… Section 15.1

[Break and Continue](#Section_15_2) …………………………………………………………….. Section 15.2

Break

Continue

[Be careful about Infinite loops](#Section_15_3) ……………………..………………… Section 15.3

[For Loops](#Section_15_4)  …………………………..……………………..………………… Section 15.4

Iterating over a List

Iterating over a String

Using the range function

Nesting For Loops

[Creating and working with Python List](#Section_16_0)  ………………………………… Section 16.0

Creating List

Accessing List Elements

Length of a List

Adding Elements to a List

Removing Elements from a List

Slicing List

Looping through List

List Comprehensions

[Creating and working with Python List](#Section_17_0) …………………………………….. Section 17.0

Append

Clear

Copy

Count

Extend

Index

Insert

Pop

Remove

Reverse

Sort

[Slice list](#Section_18_0)  …………………………………………………………………………………… Section 18.0

List name

Start Index

End Index

Step

Omitting Start and End Indices

Negative Indices

Specifying a Step Size

Negative Step Size

Modifying Lists with Slicing

[Delete Statement](#Section_19_0)  …………………………………………………………………… Section 19.0

[length of List](#Section_20_0)  …………………………………………………………………… Section 20.0

[Create a copy of List](#Section_20_1)  ………………………………………………………………… Section 20.1

Slicing

Using List Function

Using Copy Method

[List Examples](#Section_20_2)  ……………………………………………………………………… Section 20.2

[Packing and Unpacking List](#Section_21_0)  ………………………………………………… Section 21.0

Packing List

Creating Lists

Using List Function

Unpacking List

Assigning to Variables

Using \* Operator

Extra Material

Python coding examples ……………………………………………………………… Section 10

[Final project example](#Section10_1) ……………………………………………………… Section 10.1 Example 1

[Tic-Tac\_Toe example](#Section10_1example2) …………………………………………………………… Section 10.1 Example 2

[Tic-Tac-Toe (BYU-I) example](#Section10_1example3)  ……………………………………………………… Section 10.1 Example

Useful Reference:

Python Official Documentation  
<https://docs.python.org/3/>

A series of beginning videos for Python developed by Microsoft:

<https://www.youtube.com/watch?v=jFCNu1-Xdsw&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=1>

W3Schools for Python:

<https://www.w3schools.com/python/default.asp>

Python Programming reference sheet:

<https://pythonprinciples.com/reference/>

Python reference sheet:

<http://rgruet.free.fr/PQR25/PQR2.5.html>

Python lessons with examples:

<https://holypython.com/>

Section 1.0

Installing Python and VS:

<https://code.visualstudio.com/download>

<https://www.python.org/downloads/>

<https://marketplace.visualstudio.com/items?itemName=ms-python.python>

Section 2.0

Printing in Python:

The print function is one of the first commands to learn in Python. It’s a function that is built into the language itself so you are able to access it.

A few rules to remember:

print is lower case- Python is a cap sensitive language.

After the word print you must have ()

After the () you must enclose your statement inside of “”.

You may use single quotes (‘) double quotes (“) or triple (‘’’) however most often you will see double.

Think of the print function as “display to screen.” It does not print to your printer.

Example:

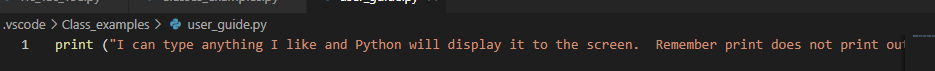


Output:

Text

Description automatically generated

You can type anything you like inside of the quotes:

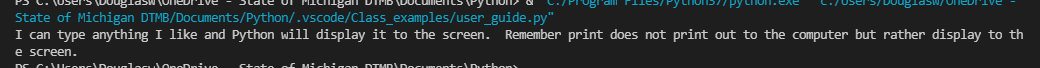


Hit the play button:

Text

Description automatically generated

Output:



Here are some things to consider about single, double, and triple quotes.

print ("Boy it's hot")

output



With single quotes we encounter a problem:

Text

Description automatically generated

Python sees the first ‘ but ending quote is in the word it’s. That caused confusion. Whereas when we did double quotes, the opening “ went to the ending “ so it could print ‘ without issue.

Triple quotes are rare but would be needed if you quote a person.

Text

Description automatically generated

Because we start with triple quotes it is looking to end on triple quotes. We have double quotes as part of the message so triple was needed.

Convention is double quotes, unless triple is needed.

Section 3.0

Creating your own variables:

**Video Resource:**

<https://www.google.com/search?q=python+create+variable+video&rlz=1C1GCEA_enUS953US953&oq=python+create+variable+video&aqs=chrome..69i57j33i160l2j33i22i29i30l7.439255045j0j15&sourceid=chrome&ie=UTF-8#kpvalbx=_amLGZKCuNbe20PEP2duk8Ac_28>

A variable is an empty box (or memory stored within your computer). You can place anything inside of it that you like. Variables seem simple at first, but it’s essential to understand them well because you will use them a lot.

Variable by convention should be lower case. Python allows upper case, but I discourage use of upper case, as programmers agree to use lower case to avoid confusion.

Example: Note the name is text (string) so it’s enclosed in quotes. Research data types as there is a lot of learn about different data types.

Graphical user interface, text, application

Description automatically generated

Create your variable name and assign the data with an equal sign.

Now anytime we use the variable name the computer knows it equal Weylin.

We can print our variable.

Text

Description automatically generated

Line 1 of code said create a variable called name and store the value “Weylin”.

Line 2 said print out whatever is stored in the variable.

We can change the value of name if we like. Just replace the value:

Text

Description automatically generated

Computer now says I see Amanda as the text in the variable name.

We can create more than 1 variable if we like.

Text

Description automatically generated

Notice the variable do not contain spaces use underscore instead. Using\_under\_score is called snake case.

Master the above before moving on. Having a solid foundation of this is necessary to understand the rest of Python programming.

Resources:

[String Concepts](https://www.youtube.com/watch?v=tSebLz1hNpA&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=9) (6 mins)

[Demo: Strings](https://www.youtube.com/watch?v=zv3cVJHCqXA&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=10) (4 mins)

Section 3.1

Printing in Python with variables:

**Video Resource:**

<https://www.youtube.com/watch?v=i83VkP0LHPI>

Learn about datatypes before attempting this. Also understand variables before moving to printing continued sections.

Resource for datatypes: https://www.w3schools.com/python/python\_datatypes.asp

Welcome to “printing continued” we were unable to scare you off so we will try again.

I created many variables with different data types.

Text

Description automatically generated

I can print this is many ways. From above you can write a print statement for each but not we will learn a few more dynamics ways.

**First the F string.**

Syntax print(f”your text here”). The f string is a way to concatenate values together. This is the easiest way, but we will show you several because you we see several different ways will you do a Google search.

Graphical user interface, text

Description automatically generated

Notice I can add a variable in my print statement. The variable is housed inside of {} brackets.

Output:  


Python said, you want to print, you are doing a f string so I am prepared to do many things here.

I see you want me to print information stored in first\_name then add the text afterword’s.

Example 2:

A screenshot of a computer

Description automatically generated with medium confidence

You can print FN/LN then text.



Note I didn’t put a space between first and last name, so it did not print a space.

Example 3:



Now we have the space.

Output:



Example 4:

You can do LN, FN than text:



Output:



Let’s get more complex. More variables:



Output:



Change value to money:



Output:



You can see we can do any number of variables and text as we like.

Resource for variables: <https://www.w3schools.com/python/python_variables.asp>

<https://www.w3schools.com/python/python_variables_names.asp>

Video on printing

[Using Print](https://www.youtube.com/watch?v=FhoASwgvZHk&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=5) (3 mins)

Know the above data well before moving on!

Section 3.2

Printing in Python formatted (f-string):

**Video Resource:**

<https://www.youtube.com/watch?v=eszrY7w83q8>

We can format the variables. We can make cocoa into Cocoa or COCOA.

We can make 2000000 into 2,000,000.00

I changed the variables to have better data to work with:

Text

Description automatically generated

Now we get.



Output



.capitalize()



Output:



Even though the data started lower case Python fixed to upper.

If data starts as upper then it’s just upper.

.capitalize() ensures the data ends up as capitalized.

Note you must end .capitalize with () or you will get an error.

Others you can use is .lower()



Prints all lower case

.upper()



Prints all upper case.



You can Google others but those are the only 3 I used on string.

Now to numbers:

Graphical user interface, text

Description automatically generated

Output:



Add two places after the decimal place. Add : after the variable then .2f.





If you commas on the thousand’s place add a , after the :

Graphical user interface, text

Description automatically generated

Output:



Resource: <https://www.w3schools.com/python/ref_string_format.asp>

[Formatting Strings](https://www.youtube.com/watch?v=bQQqxysLIGE&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=11) (4 mins)

[Demo: Formatting Strings](https://www.youtube.com/watch?v=E850-MF22P0&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=12) (4 mins)

Section 3.3

Printing in Python continued (yes again) format method:

**Video Resource:**

<https://www.youtube.com/watch?v=UwuBFtW9ni0>

Personally I tend to only use the F string. It’s likely you will do the same. However, we you Google how do I do X in Python you will see others use a different print style. F string was more recently developed and old school programmer have their methods down.

Graphical user interface

Description automatically generated

Here we create a print statement and feed values right into the print statement.



We can also use this for stored variables:

A screenshot of a computer

Description automatically generated with medium confidence

Output:



Note the first placeholder is equals the first argument. The first {} is first\_name and the second is value in this case.

We can change the order of the assignment if we like. Remember the first number in Python is 0 (actually all programming languages.)

This example makes not sense but see how we can easily change the order:

Graphical user interface, text

Description automatically generated

Output:



Believe me this statement is now false, Cocoa is better than $2,000,000.

Can we also use formatting like we did with F String? Yes.

Example:



Output:



Note: I hard coded Cocoa as the 1st (index 0) argument and value as the 2nd (index 1). This ensure false statements are no longer said against Cocoa.

I also tried to format the value and I don’t know how to do it. I am sure it’s possible but F string is easier so it’s really not worth me investigating.

For this section just make sure you understand how to read it so when you see other peoples’ code you understand it.

Section 3.4:

Input function and convert data type:  
Video Resources:

<https://www.youtube.com/watch?v=5yhn0MFLcu8&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=13>

<https://www.youtube.com/watch?v=T1j2tfZK7OI&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=14>

<https://www.w3schools.com/python/python_casting.asp>

We create a variable then add the input function with string inside the quotes:

first\_name = input("What is your first name? ")

A number given in an input is always returned as a string:

Text

Description automatically generated

Note: this is a large reason to get bugs, so remember to convert string is Int or float if comparing to a number.

Convert input to string:

Text

Description automatically generated

Convert to float:

Text

Description automatically generated

Convert back to string:

Text

Description automatically generated

Section 4.0

Manipulating Strings in Python

Resources:  
<https://www.studytonight.com/post/python-string-methods-isdigit-isnumeric-and-isdecimal>

<https://initialcommit.com/blog/python-isalpha-string-method>

In Python, string methods are built-in functions that can be called on strings to perform various operations and manipulations. These methods help you work with strings efficiently and perform tasks such as searching, modifying, splitting, joining, and formatting strings. Here are some of the most commonly used string methods in Python:

**str.capitalize()**: Returns a copy of the string with the first character capitalized and the rest in lowercase.

**str.upper()**: Returns a copy of the string with all characters converted to uppercase.

**str.lower()**: Returns a copy of the string with all characters converted to lowercase.

**str.strip()**: Returns a copy of the string with leading and trailing whitespace characters removed.

**str.startswith(prefix)**: Returns **True** if the string starts with the specified **prefix**; otherwise, returns **False**.

**str.endswith(suffix)**: Returns **True** if the string ends with the specified **suffix**; otherwise, returns **False**.

**str.replace(old, new)**: Returns a copy of the string with all occurrences of **old** replaced by **new**.

**str.split(separator)**: Splits the string into a list of substrings based on the given **separator**.

**str.join(iterable)**: Concatenates the elements of the **iterable** (e.g., a list) into a single string using **str** as the separator.

**str.find(substring)**: Returns the lowest index of the first occurrence of **substring** in the string. Returns **-1** if the **substring** is not found.

**str.count(substring)**: Returns the number of non-overlapping occurrences of **substring** in the string.

**str.isdigit()**: Returns **True** if all characters in the string are digits; otherwise, returns **False**.

**str.isalpha()**: Returns **True** if all characters in the string are alphabetic; otherwise, returns **False**.

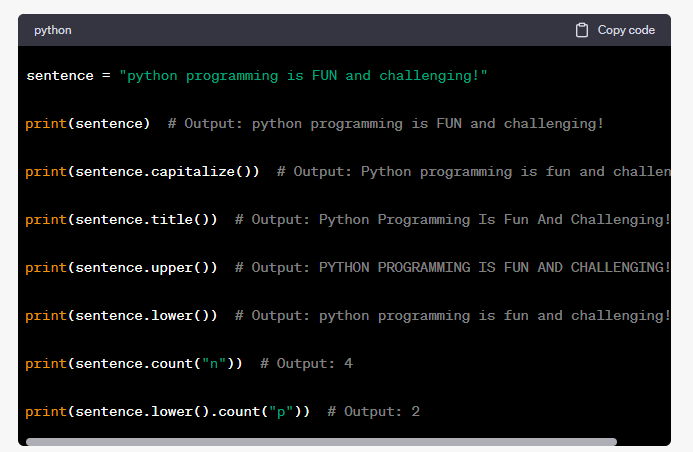
**str.isalnum()**: Returns **True** if all characters in the string are alphanumeric (letters and numbers); otherwise, returns **False**.

**str.islower()**: Returns **True** if all characters in the string are lowercase; otherwise, returns **False**.

**str.isupper()**: Returns **True** if all characters in the string are uppercase; otherwise, returns **False**.

**str.format()**: Formats the string by replacing placeholders **{}** with the specified values.

These are just a few examples of the many string methods available in Python. To explore more string methods and their usage, you can refer to the Python documentation or try them out in your code using the built-in **help()** function or online Python interpreters.

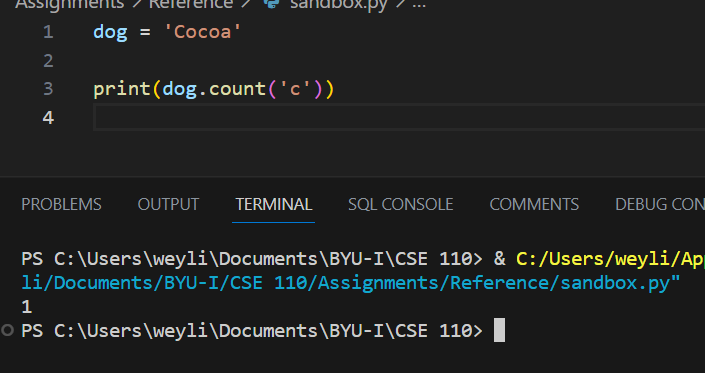


There are many formatting types. You don’t need to memorize them all. Know a few you are likely to use and know how to Google search the others when you need them.

**Section 4.1**

Examples of using Python Methods:

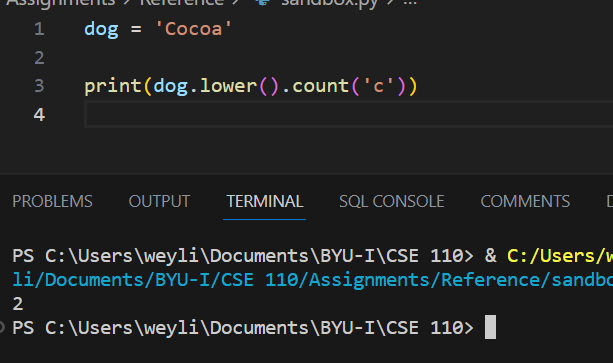
Count example:



Note there is only 1 because there is only 1 lowercase c.

**Convert to Lower and Count**

Example convert to lowercase first them counting:



Breakdown the above example



The first case prints the word exact as is: Cocoa

The second case prints the word but converting to all lowercase: cocoa

The third case count the number of lowercase “c”: 2

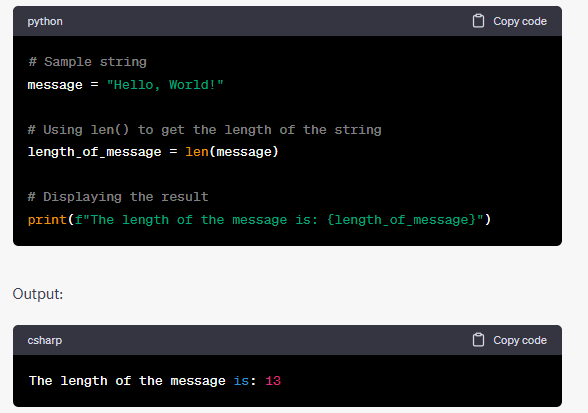
**Section 4.2**

Examples validating code with Python Methods:

Validating strings in Python typically involves performing checks to determine whether a given string meets specific criteria or conditions. Python offers several methods and functions that can be used to validate strings effectively. Here's a summary of some common string validation methods in Python:

Length Validation:

len(): To check the length of a string, use the built-in len() function. For example, to ensure a string has a specific length, compare len(your\_string) with the desired length.



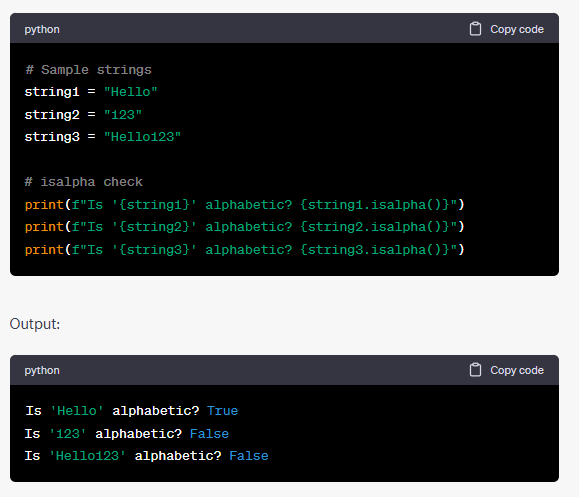
Empty or Non-empty Validation:

Checking for an empty string: Use the not keyword or len() to check if a string is empty. For example, if not your\_string or if len(your\_string) == 0.

Checking for a non-empty string: Simply use the if statement. For example, if your\_string:.

Character Validation:

isalpha(): Checks if all characters in the string are alphabetic.



isnumeric(): Checks if all characters in the string are numeric.

A screenshot of a computer

Description automatically generated with medium confidence

isalnum(): Checks if all characters in the string are alphanumeric (combination of letters and numbers).

A screenshot of a computer

Description automatically generated with medium confidence

isdigit(): Checks if all characters in the string are digits.

A screenshot of a computer

Description automatically generated with medium confidence

islower(): Checks if all characters in the string are lowercase letters.

A screenshot of a computer

Description automatically generated with medium confidence

isupper(): Checks if all characters in the string are uppercase letters.

A screenshot of a computer

Description automatically generated with medium confidence

**Prefix and Suffix Validation:**

startswith(prefix): Checks if the string starts with the specified prefix.

endswith(suffix): Checks if the string ends with the specified suffix.

**Pattern Matching (Regular Expressions):**

The re module allows more complex string validations using regular expressions.

Functions like re.match(), re.search(), and re.findall() help find patterns within strings.

**Membership Validation:**

in keyword: Checks if a substring is present in the string. For example, if 'substring' in your\_string.

**Whitespace Validation:**

isspace(): Checks if the string contains only whitespace characters.

**Custom Validation:**

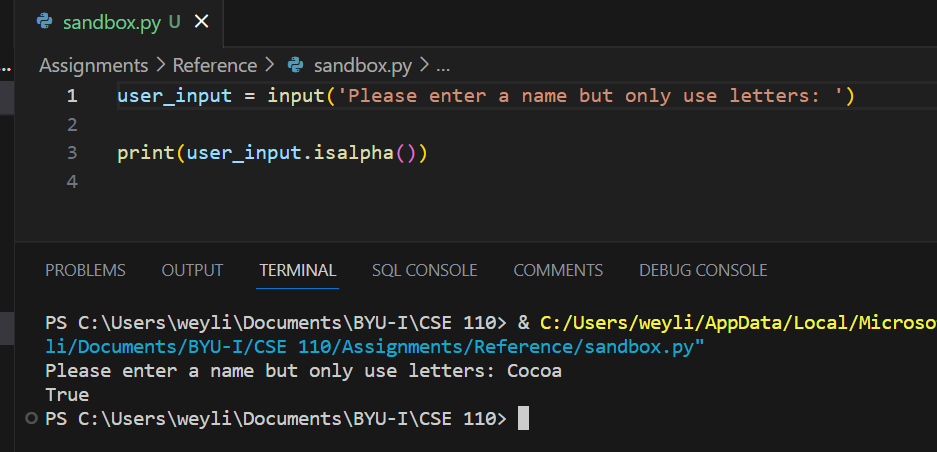
Use custom logic and loops to validate strings based on specific requirements. For example, ensuring certain characters are present or absent, or enforcing a particular pattern.

When validating strings, consider the requirements of your application and choose the appropriate method(s) to ensure the string adheres to the desired criteria. Depending on the complexity of validation, a combination of these methods may be used.

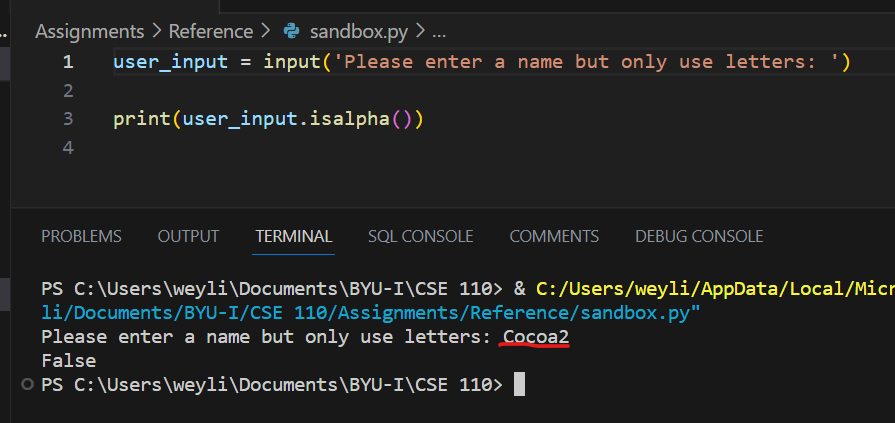
We can validate the user’s input by using Python’s build in methods.

Example:

In this example “Cocoa” contains only letters so Python returns the Boolean value of true:

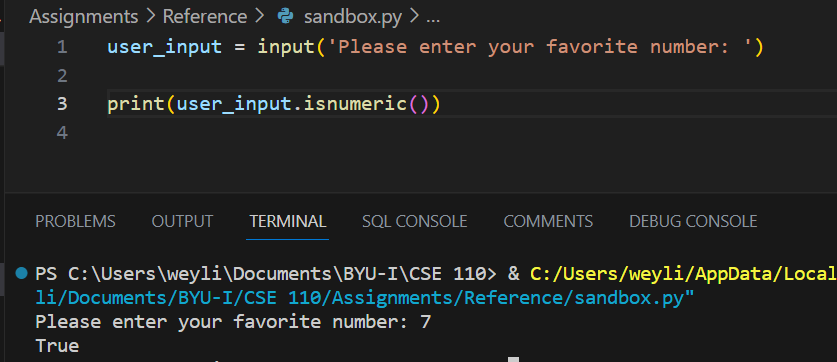


Example 2:



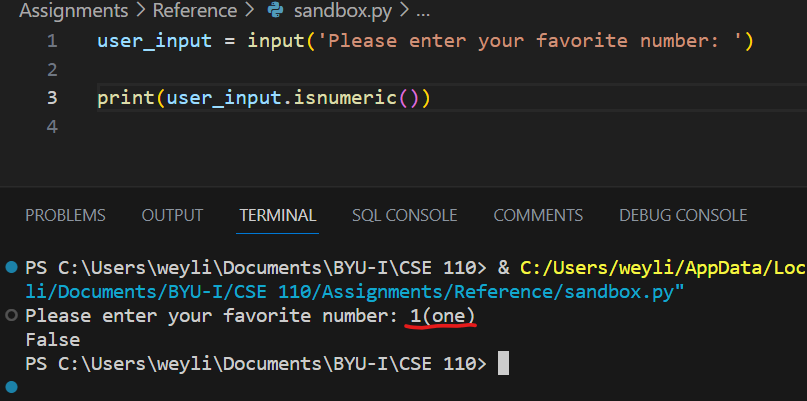
Cocoa2 contains a number, therefore the isalpha value is False

Example with numbers:



7 is a number so it returns True

Example:



I used more than just numbers so it returned false.

**Section 4.3**

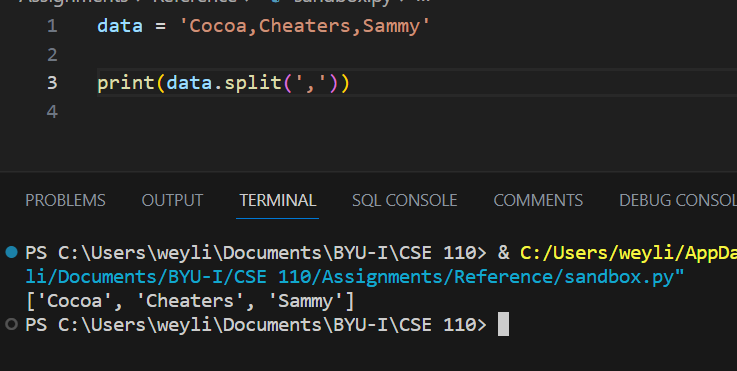
Examples splitting data using Python Method:

Resources:

<https://www.w3schools.com/python/ref_string_split.asp>

Sometimes your data is separated by a special character and you wish to spilt the data based on that character.

Example:

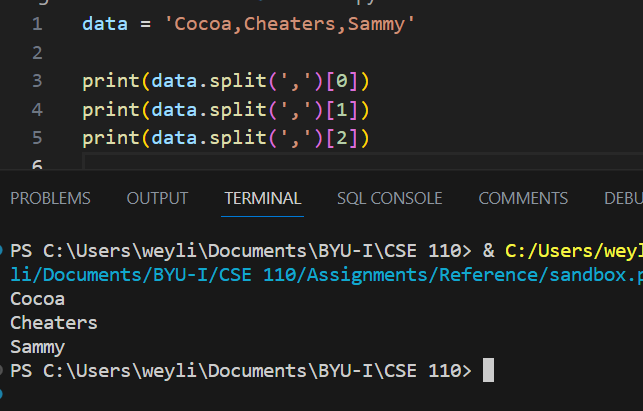


Note: the above data creates a new list (which we haven’t covered yet😊)

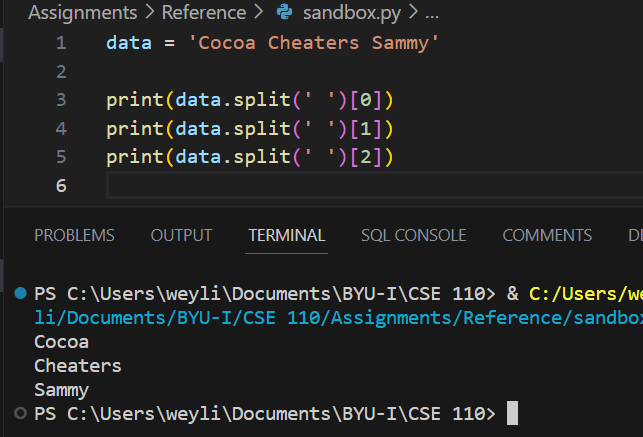
However you can the string is now 3 different animals in this case:

We can access each animal individually using indexes:

Example:



Example 2:

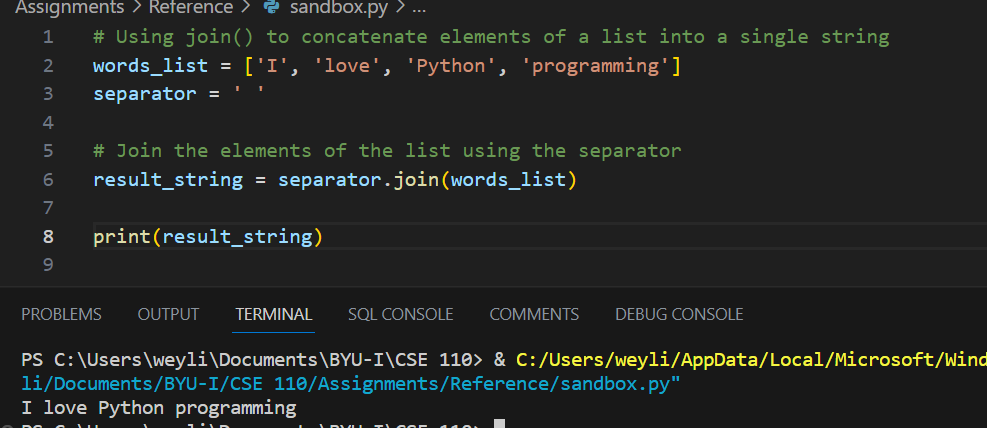


Same example but splitting on a space instead of a comma.

**Section 4.4**

Using the Join Method:

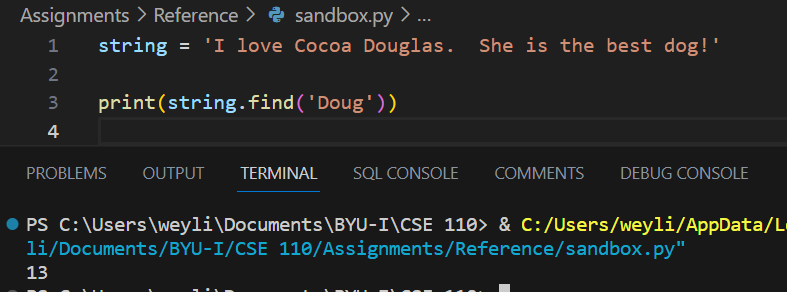
We can join a list of words into one string if we like:

Example:  


Section 4.5

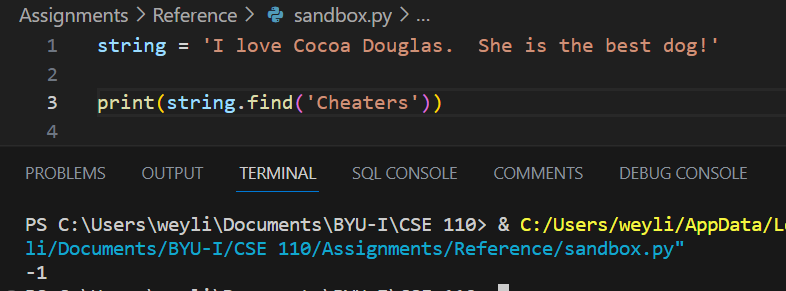
Using the Find Method:

Example:



In the variable named string, the word “Doug” starts at the index of 13.

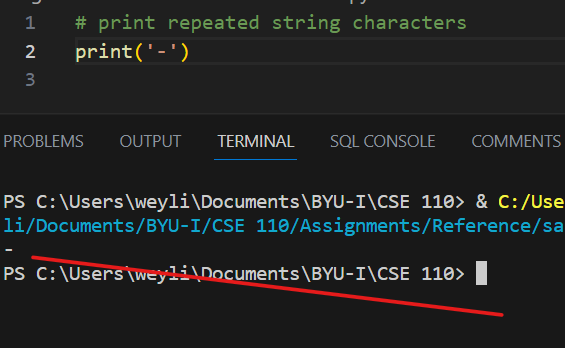
Example:

If the substring is not found in the text a -1 is returned:  


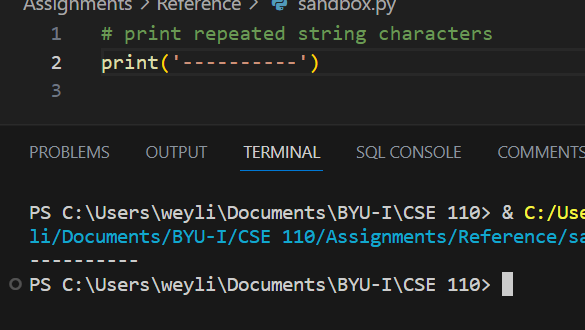
Section 5.0

Repeating string characters:

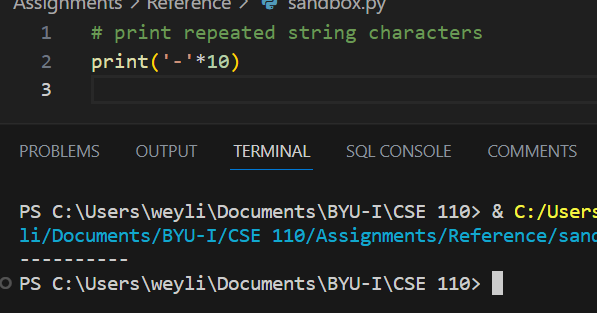
If you print one item it will show up like:



You can print many by entering as many as you like:

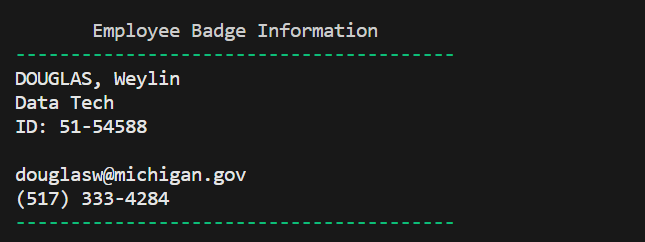


Alternatively, you can print by multiplying the string by any given number you like:



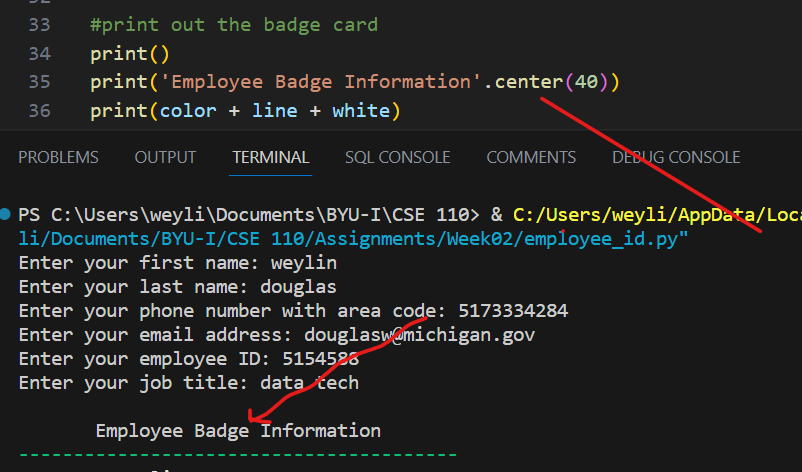
Note: in this case the \* 10 must be outside the single quotes but inside of the parenthesis.

The – and \* etc make a good border for your data. See example below:



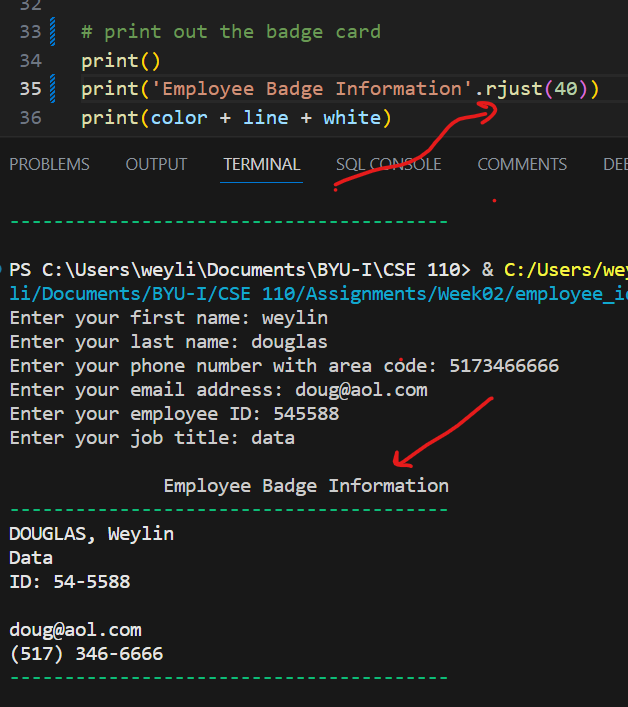
Section 5.1

Justify text:



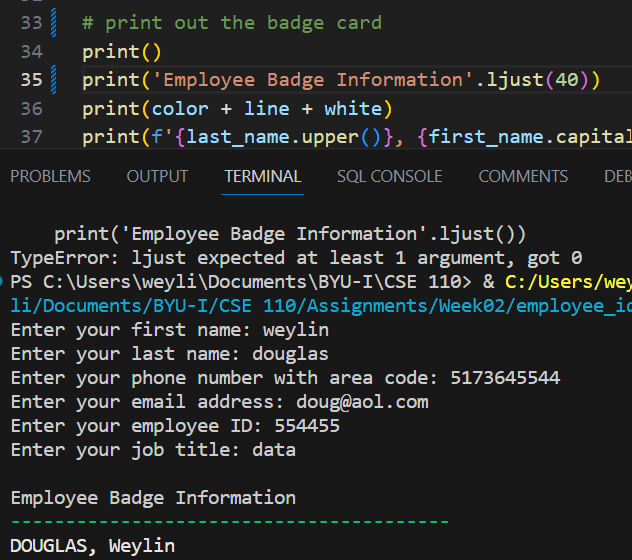
Note you must pass in an argument for the number of spaces to use for the justification.

Right justified



Note you must pass in an argument for the number of spaces to use for the justification.

Left justified (default if none is used.



You must pass in an argument. You can enter a zero or the true number you wish to use.

Section 5.2

Strip white space:

**Video Resource:**

<https://www.youtube.com/watch?v=2Pr6dc6mZdM>

Text

Description automatically generated

Notice all the white space I am entering above.

Text

Description automatically generated with medium confidence

Note my name has 6 letters and I typed 10 whites spaces:

Text

Description automatically generated

See above length is 16 because of white space.

Strip white space

Text

Description automatically generated

I added spaces after my name but it removed them all. In the strip function I said to strip “ “ (a space).

You can leave the strip blank and it assumes spaces both before and after.

Text

Description automatically generated

lstrip() will only do the left side/rstrip will only do the right side.



Note: strip will not remove the space if a string has a true space in it. Therefore, Weylin B will not become WeylinB. It does leave the string Weylin B as intended. Strip only removes spaces before or after the string as desired.

**Section 5.3**

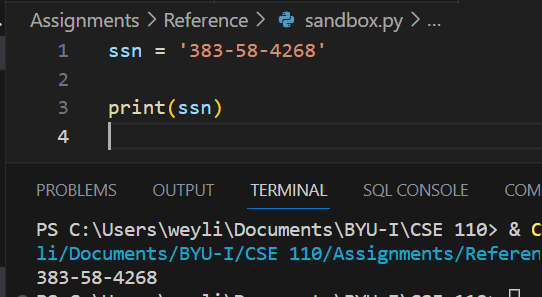
**Replace method:**

**Video Resource:**

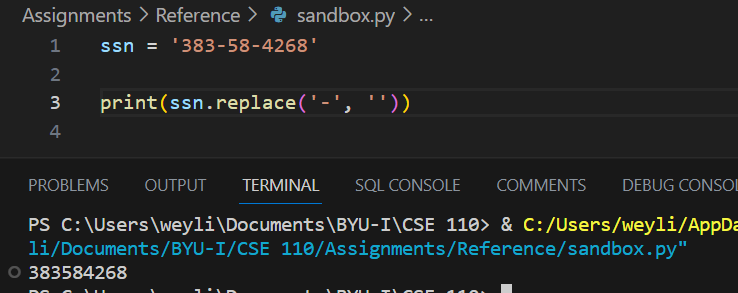
<https://www.youtube.com/watch?v=pGy8PkgPC6I>

We can remove characters that we don’t want with the replace method. We pass in the string that we want to remove:

We can see our string contains hyphens:

****

After the replace method is applied:

****

**Section 6.0:**

**Slice String and indexes:**  
Resources:

<https://www.w3schools.com/python/python_strings_slicing.asp>

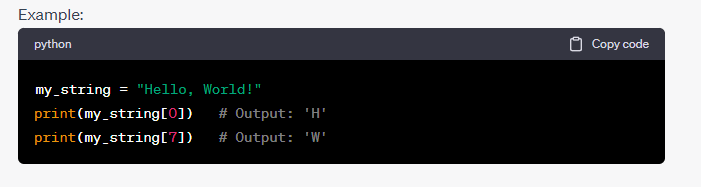
<https://www.youtube.com/watch?v=cPypu6RLijA>

In Python, indexing refers to the process of accessing individual elements within a data structure, such as strings, lists, tuples, or other sequential data types. Indexing allows you to retrieve specific elements from these data structures using their position or index.

The index is an integer value that represents the position of an element within the data structure. In Python, indexing is zero-based, which means the first element is at index 0, the second element is at index 1, and so on.

Here's a more detailed explanation of indexing for some common data structures:

1. Indexing in Strings: Strings are sequences of characters, and you can access individual characters using their index.

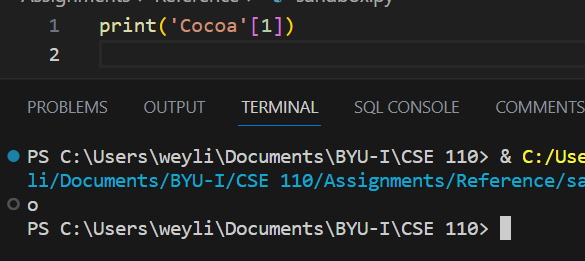
****

Indexes start at 0 and every character counts including spaces:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| H | E | L | L | O | , |  | W | O | R | L | D |

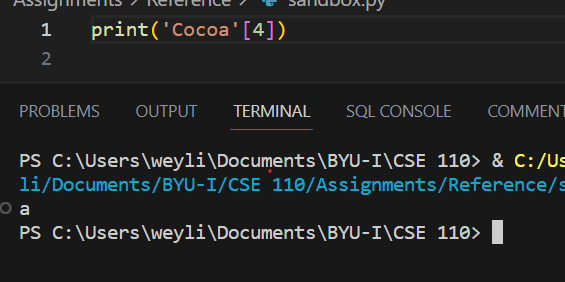
In the above example H is index of 0, the 1st L is at the index of 2, the 2nd L is at the index of 3 and space is at the index of 6.

We can access the index on a string by doing string[x] where x is the index you wish to return.

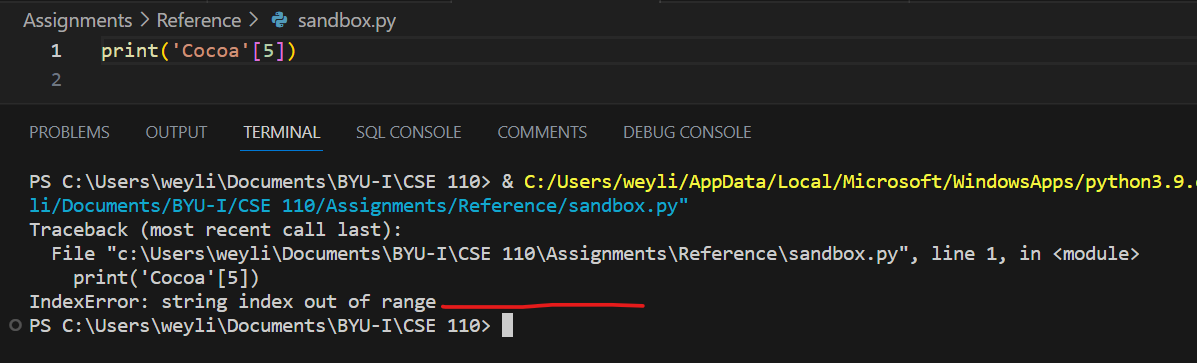


Note the 1st index of Cocoa produced o above.

Cocoa at the index of 4 is a:



Notice if we do an invalid index we will get an error:



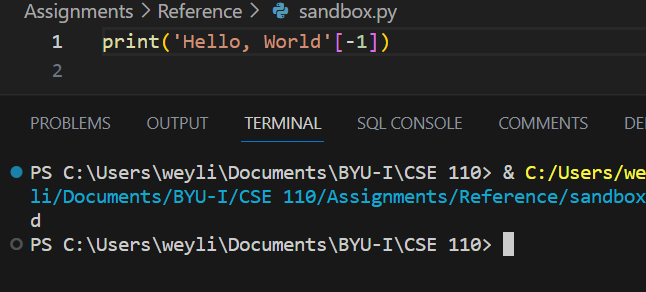
Note Python throw the IndexError and told us the string index out of range.

In this case the highest index we can use is 5. Anything higher will result in an error because it does not exist.

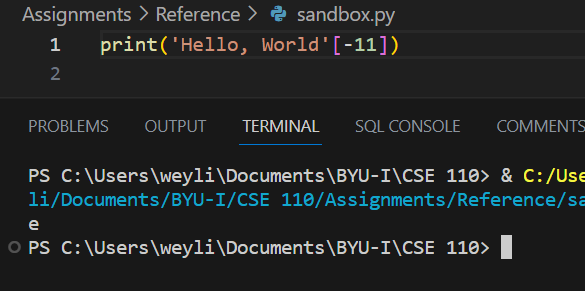
We can also do negative indexes to count starting at the back of the string. Note, we doing negative indexes the counting system starts at -1.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| -12 | -11 | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | 2- | -1 |
| H | E | L | L | O | , |  | W | O | R | L | D |

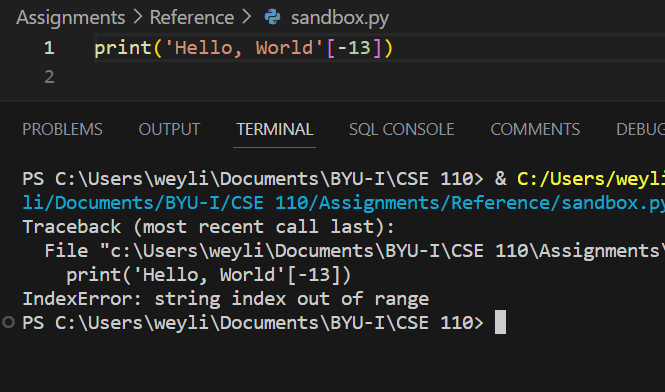
Example:



Example 2:



Example 3 produces an out of error range:



In Python, string slicing allows you to extract a portion of a string by specifying a range of indices. Slicing creates a new string containing the selected part of the original string, without modifying the original string. The syntax for string slicing is as follows:



Here's an explanation of each part of the slicing syntax:

**start**: The index from where the slice begins (inclusive). If not provided, it defaults to 0 (start of the string).

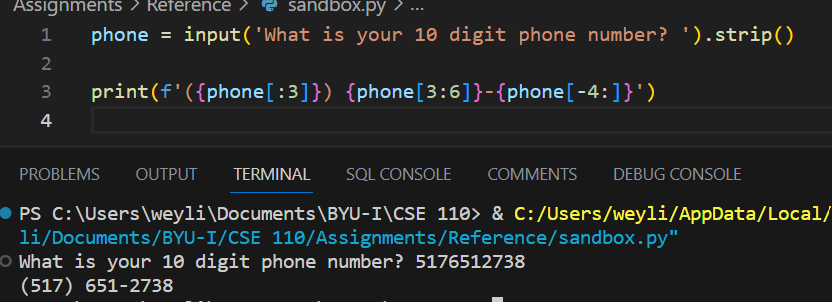
**stop**: The index where the slice ends (exclusive). The slice includes all characters up to, but not including, the element at this index. If not provided, it defaults to the end of the string.

**step**: The step value determines the increment between characters in the slice. If not provided, it defaults to 1, which means all characters within the specified range are included. A negative step value allows you to traverse the string in reverse.

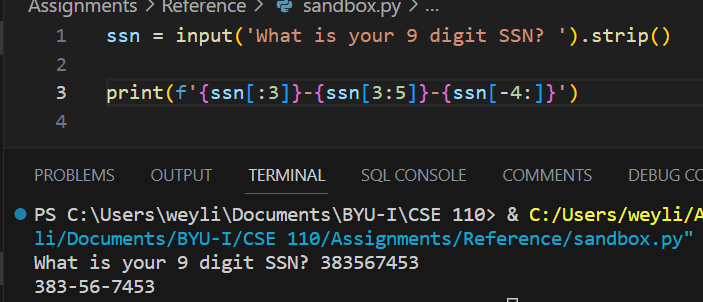


Keep in mind that when using slicing, the original string remains unchanged, and a new string object is created with the selected portion. String slicing is a powerful tool that allows you to extract substrings, manipulate text, and perform various string operations in Python.

Example on how to use f string formatting and slicing a string:



Example SSN:



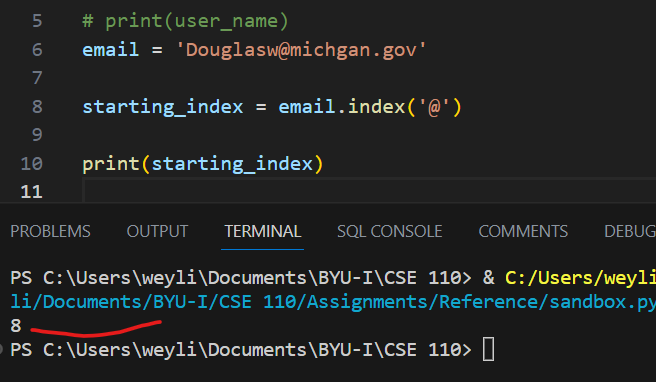
**Section 6.1**

**Finding indexes based on a character:**

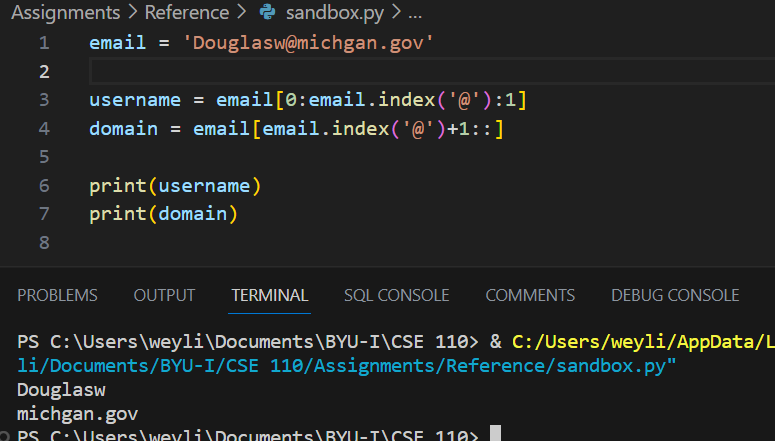
**Video Resource:**

<https://www.youtube.com/watch?v=_wmgM8xP5G4>

You can find the index position of a character then use that number to perform a slice:



Note the @ symbol is at the index of 8

We can add to the index to get the desired results as well  


References:

<https://www.w3schools.com/python/gloss_python_string_slice.asp>

Video: <https://www.youtube.com/watch?v=3PGcu6Jv9c0>

**Section 7.0:**

**Numbers and math:**

Math operations: https://www.programiz.com/python-programming/operators

Math functions: https://www.w3schools.com/python/python\_math.asp

****

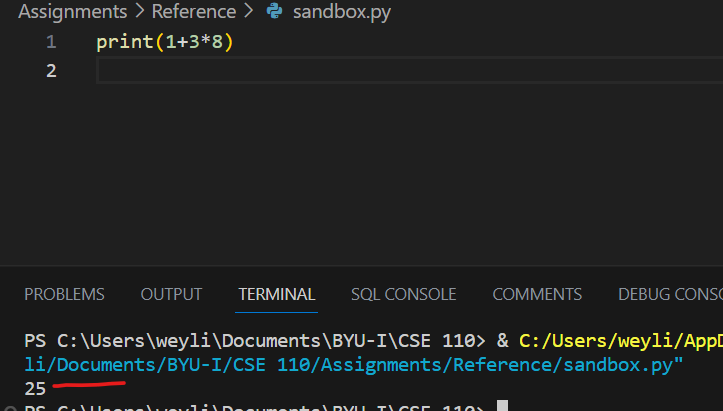
Working with numbers: <https://www.youtube.com/watch?v=5yhn0MFLcu8&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=14>

Demo numbers: <https://www.youtube.com/watch?v=T1j2tfZK7OI&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=15>

Python knows math:

****

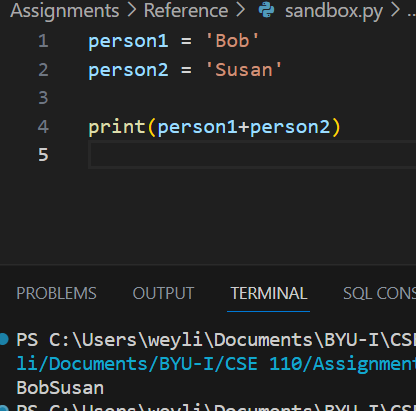
**Python also understands the order of operations**:



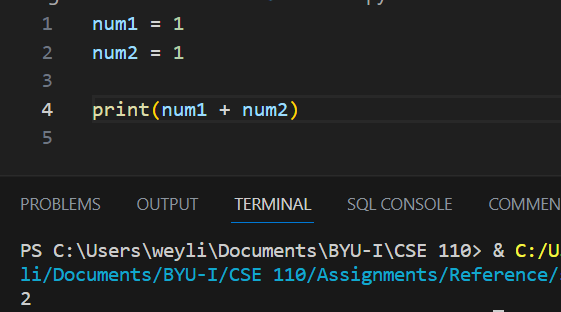
**Concatenate strings vs numbers**

Where people get confused tell the computer when to add two number together or concatenate two strings together.

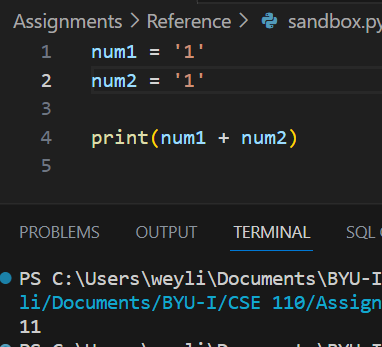
Remember: Bob and Susan concatenated is BobSusan



When we add 1 + 1 we get two:

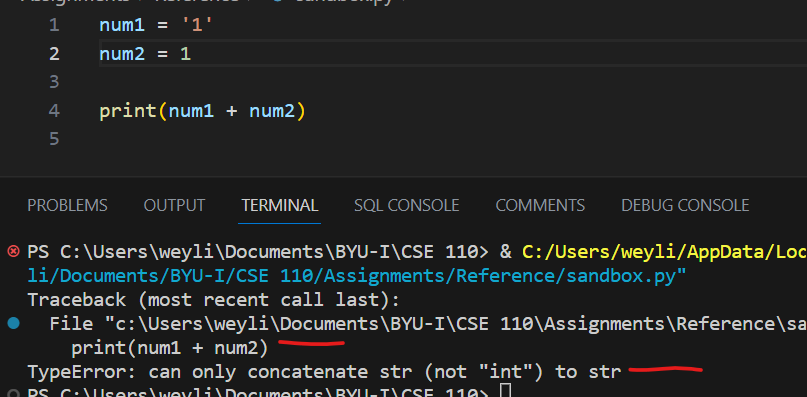


Where people start getting confused is add the string of 1 to the string of 1



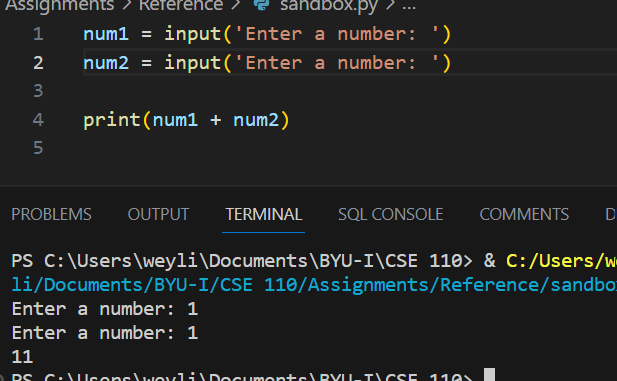
Anytime we have a string and we concatenate them together, python just push the two items together.

If you try adding the string of ‘1’ to the number 1 Python will throw an error:



It’s telling us I have no idea how to add a number to a string it doesn’t make sense. What is 4 + bob?

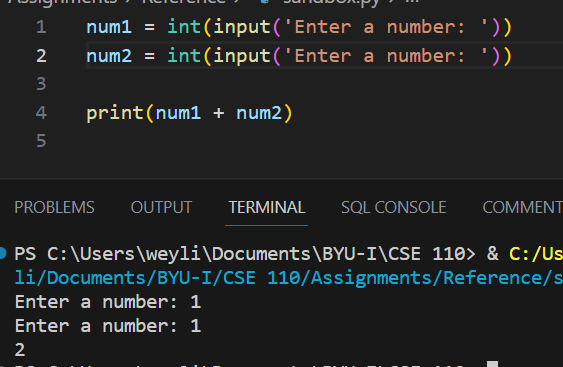
Another gotcha to consider is when a user input’s data it’s always returned as a string:



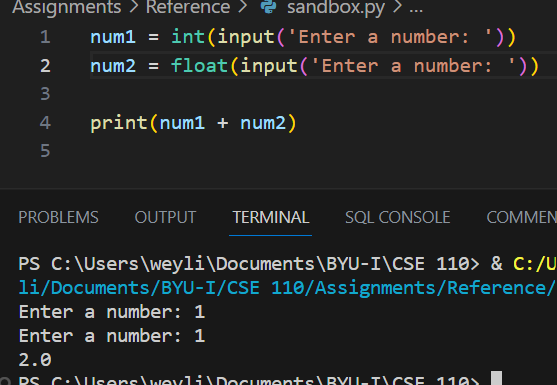
Python concatenated two strings together here.

**Casting Numbers**:

If you want to do math on the variables you must cast them to a number.

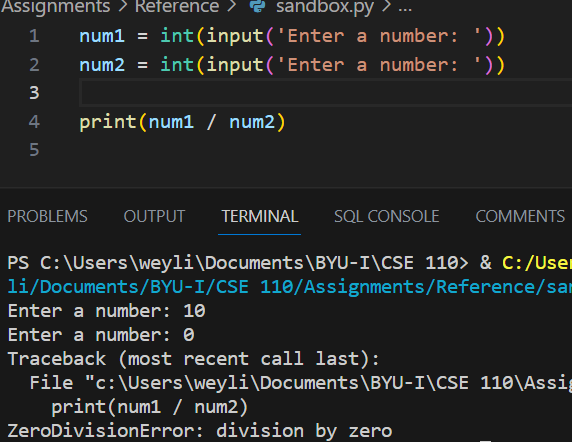


You may ever do one number as an int and one number as a float, but remember Python will always return a float.



**Division by Zero:**

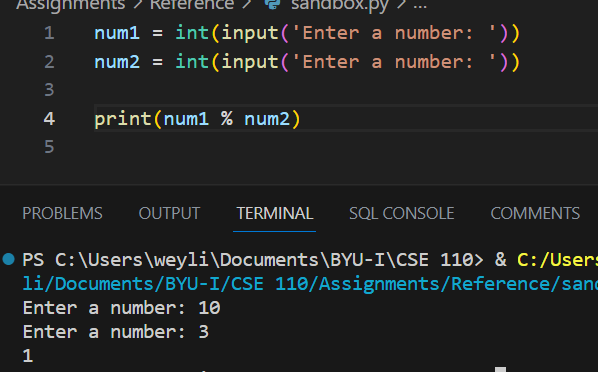
When dividing – Remember you may not divide by 0



ZeroDivisionError is raised above

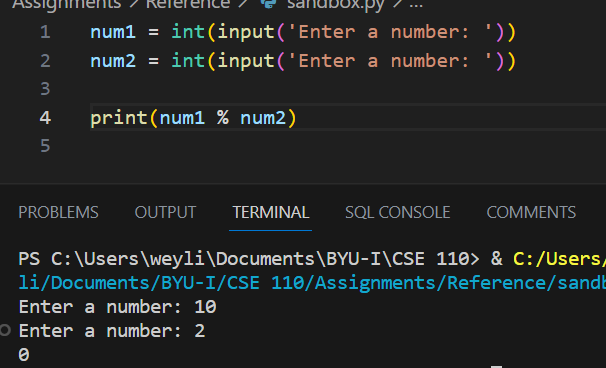
**Modulo Operator:**

The modulo property can be confusing for people newer to programming.



Here 10/3 is reminder 1

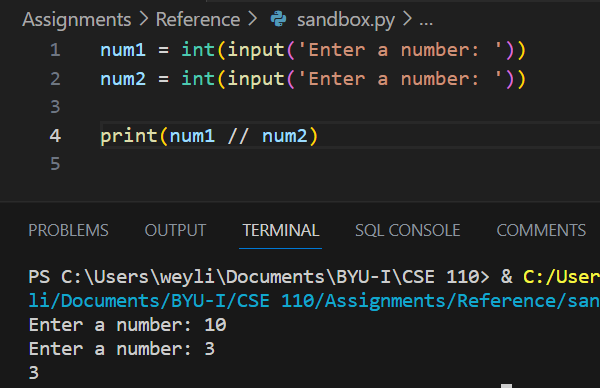
In another example we can see if a number is even or odd:



If reminder is 0 then even else odd.

**Floor Division:**

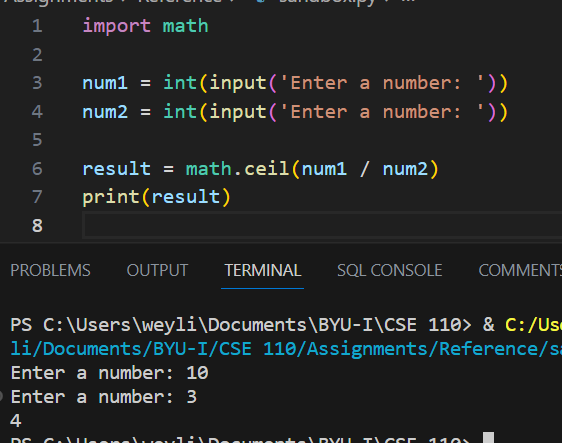
Another math function not used often is the floor division:



10 / 3 is 3.3333 however because we floor it then it drop everything after the decimal so 3 is the final answer.

**Ceiling Division:**

We can do ceiling as well; however we have to import an independent library for that.



10/3 is 3.33333 however ceiling rounds the number up no matter what, so the answer is 4.

**Python Math Library:**

The Python **math** library provides various mathematical functions for performing common mathematical operations. Here are some of the basic functions available in the **math** library:

**math.ceil(x)**: Returns the smallest integer greater than or equal to **x**.

**math.floor(x)**: Returns the largest integer less than or equal to **x**.

**math.trunc(x)**: Truncates the decimal part of **x** and returns the integer value (no rounding).

**math.sqrt(x)**: Returns the square root of **x**.

**math.pow(x, y)**: Returns **x** raised to the power of **y**.

**math.exp(x)**: Returns the exponential value of **x** (e^x).

**math.log(x, base)**: Returns the logarithm of **x** with the specified **base** (default is natural logarithm).

**math.log10(x)**: Returns the base-10 logarithm of **x**.

**math.sin(x)**, **math.cos(x)**, **math.tan(x)**: Trigonometric functions (sine, cosine, tangent) using radians as input.

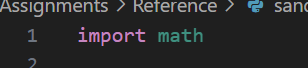
**math.radians(x)**: Converts the angle **x** from degrees to radians.

**math.degrees(x)**: Converts the angle **x** from radians to degrees.

**math.pi**: A constant representing the value of π (pi).

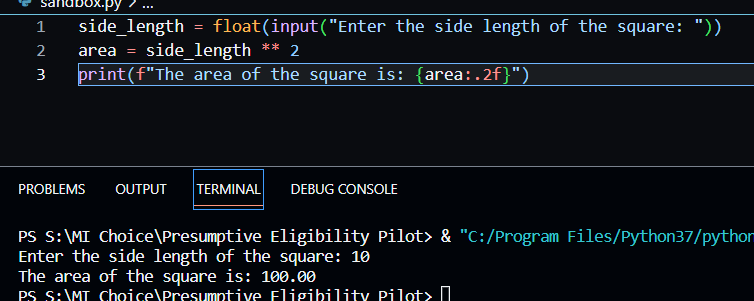
**math.e**: A constant representing the value of Euler's number (e).

To import the math library you just need the following code:



**Some common shape formulas:**

**Square:**



**Triangle:**

A screenshot of a computer

Description automatically generated with medium confidence

**Circle:**

A screenshot of a computer

Description automatically generated with medium confidence

**Cylinder:**

To calculate the lateral surface area and the total surface area of a cylinder, you can use the following formulas:

Lateral Surface Area = 2 \* π \* radius \* height Total Surface Area = 2 \* π \* radius \* (height + radius)

Here's the Python code to calculate the lateral surface area and the total surface area of a cylinder:

python

A screenshot of a computer

Description automatically generated with medium confidence

**Speed of a falling object:**

The speed of a falling object can be calculated using the formula:

Speed = √(2 \* acceleration due to gravity \* height)

where the acceleration due to gravity is approximately 9.81 m/s^2 on the surface of the Earth. Here's the Python code to calculate the speed of a falling object:

A screenshot of a computer

Description automatically generated with medium confidence

**Section 8.0**

**Comments:**

[Python Comments (w3schools.com)](https://www.w3schools.com/python/python_comments.asp)

<https://www.youtube.com/watch?v=Y58AyTR3fX0>

Writing comments in Python is essential for making your code more readable and maintainable. Comments are lines in your code that are ignored by the Python interpreter and serve as explanatory text to help you and others understand the code's purpose and functionality. Here are some notes on writing comments in Python:

Single-line comments: To write a single-line comment, use the hash symbol (#) followed by the comment text. Anything after the hash symbol on the same line will be treated as a comment and will not be executed by Python.

Text

Description automatically generated

Multi-line comments: Python does not have a specific syntax for multi-line comments like some other languages do. However, you can use triple quotes (''' or """) to create multi-line strings, which can serve as multi-line comments.Text

Description automatically generated

**Section 8.1 Python add ons:**

**Time delay**

import time

for i in range(1,10000):

    print("Hello")

    time.sleep(10)

    i+=1

**Section 9.0**

**Creating Colors:**

Create a variable to hold the colors:

Text

Description automatically generated

Create a f string to call the color and reset it back other the whole terminal will stay that color.

Note the red continues on because I did not reset. You can see the name “Renae” in red:

Text

Description automatically generated

With reset back to white:

Text

Description automatically generated

Website: <https://www.lihaoyi.com/post/BuildyourownCommandLinewithANSIescapecodes.html>

**Section 10.0:**

**Comparison Operators:**

Table

Description automatically generated with medium confidence

In Python, logical comparison operators are used to compare two or more values. They evaluate the values and return either True or False based on the condition. These operators play a crucial role in conditional statements, such as if, elif, and else, and loop statements, such as while and for.

List of Logical Comparison Operators

Here are the logical comparison operators used in Python:

Equal (==): Checks if the value of two operands is equal. If yes, the condition becomes True.

Example:

Graphical user interface, text

Description automatically generated

Not Equal (!=): Checks if the value of two operands is not equal. If the values are not equal, the condition becomes True.

Example:

Text

Description automatically generated with medium confidence

Greater Than (>): Checks if the value of the left operand is greater than the value of the right operand. If yes, the condition becomes True.

Example:

Graphical user interface, text

Description automatically generated with medium confidence

Less Than (<): Checks if the value of the left operand is less than the value of the right operand. If yes, the condition becomes True.

Example:

Graphical user interface, text

Description automatically generated with medium confidence

Greater Than or Equal to (>=): Checks if the value of the left operand is greater than or equal to the value of the right operand. If yes, the condition becomes True.

Example:

Graphical user interface, text

Description automatically generated

Less Than or Equal to (<=): Checks if the value of the left operand is less than or equal to the value of the right operand. If yes, the condition becomes True.

Example:

Text

Description automatically generated

Practical Usage with Conditional Statements

Python logical comparison operators are commonly used in conditional statements.

Here's an example using the if statement:

Text

Description automatically generated

In this example, the program will print 'a is greater than b.' because 5 is indeed greater than 3.

You can also use multiple conditions using logical operators like and and or. Here's an example:

A screenshot of a computer

Description automatically generated with medium confidence

In this example, the program will print 'Both conditions are true.' because 5 is greater than 3 and 7 is greater than 5.

**Section 11.0:**

**Logical Operators:**

In Python, logical operators are used to combine conditional statements, and they return True or False depending on the condition. There are three logical operators: and, or, and not.

List of Logical Operators

AND (and): If both the operands (conditions) are True, then the condition becomes True.

Example:

A screenshot of a computer

Description automatically generated with medium confidence

OR (or): If any of the two operands (conditions) is True, then the condition becomes True.

Example:

Text

Description automatically generated

NOT (not): Used to reverse the logical state of its operand. If a condition is True, the not operator makes it False and vice versa.

Example:

Graphical user interface, text, application

Description automatically generated

Practical Usage with Conditional Statements

Logical operators are often used in if statements to combine multiple conditions.

Here's an example using the if statement with the and operator:

A screenshot of a computer

Description automatically generated with medium confidence

In this example, the program will print 'Both conditions are true.' because 5 is greater than 3 and 7 is greater than 5.

You can also use multiple operators in one statement. Here's an example:

Text

Description automatically generated

In this example, the program will print 'At least one condition is true.' because even though the second condition (not c > a) is False, the or operator only requires one condition to be True.

It's important to remember that Python evaluates conditions from left to right. This means in a combined statement using and or or, Python will stop evaluating as soon as it finds the result. For the and operator, if the first condition is False, Python stops immediately and returns False. For the or operator, if the first condition is True, Python stops and returns True.

**Section 12.0:**

**Conditional statements:**

**Video References:**[**https://www.youtube.com/watch?v=5pPKYWqkoek&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=19**](https://www.youtube.com/watch?v=5pPKYWqkoek&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=19)

[Demo: Conditional Logic | Python for Beginners [20 of 44] - YouTube](https://www.youtube.com/watch?v=zqVmqtTLmgw&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=20)

[Handling Multiple Conditions | Python for Beginners [21 of 44] - YouTube](https://www.youtube.com/watch?v=oYaGJBMoXok&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=21)

[Demo: Multiple Conditions | Python for Beginners [22 of 44] - YouTube](https://www.youtube.com/watch?v=J9luo4cODzM&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=22)

[Complex Conditions | Python for Beginners [23 of 44] - YouTube](https://www.youtube.com/watch?v=IBOHc87yFYw&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=23)

[Demo: Complex Conditions | Python for Beginners [24 of 44] - YouTube](https://www.youtube.com/watch?v=Iui6K2STtbA&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=24)

In Python, condition statements are used to control the flow of execution based on certain conditions. The most common types of condition statements in Python are:

if statement: The if statement is used to execute a block of code if a certain condition is true. It is often used as a basic branching mechanism.

Shape, rectangle

Description automatically generated

if-else statement: The if-else statement allows you to execute one block of code if the condition is true, and another block if the condition is false.

Graphical user interface

Description automatically generated

if-elif-else statement: The if-elif-else statement lets you test multiple conditions and execute different blocks of code based on the first condition that evaluates to true. The elif stands for "else if".

Text

Description automatically generated

Nested if statements: You can also nest condition statements within each other to handle more complex cases.

Text

Description automatically generated

Ternary Operator: Python allows you to write simple conditional expressions in a more compact form using the ternary operator.

Graphical user interface, text

Description automatically generated

Example:

Text

Description automatically generated

Example:

Text

Description automatically generated

Nested if statement example:

Text

Description automatically generated

Ternary Operator example:

A screenshot of a computer

Description automatically generated with medium confidence

Guess the number game:

import random

count=0

user\_guess=-1

number = random.randint(0,100)

while user\_guess != number:

    user\_guess = int(input("What number do you guess? "))

    count +=1

    if user\_guess > number:

        print("The number is lower")

    elif user\_guess < number:

        print("The number is higher")

    else:

        user\_guess == number

        print("You guessed the number")

        print(f"It took you {count} to guess the correct number")

        break

W3Shool conditional practice: [Python Conditions (w3schools.com)](https://www.w3schools.com/python/python_conditions.asp)

General validation with conditions

**Section 13.0:**

**Special or Escape Characters:**

**Video Resource:**

<https://www.youtube.com/watch?v=r2CtqCPJyKs>

Special characters, also known as escape characters, in Python are used in strings to represent certain whitespace characters and to execute certain commands such as creating a new line, adding a tab, etc. Escape characters are prefixed with a backslash (\).

Here's a list of some common special or escape characters in Python:

**New Line (\n):** Creates a new line in the text at the point where it is used.

Example:

Text

Description automatically generated

**Tab (\t):** Creates a tab space in the text at the point where it is used.

Example:

Graphical user interface, text

Description automatically generated

**Backslash (\\):** Creates a backslash character in the text at the point where it is used.

Example:

Graphical user interface, text

Description automatically generated

**Single Quote (\'):** Allows inclusion of single quotes within a string enclosed by single quotes.

Example:

Graphical user interface, text

Description automatically generated

**Double Quote (\"):** Allows inclusion of double quotes within a string enclosed by double quotes.

Example:

Graphical user interface

Description automatically generated

**Backspace (\b):** Prints backspace, but this works in string output in consoles and terminals, not in most of the modern IDEs.

Example:

Graphical user interface

Description automatically generated

**Carriage Return (\r):** Moves all characters after (and including) it to the beginning of the string.

Example:

Text

Description automatically generated

**Form Feed (\f):** Works as a page break in a text document. Similar to backspace, it doesn't work in modern IDEs but works on consoles and terminals.

Example

A screenshot of a computer

Description automatically generated with medium confidence

These escape characters are especially useful when you need to insert characters that are not easily entered into a string, or when you want to insert special commands into a string.

Please note that if you do not want characters prefaced by \ to be interpreted as special characters, you can use raw strings by adding an r before the first quote.

Graphical user interface, text

Description automatically generated

**Section 14.0**

Try/Except/Finally:

Video Reference:

<https://www.youtube.com/watch?v=HQqqNBZosn8&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=17>

<https://www.youtube.com/watch?v=LrRh-V-hYEc&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=18>

In Python, the try/except block is used for exception handling. It allows the program to continue running even if it encounters an error or an exception.

Syntax

The basic syntax for a try/except block is as follows:

Text

Description automatically generated

How it Works

**try:** The try block contains a segment of code that might potentially raise an exception. If everything runs smoothly, the code executes and the except block is skipped.

**except**: If an exception or error occurs within the try block, the try block immediately terminates, and the program executes the except block.

Examples

Here is a simple example:

Text

Description automatically generated

In this example, trying to divide by zero raises a ZeroDivisionError, so the program prints "You can't divide by zero!".

**Catching Multiple Exceptions**

You can have multiple except blocks to catch different types of exceptions. Here's an example:

A screenshot of a computer

Description automatically generated with medium confidence

**The else Block**

An else block can be included after the except block(s), which is executed if the code in the try block does not raise an exception.

Text

Description automatically generated

**The finally Block**

A finally block can also be included, which will be executed no matter whether an exception is raised or not.

Text

Description automatically generated

This is particularly useful when you need to ensure that certain resources, like open files or network connections, are properly closed whether an exception was raised or not.

In summary, the try/except block in Python is a powerful tool for handling exceptions and errors, which can help you make your programs more robust and reliable.

**Section 15.0**

**While Loops:**

**Video Resource:**

<https://video.byui.edu/media/t/1_5faplwse>

Introduction

Python is a versatile programming language that allows you to repeat certain tasks using loops. One of the most commonly used loops is the while loop, which repeatedly executes a block of code as long as a specified condition remains true. This user guide aims to introduce you to the basics of using while loops in Python.

Prerequisites

Before using while loops in Python, you should have a basic understanding of the following:

Python programming fundamentals

Conditional statements (if, else, elif)

Basic arithmetic and logical operations

Using the while Loop

The syntax for the while loop is as follows:

Text

Description automatically generated

Here's how it works:

The condition is evaluated before entering the loop. If it is True, the code inside the loop is executed.

After executing the code block, the condition is re-evaluated.

If the condition is still True, the loop continues to execute. If the condition is False, the loop terminates, and the program moves on to the next line of code after the loop.

Examples

Example 1: Simple While Loop

Text

Description automatically generated

Examples

**Example 1: Simple While Loop**

Text

Description automatically generated with medium confidence

Shape, rectangle

Description automatically generated

**Example 2: Using Input to Terminate the Loop**

Text

Description automatically generated

This example will repeatedly ask the user to enter a value until they type 'quit'.

Best Practices

To avoid infinite loops and create efficient code, consider the following best practices:

Ensure the Loop Ends: Always make sure that the condition inside the while loop eventually becomes False. Otherwise, you'll end up with an infinite loop that will keep executing indefinitely.

Initialize Variables: Initialize any variables used inside the loop before the loop starts to avoid unexpected behavior.

Avoid Unnecessary Complexity: Keep the code inside the loop concise and easy to understand. Complex logic can lead to bugs that are hard to trace.

Use for Loops When Appropriate: If you know the number of iterations beforehand, consider using for loops instead of while loops. for loops are more straightforward and generally easier to control.

Break Statement: You can use the break statement to exit the while loop prematurely based on a certain condition.

**Conclusion**

Congratulations! You now have a basic understanding of using while loops in Python. You can use while loops to repeat actions as long as a particular condition is true, making your programs more dynamic and versatile. Remember to be careful with the loop condition to avoid infinite loops and to keep your code clean and efficient. Happy coding!

**Section 15.1**

**Data validation with While Loops:**

**Video Resource:**

<https://www.youtube.com/watch?v=akDSbJJI-J0>

User Guide: Using While Loops in Python to Validate User Input

Introduction

Validating user input is an essential aspect of programming to ensure that your program processes data correctly and securely. Python's while loop provides a powerful tool to repeatedly prompt users for input until they provide valid data. This user guide will walk you through the steps of using a while loop to validate user input in Python.

Prerequisites

Before proceeding, make sure you have a basic understanding of the following:

Python programming fundamentals

Basic input/output operations (input() function)

Basic data types and data validation

Using While Loops for User Input Validation

The general approach for using a while loop to validate user input involves the following steps:

Initialize a variable to store the user input.

Create a while loop that runs until the user provides valid input.

Inside the loop, prompt the user for input using the input() function.

Validate the input using conditional statements.

If the input is valid, break out of the loop; otherwise, continue the loop and prompt the user again.

Let's look at an example of using a while loop to validate user input for an integer:

Text

Description automatically generated

In this example, we use a while True loop to repeatedly prompt the user for input. The try block attempts to convert the user input to an integer using int(). If the conversion is successful, the break statement will exit the loop. If the conversion raises a ValueError, it means the user provided invalid input, and the except block will be executed, displaying an error message and prompting the user again.

Example: Validating Floating-Point Numbers

Let's consider another example, where we validate user input for a floating-point number:

Text

Description automatically generated

This example follows a similar approach as before, but instead of using int(), we use float() to convert the input into a floating-point number.

**Additional Validation**

You can add more complex validation logic inside the try block based on your specific use case. For example, you can check if the input falls within a particular range or meets certain conditions using if statements. If the input does not meet the required criteria, you can raise custom exceptions or display appropriate error messages.

**Conclusion**

Using while loops to validate user input in Python provides a robust way to ensure that your programs handle user data correctly. By combining the while loop with the try-except block, you can repeatedly prompt users for input until they provide valid data. Remember to provide informative error messages to guide users towards providing the correct input. Happy coding!

**Section 15.2**

**Break and Continue:**

**Video Resource:**

[**https://www.youtube.com/watch?v=4qw5atfs\_As**](https://www.youtube.com/watch?v=4qw5atfs_As)

**The break Statement**

The break statement is used to exit a loop prematurely when a certain condition is met. It allows you to stop the loop's execution even before the loop's condition becomes false. The syntax for using break is straightforward:

Text

Description automatically generated

**Here's how it works:**

The condition is evaluated before each iteration of the loop.

If the condition is True, the code block inside the loop is executed.

Inside the loop, you can use an if statement to check if a specific condition (some\_condition) is met.

If the some\_condition is True, the break statement is executed, and the loop is terminated immediately.

Example: Using break

Text

Description automatically generated

In this example, the loop will keep prompting the user for input until they enter 'quit', at which point the loop will terminate due to the break statement.

**The continue Statement**

The continue statement is used to skip the rest of the current iteration of the loop and proceed to the next iteration. It allows you to effectively skip some specific steps or actions inside the loop based on certain conditions. The syntax for using continue is as follows:

Text

Description automatically generated

Here's how it works:

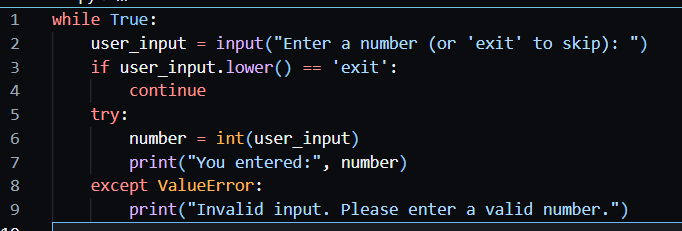
The condition is evaluated before each iteration of the loop.

If the condition is True, the code block inside the loop is executed.

Inside the loop, you can use an if statement to check if a specific condition (some\_condition) is met.

If the some\_condition is True, the continue statement is executed, and the rest of the code inside the loop for the current iteration is skipped. The loop proceeds to the next iteration.

Example: Using continue



In this example, if the user enters 'exit', the continue statement will skip the rest of the code inside the loop for that iteration, and it will prompt the user again for input without attempting to convert the input to an integer.

**Best Practices**

When using break and continue statements in while loops, consider the following best practices:

Use with Caution: While break and continue can be powerful, using them excessively can make the code harder to read and understand. Make sure to use them only when they improve code clarity and efficiency.

Avoid Infinite Loops: Be cautious when using break in nested loops to avoid unintentional infinite loops. Ensure that the loop condition is eventually met or that the break is triggered based on an appropriate condition.

Use Clear Conditions: Make sure the conditions for using break and continue are clear and easily understandable by other developers who might read your code.

**Conclusion**

Using break and continue statements in while loops allows you to have more control over the loop's execution flow. break helps you terminate the loop prematurely, while continue allows you to skip specific iterations. By using these statements judiciously, you can write more efficient and concise code for handling different scenarios in Python programs. Happy coding!

**Section 15.3**

**Be careful about Infinite loops:**

Introduction

An infinite loop is a loop that continues to execute indefinitely without ever terminating. This can happen due to a flawed loop condition or a missing mechanism to break out of the loop. Infinite loops can cause your program to become unresponsive, consume excessive CPU resources, and even crash the entire system. In this user guide, we'll explore why it's essential to avoid infinite loops in programming and provide examples of how they can occur.

**The Dangers of Infinite Loops**

**1. Unresponsive Program**

When an infinite loop occurs, the program gets stuck in an endless repetition of the loop's code block. As a result, the program fails to move beyond that point, making it unresponsive to user input or any other commands. This can be frustrating for users and may lead them to force quit the program.

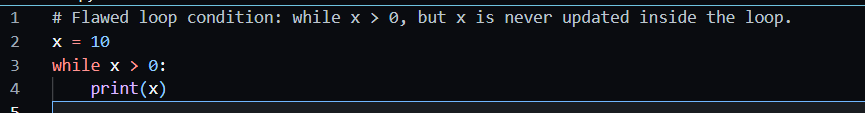
**2. High CPU Usage**

Infinite loops consume CPU resources without accomplishing any meaningful tasks. As the loop continuously executes, it keeps the CPU busy, causing it to run at maximum capacity. This high CPU usage can slow down the system and impact the performance of other programs running simultaneously.

**3. System Crash**

In some cases, an infinite loop can lead to a system crash. If the loop continuously consumes resources, it can cause memory exhaustion or other critical issues, resulting in a system-wide failure.

Example 1: Infinite Loop Due to Flawed Condition



In this example, the value of x is initially set to 10. The loop condition checks if x > 0, which is true. However, the value of x is never updated inside the loop, so the condition remains true for all subsequent iterations, leading to an infinite loop.

Example 2: Infinite Loop with No Break Mechanism

# Infinite loop with no break mechanism

while True:

    user\_input = input("Enter a number (or 'exit' to quit): ")

    if user\_input.lower() == 'exit':

        # Missing break statement, so the loop will continue indefinitely

        continue

    print("You entered:", user\_input)

In this example, the loop condition while True ensures that the loop will run forever. Although there is an if statement to check if the user input is 'exit', there's no break statement to terminate the loop. As a result, the loop will keep running indefinitely, asking for user input and processing it without a way to exit.

How to Avoid Infinite Loops

To avoid infinite loops, follow these best practices:

**Double-Check Loop Conditions**: Always ensure that the loop condition can eventually become False. Verify that the condition involves variables that change during the loop's execution.

**Break Mechanism**: If you use a loop that should terminate under specific conditions, implement a proper break mechanism. Use break statements inside the loop to exit when necessary.

**Test with Small Inputs**: Before running a loop with extensive data, test it with small inputs to ensure it behaves as expected. This helps catch any potential infinite loop issues early in development.

**Use Debugger and Print Statements**: If you suspect an infinite loop, use a debugger or insert print statements within the loop to track its progress and identify any issues.

**Conclusion**

Avoiding infinite loops is crucial to maintain the stability and responsiveness of your programs. Infinite loops can lead to unresponsive programs, high CPU usage, and even system crashes. Always double-check your loop conditions, implement proper break mechanisms, and thoroughly test your code to ensure it behaves as expected and doesn't enter into infinite loops. By following these practices, you can write more robust and reliable programs. Happy coding!

**Section 15.4**

**For Loops :**

**Video Resource:**

[Loops | Python for Beginners [27 of 44] - YouTube](https://www.youtube.com/watch?v=LrOAl8vUFHY&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=27)

[Demo: Loops | Python for Beginners [28 of 44] - YouTube](https://www.youtube.com/watch?v=rAvD-6MpTw4&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=28)

**Introduction**

Python's for loop is a powerful control flow statement that allows you to iterate over a sequence of elements, such as lists, tuples, strings, or other iterable objects. With the for loop, you can perform repetitive tasks on each item in the sequence without having to write the iteration logic explicitly. This user guide will introduce you to the basics of using for loops in Python.

**Prerequisites**

Before using for loops, you should have a basic understanding of the following:

**Python programming fundamentals**

Basic data types, such as lists, tuples, and strings

Basic knowledge of control flow statements, including if and else

Syntax of the for Loop

The syntax for a for loop in Python is as follows:

for item in sequence:

    # Code block to be executed for each item in the sequence

**Here's how it works:**

The for keyword starts the loop declaration.

item represents a variable that will take on the value of each element in the sequence.

The sequence can be any iterable object like a list, tuple, string, dictionary, etc.

The code block indented below the for statement is executed for each item in the sequence.

Examples of Using for Loops

Example 1: Iterating Over a List

fruits = ['apple', 'banana', 'orange', 'grape']

for fruit in fruits:

    print(fruit)

**Results:**

Text

Description automatically generated

**Example 2: Iterating Over a String**

message = "Hello, Python!"

for char in message:

    print(char)

A screenshot of a computer

Description automatically generated with medium confidence

**Example 3: Using the range() Function**

The range() function is commonly used to generate a sequence of numbers that can be used in for loops.

for num in range(1, 6):

    print(num)

Shape, rectangle

Description automatically generated

The range() function generates numbers from the starting value (inclusive) to the ending value (exclusive). In this example, range(1, 6) generates numbers from 1 to 5.

**Nesting for Loops**

You can also nest for loops within each other to perform complex iterations, such as iterating over lists of lists or performing matrix operations.

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for row in matrix:

    for num in row:

        print(num, end=' ')

    print()

Shape, rectangle

Description automatically generated

break and continue in for Loops

You can use the break and continue statements inside for loops in the same way as in while loops. They provide control over the loop's execution, allowing you to break out of the loop prematurely or skip certain iterations based on specific conditions.

**Conclusion**

Congratulations! You now have a basic understanding of for loops in Python. They are powerful constructs for iterating over sequences, enabling you to perform repetitive tasks on each element of a list, tuple, string, or other iterable objects. Remember to use the correct indentation and be careful with break and continue statements when using nested loops. Happy coding!

**Section 16.0**

**Creating and working with Python List:**

W3Schools: <https://www.w3schools.com/python/python_lists.asp>

Note do all of the list in w3Schools:

Graphical user interface, application

Description automatically generated

**Creating Lists**

In Python, you can create a list by enclosing comma-separated elements within square brackets [ ].

# Example of creating a list

numbers = [1, 2, 3, 4, 5]

fruits = ['apple', 'banana', 'orange']

mixed\_list = [1, 'apple', True, 3.14]

**Accessing List Elements**

You can access individual elements in a list using their index. The index starts from 0 for the first element and increments by 1 for each subsequent element.

fruits = ['apple', 'banana', 'orange']

print(fruits[0])  # Output: 'apple'

print(fruits[2])  # Output: 'orange'

Basic List Operations

**1. Length of a List**

You can find the number of elements in a list using the len() function.

numbers = [1, 2, 3, 4, 5]

print(len(numbers))  # Output: 5

**2. Adding Elements to a List**

You can append elements to the end of a list using the append() method.

fruits = ['apple', 'banana', 'orange']

fruits.append('grape')

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

**3. Removing Elements from a List**

You can remove elements from a list using the remove() method.

fruits = ['apple', 'banana', 'orange']

fruits.remove('banana')

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

**4. Slicing Lists**

You can extract a portion of a list using slicing. Slicing creates a new list containing elements from a specified start index (inclusive) to an end index (exclusive).

numbers = [1, 2, 3, 4, 5]

subset = numbers[1:4]

print(subset)  # Output: [2, 3, 4]

**Looping Through Lists**

Lists are often used in conjunction with loops to perform operations on each element of the list.

fruits = ['apple', 'banana', 'orange']

for fruit in fruits:

    print(fruit)

Shape, rectangle

Description automatically generated

**List Comprehensions**

List comprehensions are a concise and efficient way to create lists based on existing lists or other iterables.

numbers = [1, 2, 3, 4, 5]

squared\_numbers = [num\*\*2 for num in numbers]

print(squared\_numbers)  # Output: [1, 4, 9, 16, 25]

**Conclusion**

Congratulations! You now have a basic understanding of using lists in Python. Lists are versatile data structures that allow you to store collections of elements and perform various operations on them. They are an essential tool for any Python programmer. With this knowledge, you can confidently work with lists and leverage their power to solve a wide range of programming problems. Happy coding!

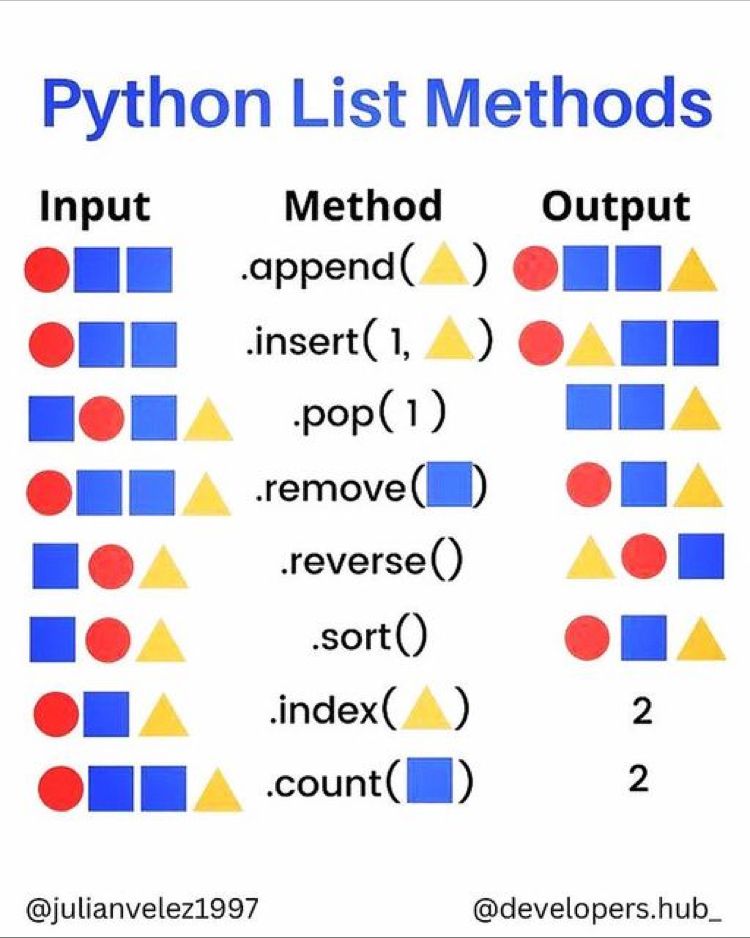
**Section 17.0:**

**Essential List Methods in Python:**

**Video Resource:**

[Collections | Python for Beginners [25 of 44] - YouTube](https://www.youtube.com/watch?v=beA8IsY3mQs&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=25)

[Demo: Collections | Python for Beginners [26 of 44] - YouTube](https://www.youtube.com/watch?v=4PaSlXNjawM&list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6&index=26)



Text

Description automatically generated with low confidence

Creating a Sample List

To demonstrate the list methods, let's start by creating a sample list:

fruits = ['apple', 'banana', 'orange', 'apple', 'mango']

**append():** Add an Element at the End of the List

The append() method is used to add an element to the end of the list.

fruits.append('grape')

print(fruits)  # Output: ['apple', 'banana', 'orange', 'apple', 'mango', 'grape']

**clear():** Remove All Elements from the List

The clear() method removes all elements from the list, making it an empty list.

fruits.clear()

print(fruits)  # Output: []

**copy():** Return a Copy of the List

The copy() method creates a shallow copy of the list.

fruits = ['apple', 'banana', 'orange', 'apple', 'mango']

fruits\_copy = fruits.copy()

print(fruits\_copy)  # Output: ['apple', 'banana', 'orange', 'apple', 'mango']

**count():** Return the Number of Occurrences of an Element

The count() method returns the number of occurrences of a specified element in the list.

fruits = ['apple', 'banana', 'orange', 'apple', 'mango']

count\_apple = fruits.count('apple')

print(count\_apple)  # Output: 2

**extend():** Add Elements from Another List (or Any Iterable)

The extend() method adds elements from another list (or any iterable) to the end of the current list.

fruits = ['apple', 'banana', 'orange']

more\_fruits = ['mango', 'grape']

fruits.extend(more\_fruits)

print(fruits)  # Output: ['apple', 'banana', 'orange', 'mango', 'grape']

**index():** Return the Index of the First Occurrence of an Element

The index() method returns the index of the first occurrence of the specified element in the list.

fruits = ['apple', 'banana', 'orange', 'mango']

index\_banana = fruits.index('banana')

print(index\_banana)  # Output: 1

**insert():** Add an Element at the Specified Position

The insert() method adds an element at the specified index in the list.

fruits = ['apple', 'banana', 'orange']

fruits.insert(1, 'mango')

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

**pop():** Remove the Element at the Specified Position

The pop() method removes and returns the element at the specified index.

fruits = ['apple', 'banana', 'orange', 'mango']

removed\_fruit = fruits.pop(2)

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

print(removed\_fruit)  # Output: 'orange'

**remove():** Remove the First Occurrence of an Element

The remove() method removes the first occurrence of the specified element from the list.

fruits = ['apple', 'banana', 'orange', 'apple', 'mango']

fruits.remove('apple')

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

**reverse():** Reverse the Order of the List

The reverse() method reverses the order of the elements in the list.

fruits = ['apple', 'banana', 'orange', 'mango']

fruits.reverse()

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

**sort():** Sort the List

The sort() method sorts the list in ascending order (for numbers) or lexicographic order (for strings).

numbers = [5, 2, 8, 1, 3]

numbers.sort()

print(numbers)  # Output: [1, 2, 3, 5, 8]

fruits = ['apple', 'banana', 'orange', 'mango']

fruits.sort()

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

Conclusion

Congratulations! You now have a comprehensive understanding of essential list methods in Python. Lists are dynamic and versatile data structures that offer a wide range of operations for adding, removing, modifying, and manipulating elements. With this knowledge, you can confidently work with lists and leverage their power to solve a wide range of programming problems. Happy coding!

**Append:**

Syntax of append()

The append() method is used on a list to add an element to its end. The syntax is as follows:

A picture containing text

Description automatically generated

list\_name: The name of the list to which the element will be added.

element: The value to be appended to the list.

Example: Using append()

Let's start by creating a list and then using the append() method to add elements to it.

# Create an empty list

fruits = []

# Add elements using the append() method

fruits.append('apple')

fruits.append('banana')

fruits.append('orange')

# Display the list

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

In this example, we first create an empty list called fruits. Then, we use the append() method to add three elements, 'apple', 'banana', and 'orange', to the list. The print() function shows the resulting list.

Appending Multiple Elements

The append() method allows you to add one element at a time. If you want to add multiple elements to the list, you can use a loop or list concatenation.

**Using a Loop**

numbers = []

for i in range(1, 6):

    numbers.append(i)

print(numbers)  # Output: [1, 2, 3, 4, 5]

In this example, we use a for loop to iterate through the range of numbers from 1 to 5, and then we use append() to add each number to the numbers list.

**Using List Concatenation**

fruits = ['apple', 'banana', 'orange']

more\_fruits = ['mango', 'grape']

fruits += more\_fruits

print(fruits)  # Output: ['apple', 'banana', 'orange', 'mango', 'grape']

In this example, we create two lists, fruits and more\_fruits, and then we use the += operator to concatenate the two lists. This operation adds all elements of more\_fruits to the end of fruits.

**Conclusion**

Congratulations! You now know how to use the append() method in Python lists to add elements at the end of the list. The append() method is a simple and efficient way to expand your lists dynamically as your program runs. With this knowledge, you can confidently use the append() method in your Python projects to manage and manipulate lists effectively. Happy coding!

**Clear Method:**

Syntax of clear()

The clear() method is used on a list to remove all its elements. The syntax is as follows:

Background pattern, rectangle

Description automatically generated

list\_name: The name of the list from which all elements will be removed.

Example: Using clear()

Let's start by creating a list, adding elements to it, and then using the clear() method to remove all the elements.

# Create a list with elements

fruits = ['apple', 'banana', 'orange', 'mango']

# Display the original list

print("Original List:", fruits)

# Clear the list using the clear() method

fruits.clear()

# Display the list after clearing

print("Cleared List:", fruits)

Text

Description automatically generated

In this example, we first create a list called fruits with four elements. We then print the original list to verify its contents. After that, we use the clear() method to remove all elements from the list. The second print statement shows the list after it has been cleared, and you can see that the list is now empty.

**Use Cases for clear()**

Resetting a List: The clear() method is useful when you want to reset a list to its initial state, removing all elements and starting with an empty list.

**Memory Management:** Clearing a list using clear() can help free up memory occupied by the elements in the list, especially if the list is no longer needed.

**Reusing a List**: After using a list for a particular task, you can clear it and reuse it for a different purpose without having to create a new list.

**Precautions**

Keep in mind the following precautions while using the clear() method:

Be careful when using clear() as it permanently removes all elements from the list. If you need to keep a copy of the original list, create a new list or make a copy of the list using slicing or the copy() method before clearing it.

# Copying the list before clearing it

fruits = ['apple', 'banana', 'orange', 'mango']

fruits\_copy = fruits.copy()

# Clear the original list

fruits.clear()

# You can still access the elements in the copied list

print(fruits\_copy)  # Output: ['apple', 'banana', 'orange', 'mango']

**Conclusion**

Congratulations! You now know how to use the clear() method in Python lists to remove all elements from a list. The clear() method is a convenient way to reset and empty a list. With this knowledge, you can confidently use the clear() method in your Python projects when you need to reset lists or free up memory occupied by lists. Happy coding!

**Copy Method:**

Syntax of copy()

The copy() method is used on a list to create a shallow copy of the list. The syntax is as follows:

Graphical user interface

Description automatically generated with low confidence

list\_name: The name of the list to be copied.

new\_list: The name of the new list that will be the shallow copy of the original list.

Example: Using copy()

Let's start by creating a list and then using the copy() method to create a shallow copy of the list.

# Create a list with elements

fruits = ['apple', 'banana', 'orange', 'mango']

# Create a shallow copy of the list

fruits\_copy = fruits.copy()

# Display both lists

print("Original List:", fruits)

print("Shallow Copy:", fruits\_copy)

Text

Description automatically generated with medium confidence

In this example, we first create a list called fruits with four elements. We then use the copy() method to create a new list called fruits\_copy, which is a shallow copy of the original list. Both lists have the same elements, but they are independent of each other.

**Shallow Copy vs. Deep Copy**

It's essential to understand the difference between a shallow copy and a deep copy when dealing with nested lists or objects. A shallow copy only copies the outer list and the references to the inner objects. However, the inner objects themselves are not duplicated. On the other hand, a deep copy creates an entirely independent copy of both the outer list and all the inner objects.

Let's look at an example to illustrate this difference:

# Nested list

original\_list = [[1, 2, 3], [4, 5, 6]]

# Create a shallow copy

shallow\_copy = original\_list.copy()

# Modify the shallow copy

shallow\_copy[0][0] = 100

# Display both lists

print("Original List:", original\_list)  # Output: [[100, 2, 3], [4, 5, 6]]

print("Shallow Copy:", shallow\_copy)    # Output: [[100, 2, 3], [4, 5, 6]]

In this example, when we modify the first element of the first inner list in the shallow copy, the same change is reflected in the original list. This is because the inner lists are still references to the same objects in memory.

**When to Use copy()**

Use the copy() method when you want to create an independent duplicate of a list but don't need a deep copy of nested objects or elements. It is an efficient way to duplicate lists, especially when you need to modify one version of the list while preserving the original.

**Conclusion**

Congratulations! You now know how to use the copy() method in Python lists to create shallow copies. The copy() method is a handy tool for duplicating lists while keeping them independent of each other. With this knowledge, you can confidently use the copy() method in your Python projects whenever you need to create identical copies of lists. Happy coding!

**Count Method:**

Syntax of count()

The count() method is used on a list to count the occurrences of a specific element. The syntax is as follows:

Graphical user interface, website

Description automatically generated with medium confidence

list\_name: The name of the list in which you want to count occurrences.

element: The value whose occurrences you want to count.

count: The variable that stores the number of occurrences of the element in the list.

Example: Using count()

Let's start by creating a list and then using the count() method to count the occurrences of a specific element.

# Create a list with elements

numbers = [1, 2, 3, 2, 4, 2, 5]

# Count the occurrences of number 2

count\_of\_twos = numbers.count(2)

# Display the count

print("Occurrences of 2:", count\_of\_twos)  # Output: Occurrences of 2: 3

In this example, we first create a list called numbers with several elements, including multiple occurrences of the number 2. We then use the count() method to count how many times the number 2 appears in the list. The result, stored in the variable count\_of\_twos, is printed to the console.

**Use Cases for count()**

**Frequency Analysis**: The count() method is useful for performing frequency analysis on a list to determine how often specific elements occur.

**Element Validation**: You can use the count() method to validate the presence of a specific element in the list before performing any operations on it.

**Duplicates Identification**: The count() method can help identify duplicate elements in a list.

**Handling Non-existent Elements**

If the element you are searching for using count() does not exist in the list, the method will return 0, indicating that the element occurs zero times.

# Create a list with elements

numbers = [1, 2, 3, 2, 4, 2, 5]

# Count the occurrences of number 7 (which doesn't exist in the list)

count\_of\_sevens = numbers.count(7)

# Display the count

print("Occurrences of 7:", count\_of\_sevens)  # Output: Occurrences of 7: 0

**Conclusion**

Congratulations! You now know how to use the count() method in Python lists to count the occurrences of specific elements. The count() method is a simple yet powerful tool for analyzing the contents of lists and identifying the frequency of particular values. With this knowledge, you can confidently use the count() method in your Python projects to perform frequency analysis and validate the presence of elements in lists. Happy coding!

**Extend Method:**

Syntax of extend()

The extend() method is used on a list to add elements from another list (or any iterable) to the end of the current list. The syntax is as follows:

Graphical user interface

Description automatically generated with low confidence

list\_name: The name of the list to which elements will be added.

iterable: An iterable (e.g., list, tuple, string) containing elements to be added to the list.

Example: Using extend()

Let's start by creating two lists and then using the extend() method to add elements from one list to the end of the other.

# Create two lists with elements

fruits = ['apple', 'banana', 'orange']

more\_fruits = ['mango', 'grape']

# Extend the first list with elements from the second list

fruits.extend(more\_fruits)

# Display the extended list

print(fruits)  # Output: ['apple', 'banana', 'orange', 'mango', 'grape']

In this example, we first create two lists, fruits and more\_fruits, each containing different elements. We then use the extend() method on the fruits list to add all elements from the more\_fruits list to the end of fruits. The print() function displays the extended list, which now contains all the elements from both lists.

**Extend with Other Iterables**

The extend() method is not limited to lists; it can also be used with other iterables like tuples and strings.

**Using a Tuple**

fruits = ['apple', 'banana', 'orange']

tuple\_of\_fruits = ('mango', 'grape')

fruits.extend(tuple\_of\_fruits)

print(fruits)  # Output: ['apple', 'banana', 'orange', 'mango', 'grape']

**Using a String**

fruits = ['apple', 'banana', 'orange']

string\_of\_fruits = "mango"

fruits.extend(string\_of\_fruits)

print(fruits)  # Output: ['apple', 'banana', 'orange', 'm', 'a', 'n', 'g', 'o']

In both cases, the extend() method adds the elements from the iterable to the end of the list.

**Combining Multiple Lists**

Using extend(), you can efficiently combine multiple lists into a single list.

list1 = [1, 2, 3]

list2 = [4, 5, 6]

list3 = [7, 8, 9]

combined\_list = []

combined\_list.extend(list1)

combined\_list.extend(list2)

combined\_list.extend(list3)

print(combined\_list)  # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]

**Conclusion**

Congratulations! You now know how to use the extend() method in Python lists to add elements from other lists (or any iterable) to the end of a list. The extend() method is a powerful tool for combining multiple lists and efficiently building large lists without creating new list objects. With this knowledge, you can confidently use the extend() method in your Python projects to manage and manipulate lists effectively. Happy coding!

**Index Method:**

Syntax of index()

The index() method is used on a list to find the index of a specified element. The syntax is as follows:

A picture containing text

Description automatically generated

list\_name: The name of the list in which you want to find the index.

element: The value whose index you want to locate.

index: The variable that stores the index of the first occurrence of the element in the list.

Example: Using index()

Let's start by creating a list and then using the index() method to find the index of a specific element.

# Create a list with elements

fruits = ['apple', 'banana', 'orange', 'mango']

# Find the index of 'orange'

index\_of\_orange = fruits.index('orange')

# Display the index

print("Index of 'orange':", index\_of\_orange)  # Output: Index of 'orange': 2

In this example, we first create a list called fruits with four elements. We then use the index() method to find the index of the element 'orange' in the list. The result, stored in the variable index\_of\_orange, is printed to the console.

**Handling Non-existent Elements**

If the element you are searching for using index() does not exist in the list, it will raise a ValueError. This error indicates that the specified element is not present in the list.

# Create a list with elements

fruits = ['apple', 'banana', 'orange', 'mango']

# Find the index of 'grape' (which doesn't exist in the list)

try:

    index\_of\_grape = fruits.index('grape')

    print("Index of 'grape':", index\_of\_grape)

except ValueError:

    print("Element 'grape' not found in the list.")

A picture containing shape

Description automatically generated

To avoid this error, you can use the in operator to check if the element exists in the list before using the index() method.

# Check if 'grape' exists in the list before finding its index

if 'grape' in fruits:

    index\_of\_grape = fruits.index('grape')

    print("Index of 'grape':", index\_of\_grape)

else:

    print("Element 'grape' not found in the list.")

**Multiple Occurrences of an Element**

The index() method only returns the index of the first occurrence of the specified element. If the element appears multiple times in the list, it will only return the index of the first occurrence.

# Create a list with elements

numbers = [1, 2, 3, 2, 4, 2, 5]

# Find the index of the first occurrence of number 2

index\_of\_2 = numbers.index(2)

print("Index of first occurrence of 2:", index\_of\_2)  # Output: Index of first occurrence of 2: 1

**Conclusion**

Congratulations! You now know how to use the index() method in Python lists to find the index of a specific element. The index() method is a handy tool for locating the position of an element in the list. With this knowledge, you can confidently use the index() method in your Python projects to search and manipulate lists effectively. Happy coding!

**Insert Method:**

Syntax of insert()

The insert() method is used on a list to add an element at a specific index. The syntax is as follows:

Graphical user interface, website

Description automatically generated

list\_name: The name of the list in which you want to insert the element.

index: The position where you want to insert the element. It should be an integer, and the element will be inserted before the element currently at that index.

element: The value to be inserted into the list.

Example: Using insert()

Let's start by creating a list and then using the insert() method to add an element at a specific index.

# Create a list with elements

fruits = ['apple', 'banana', 'orange']

# Insert 'mango' at index 1

fruits.insert(1, 'mango')

# Display the modified list

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

In this example, we first create a list called fruits with three elements. We then use the insert() method to add the element 'mango' at index 1. As a result, the 'mango' is inserted before the element 'banana', and the list is modified accordingly.

**Inserting at the End of the List**

To insert an element at the end of the list, you can use the insert() method with the index equal to the length of the list.

# Create a list with elements

fruits = ['apple', 'banana', 'orange']

# Insert 'mango' at the end of the list

fruits.insert(len(fruits), 'mango')

# Display the modified list

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

In this example, we calculate the index for inserting 'mango' at the end of the list by using len(fruits), which gives the length of the list. The insert() method places 'mango' at the appropriate position, and the list is updated accordingly.

**Negative Indexing**

You can also use negative indexing with the insert() method to insert an element from the end of the list. Negative indices count from the end of the list, with -1 referring to the last element, -2 to the second-last, and so on.

# Create a list with elements

fruits = ['apple', 'banana', 'orange']

# Insert 'mango' at the second-last position

fruits.insert(-1, 'mango')

# Display the modified list

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

In this example, we use -1 as the index to insert 'mango', which places it as the second-last element in the list.

**Conclusion**

Congratulations! You now know how to use the insert() method in Python lists to add elements at specific positions. The insert() method is a powerful tool for inserting new elements into a list and is especially useful when you need to maintain the order of the elements. With this knowledge, you can confidently use the insert() method in your Python projects to manipulate lists effectively. Happy coding!

**Pop Method:**

Syntax of pop()

The pop() method is used on a list to remove and return an element from a specific index. The syntax is as follows:

A picture containing website

Description automatically generated

list\_name: The name of the list from which you want to remove the element.

index (optional): The position from which the element will be removed. If not specified, pop() removes and returns the last element in the list.

removed\_element: The variable that stores the value of the removed element.

Example: Using pop()

Let's start by creating a list and then using the pop() method to remove an element from a specific index.

# Create a list with elements

fruits = ['apple', 'banana', 'orange', 'mango']

# Remove and return the element at index 1

removed\_fruit = fruits.pop(1)

# Display the modified list and the removed element

print("Modified List:", fruits)        # Output: ['apple', 'orange', 'mango']

print("Removed Fruit:", removed\_fruit)  # Output: Removed Fruit: banana

In this example, we first create a list called fruits with four elements. We then use the pop() method with index 1 to remove the element 'banana' from the list. The value of the removed element, 'banana', is stored in the variable removed\_fruit, and the modified list is displayed.

**Removing the Last Element**

If you use the pop() method without providing an index, it will remove and return the last element from the list.

# Create a list with elements

numbers = [1, 2, 3, 4, 5]

# Remove and return the last element

removed\_number = numbers.pop()

# Display the modified list and the removed element

print("Modified List:", numbers)       # Output: [1, 2, 3, 4]

print("Removed Number:", removed\_number)  # Output: Removed Number: 5

In this example, we use pop() without specifying an index, so it removes and returns the last element '5' from the numbers list.

**Handling Invalid Index**

If you provide an invalid index (out of range) to the pop() method, it will raise an IndexError.

# Create a list with elements

numbers = [1, 2, 3, 4, 5]

# Try to remove an element with an invalid index

try:

    removed\_number = numbers.pop(10)

    print("Removed Number:", removed\_number)

except IndexError:

    print("Invalid index. The list does not have an element at index 10.")

Text

Description automatically generated with medium confidence

To avoid this error, you should ensure that the index you provide is within the valid range of the list.

**Conclusion**

Congratulations! You now know how to use the pop() method in Python lists to remove and return elements from specific positions. The pop() method is a powerful tool for manipulating lists by removing elements, and it is especially useful when you need to interact with the removed elements after removal. With this knowledge, you can confidently use the pop() method in your Python projects to manage and modify lists effectively. Happy coding!

**Remove Method:**

Syntax of remove()

The remove() method is used on a list to remove the first occurrence of a specified element. The syntax is as follows:

A picture containing graphical user interface

Description automatically generated

list\_name: The name of the list from which you want to remove the element.

element: The value of the element you want to remove from the list.

Example: Using remove()

Let's start by creating a list and then using the remove() method to remove the first occurrence of a specific element.

# Create a list with elements

fruits = ['apple', 'banana', 'orange', 'banana']

# Remove the first occurrence of 'banana'

fruits.remove('banana')

# Display the modified list

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

In this example, we first create a list called fruits with four elements, including two occurrences of 'banana'. We then use the remove() method to remove the first occurrence of 'banana' from the list. As a result, only the first occurrence is removed, and the modified list is displayed.

**Handling Non-existent Elements**

If the element you are trying to remove using remove() does not exist in the list, it will raise a ValueError. This error indicates that the specified element is not present in the list.

# Create a list with elements

fruits = ['apple', 'banana', 'orange']

# Try to remove 'grape' (which doesn't exist in the list)

try:

    fruits.remove('grape')

except ValueError:

    print("Element 'grape' not found in the list.")

A picture containing text

Description automatically generated

To avoid this error, you can use the in operator to check if the element exists in the list before using the remove() method.

# Check if 'grape' exists in the list before removing it

if 'grape' in fruits:

    fruits.remove('grape')

else:

    print("Element 'grape' not found in the list.")

**Removing All Occurrences**

If you want to remove all occurrences of a specific element from the list, you can use a loop or a list comprehension.

# Create a list with elements

numbers = [1, 2, 3, 2, 4, 2, 5]

# Remove all occurrences of number 2 using a loop

while 2 in numbers:

    numbers.remove(2)

print(numbers)  # Output: [1, 3, 4, 5]

**Conclusion**

Congratulations! You now know how to use the remove() method in Python lists to remove the first occurrence of a specific element. The remove() method is a useful tool for manipulating lists by removing unwanted elements based on their values. With this knowledge, you can confidently use the remove() method in your Python projects to manage and modify lists effectively. Happy coding!

**Reverse Method:**

Syntax of reverse()

The reverse() method is used on a list to reverse the order of its elements. The syntax is as follows:

Shape, background pattern, rectangle

Description automatically generated

list\_name: The name of the list that you want to reverse.

Example: Using reverse()

Let's start by creating a list and then using the reverse() method to reverse the order of its elements.

# Create a list with elements

numbers = [1, 2, 3, 4, 5]

# Reverse the order of elements

numbers.reverse()

# Display the modified list

print(numbers)  # Output: [5, 4, 3, 2, 1]

In this example, we first create a list called numbers with five elements. We then use the reverse() method to reverse the order of the elements in the list. As a result, the list is modified to have its elements arranged from the last to the first.

**Reversing a List in Place**

The reverse() method reverses the list in place, meaning it modifies the original list and does not create a new list with reversed elements.

# Create a list with elements

fruits = ['apple', 'banana', 'orange']

# Reverse the order of elements in place

fruits.reverse()

# Display the modified list

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

**Reversing a Copy of the List**

If you want to reverse a list without modifying the original list, you can create a copy of the list and then use the reverse() method on the copy.

# Create a list with elements

fruits = ['apple', 'banana', 'orange']

# Create a copy of the list and reverse it

reversed\_fruits = fruits.copy()

reversed\_fruits.reverse()

# Display the modified list and the original list

print("Reversed List:", reversed\_fruits)  # Output: ['orange', 'banana', 'apple']

print("Original List:", fruits)           # Output: ['apple', 'banana', 'orange']

In this example, we create a copy of the fruits list using the copy() method and store it in reversed\_fruits. Then we use the reverse() method on the reversed\_fruits list to reverse the order of its elements without affecting the original fruits list.

**Conclusion**

Congratulations! You now know how to use the reverse() method in Python lists to reverse the order of elements in a list. The reverse() method is a powerful tool for rearranging list elements from the last to the first. With this knowledge, you can confidently use the reverse() method in your Python projects to manipulate lists effectively. Happy coding!

**Sort Method:**

Syntax of sort()

The sort() method is used on a list to sort its elements in ascending order. The syntax is as follows:

Shape, background pattern, rectangle

Description automatically generated

To sort the elements in descending order, you can use the reverse parameter:

Graphical user interface

Description automatically generated

list\_name: The name of the list that you want to sort.

reverse (optional): If set to True, the elements will be sorted in descending order. If not provided or set to False, the elements will be sorted in ascending order.

Example: Using sort()

Let's start by creating a list and then using the sort() method to sort its elements.

# Create a list with elements

numbers = [4, 1, 6, 3, 7, 2, 5]

# Sort the elements in ascending order

numbers.sort()

# Display the sorted list

print(numbers)  # Output: [1, 2, 3, 4, 5, 6, 7]

In this example, we first create a list called numbers with seven elements. We then use the sort() method to sort the elements in ascending order. As a result, the list is modified to have its elements arranged in ascending numerical order.

**Sorting in Descending Order**

To sort the elements in descending order, use the reverse=True parameter.

# Create a list with elements

numbers = [4, 1, 6, 3, 7, 2, 5]

# Sort the elements in descending order

numbers.sort(reverse=True)

# Display the sorted list

print(numbers)  # Output: [7, 6, 5, 4, 3, 2, 1]

In this example, we use the sort() method with reverse=True to sort the elements in descending order. The list is modified accordingly, with elements arranged from the highest to the lowest.

**Sorting Lists of Strings**

The sort() method can also be used to sort lists of strings in alphabetical order.

# Create a list with string elements

fruits = ['banana', 'orange', 'apple', 'mango']

# Sort the elements in alphabetical order

fruits.sort()

# Display the sorted list

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

In this example, we create a list called fruits with four string elements. By using the sort() method, we sort the strings in alphabetical order.

**Sorting Lists with Custom Criteria**

For more complex sorting, you can use the key parameter of the sort() method to specify a custom function that determines the sort order.

# Create a list of dictionaries with 'name' and 'age'

people = [

    {'name': 'Alice', 'age': 25},

    {'name': 'Bob', 'age': 20},

    {'name': 'Charlie', 'age': 30}

]

# Sort the list of dictionaries based on 'age'

people.sort(key=lambda x: x['age'])

# Display the sorted list

print(people)

Graphical user interface, text, application

Description automatically generated

In this example, we have a list of dictionaries representing people with their names and ages. We use the sort() method with a custom key function, which sorts the list based on the 'age' key of each dictionary.

**Conclusion**

Congratulations! You now know how to use the sort() method in Python lists to sort elements in ascending or descending order. The sort() method is a powerful tool for organizing list elements numerically, alphabetically, or based on custom criteria. With this knowledge, you can confidently use the sort() method in your Python projects to manage and manipulate lists effectively. Happy coding!

**Section 18.0:**

**Slice list:**

Basic Slicing Syntax

The syntax for slicing a list in Python is as follows:

Text

Description automatically generated

list\_name: The name of the list you want to slice.

start\_index: The index at which the slice starts (inclusive).

end\_index: The index at which the slice ends (exclusive). The element at this index will not be included in the slice.

Example: Basic Slicing

Let's start with a simple example of slicing a list:

# Create a list with elements

numbers = [1, 2, 3, 4, 5]

# Slice the list to get a subset of elements

subset = numbers[1:4]

# Display the sliced list

print(subset)  # Output: [2, 3, 4]

In this example, we have a list called numbers with five elements. We use slicing to create a new list called subset containing elements from index 1 to index 3 (excluding the element at index 4). The sliced list subset contains elements [2, 3, 4].

**Omitting Start and End Indices**

If you omit the start\_index, the slice will start from the beginning of the list. If you omit the end\_index, the slice will go until the end of the list.

# Create a list with elements

numbers = [1, 2, 3, 4, 5]

# Slice from the beginning to index 2 (exclusive)

subset1 = numbers[:3]

# Slice from index 2 to the end

subset2 = numbers[2:]

# Display the sliced lists

print(subset1)  # Output: [1, 2, 3]

print(subset2)  # Output: [3, 4, 5]

In the above example, subset1 starts from the beginning of the list and goes up to index 2 (exclusive), while subset2 starts from index 2 and goes until the end of the list.

**Negative Indices**

You can also use negative indices for slicing, where -1 refers to the last element, -2 to the second-to-last, and so on.

# Create a list with elements

numbers = [1, 2, 3, 4, 5]

# Slice the last three elements using negative indices

subset = numbers[-3:]

# Display the sliced list

print(subset)  # Output: [3, 4, 5]

In this example, subset contains the last three elements of the numbers list, as we use a negative index -3 to start the slice from the third-to-last element.

**Specifying a Step Size**

You can also specify a step size to include only every nth element in the slice.

# Create a list with elements

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Slice to get every second element

subset = numbers[::2]

# Display the sliced list

print(subset)  # Output: [1, 3, 5, 7, 9]

In this example, the subset contains every second element from the numbers list, as we use a step size of 2 (::2) to skip one element after each inclusion.

**Negative Step Size**

Using a negative step size allows you to reverse the order of the slice.

# Create a list with elements

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Slice to get every second element in reverse order

subset = numbers[::-2]

# Display the sliced list

print(subset)  # Output: [10, 8, 6, 4, 2]

In this example, subset contains every second element from the numbers list, but in reverse order, as we use a negative step size of -2 ([::-2]).

**Modifying Lists with Slicing**

You can also use slicing to modify elements in a list:

# Create a list with elements

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Modify the list, replacing elements from index 3 to index 6 with new elements

numbers[3:7] = [10, 11, 12]

# Display the modified list

print(numbers)  # Output: [1, 2, 3, 10, 11, 12, 7, 8, 9]

In this example, we modify the list numbers by replacing elements from index 3 to index 6 with new elements [10, 11, 12].

**Conclusion**

Congratulations! You now know how to use slicing to extract, skip, and modify elements in Python lists. Slicing is a powerful and versatile tool that enables you to manipulate lists efficiently. With this knowledge, you can confidently use slicing in your Python projects to work with lists effectively. Happy coding!

**Section 19.0:**

**Delete (del) statement:**

Syntax of the del Statement

The del statement is used to delete a variable or an element from a list. The syntax is as follows:

Graphical user interface, text

Description automatically generated

variable\_name: The name of the variable you want to delete.

list\_name: The name of the list from which you want to delete an element.

index: The position of the element you want to delete from the list. It should be an integer representing the index of the element you wish to remove.

Example: Using the del Statement with Variables

Let's start by using the del statement to delete a variable:

# Define a variable

x = 10

# Delete the variable 'x'

del x

# Try to access 'x' after deletion

# This will raise a NameError since 'x' no longer exists

print(x)

In this example, we define a variable x and then use the del statement to delete it. After deletion, attempting to access the variable x will result in a NameError because the variable no longer exists.

Example: Using the del Statement with Lists

Next, let's use the del statement to remove an element from a list:

# Create a list with elements

fruits = ['apple', 'banana', 'orange', 'mango']

# Delete the element at index 1 ('banana')

del fruits[1]

# Display the modified list

print(fruits)  # Output: ['apple', 'orange', 'mango']

In this example, we have a list called fruits. Using the del statement with the index 1, we remove the element 'banana' from the list, resulting in the list ['apple', 'orange', 'mango'].

**Conclusion**

Congratulations! You now know how to use the del statement in Python to delete variables or elements from a list. The del statement is a valuable tool for managing memory and removing unwanted data from your program. With this knowledge, you can confidently use the del statement in your Python projects to efficiently manage resources and manipulate data as needed. Happy coding!

**Section 20.0:**

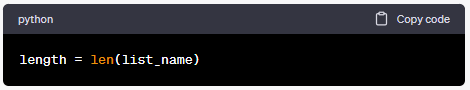
**Length of list:**

**Video Resource:**

<https://www.google.com/search?q=python+length+of+list+video&rlz=1C1GCEA_enUS953US953&ei=4WnGZLvYGLfN0PEP8ZeboAg&ved=0ahUKEwj7zO7oyLaAAxW3JjQIHfHLBoQQ4dUDCBA&uact=5&oq=python+length+of+list+video&gs_lp=Egxnd3Mtd2l6LXNlcnAiG3B5dGhvbiBsZW5ndGggb2YgbGlzdCB2aWRlbzIIECEYoAEYwwQyCBAhGKABGMMESNsnUPEFWOUkcAF4AZABAJgBkAKgAfcTqgEFMC45LjW4AQPIAQD4AQHCAgoQABhHGNYEGLADwgIGEAAYBxgewgIIEAAYCBgHGB7CAggQABiKBRiGA8ICBhAAGAgYHuIDBBgAIEGIBgGQBgg&sclient=gws-wiz-serp#kpvalbx=_a2rGZO6LBMng0PEP_Z2bWA_28>

**Method 1: Using len() Function**

The most common and straightforward way to get the length of a list in Python is by using the built-in len() function. The syntax is as follows:



list\_name: The name of the list for which you want to get the length.

length: The variable that will store the length of the list.

Example:

# Create a list with elements

fruits = ['apple', 'banana', 'orange', 'mango']

# Get the length of the list

length = len(fruits)

# Display the length

print(length)  # Output: 4

In this example, we have a list called fruits. We use the len() function to obtain the length of the list, which is 4 since it contains four elements.

**Method 2: Using a Loop**

While using the len() function is the preferred and efficient method, you can also find the length of a list using a loop. Here's an example:

# Create a list with elements

numbers = [1, 2, 3, 4, 5]

# Initialize a variable to count the elements

length = 0

# Loop through the list and increment the count for each element

for \_ in numbers:

    length += 1

# Display the length

print(length)  # Output: 5

In this example, we have a list called numbers. We use a for loop to iterate through the list and increment the length variable for each element in the list, ultimately finding its length.

**Conclusion**

Congratulations! You now know how to get the length of a list in Python using the len() function or a loop. The len() function is the standard and efficient way to determine the number of elements in a list. With this knowledge, you can confidently use the length of a list in your Python projects to perform various operations based on the number of elements. Happy coding!

**Section 20.1**

**Create a copy of a list:**

**Method 1: Slicing**

One of the simplest and most common ways to copy a list in Python is by using slicing. Slicing allows you to create a new list containing all elements from the original list. The syntax is as follows:

Shape, rectangle

Description automatically generated

my\_list: The name of the original list that you want to copy.

new\_list: The new list that will be an exact copy of the original list.

Example:

# Create a list with elements

fruits = ['apple', 'banana', 'orange', 'mango']

# Copy the list using slicing

new\_fruits = fruits[:]

# Display the original and copied lists

print("Original List:", fruits)

print("Copied List:", new\_fruits)

In this example, we have a list called fruits. We use slicing [:] to create a new list called new\_fruits, which is an exact copy of the original list.

**Method 2: Using list() Function**

You can also use the list() function to create a copy of the list. The list() function takes an iterable (like a list) as an argument and returns a new list containing all the elements of the iterable.

A picture containing rectangle

Description automatically generated

Example:

# Create a list with elements

numbers = [1, 2, 3, 4, 5]

# Copy the list using the list() function

new\_numbers = list(numbers)

# Display the original and copied lists

print("Original List:", numbers)

print("Copied List:", new\_numbers)

In this example, we have a list called numbers. We use the list() function to create a new list called new\_numbers, which is an exact copy of the original list.

**Method 3: Using copy() Method**

For lists, you can use the copy() method to create a shallow copy of the list. The copy() method returns a new list containing all the elements of the original list.

A picture containing graphical user interface

Description automatically generated

# Create a list with elements

colors = ['red', 'green', 'blue']

# Copy the list using the copy() method

new\_colors = colors.copy()

# Display the original and copied lists

print("Original List:", colors)

print("Copied List:", new\_colors)

In this example, we have a list called colors. We use the copy() method to create a new list called new\_colors, which is an exact copy of the original list.

Conclusion

Congratulations! You now know different methods to copy a list in Python. Creating a copy of a list is essential when you want to manipulate one list independently of the other. Whether you use slicing, the list() function, or the copy() method, you can confidently create copies of lists in your Python projects. Happy coding!

**Section 20.2:**

**List Examples:**

#Create a list of colors

colors = ["Red","Yellow","Blue","Green","Yellow"]

#determine the length of the list of colors

length= len(colors)

#print the item at index 2 in the color list

item  = colors[2]

print (item)

#change the the 3rd item in to list from Blue to white

colors[2] = "White"

print (colors)

#Example 2

fabrics = []

#add items to the fabrics list

fabrics.append("velvet")

fabrics.append("denim")

fabrics.append("gingham")

#insert an at the beggining of the fabric list

fabrics.insert(0,"chiffon")

print (fabrics)

#determine if gingham is in the list of fabrics

if "gingham" in fabrics:

    print("gingham is in the list")

else:

    print ("gingham is not in the list")

#get the index where velvet is stored in the fabrics list

print (fabrics.index("velvet"))

#remove denim from the fabrics list in print it.

x = fabrics.index("denim")  #fabircs.remove("denim")  \*\*\*Would also work

fabrics.pop(x)

print (fabrics)

#Example 3:

sum = 0

count=0

#Get ten or fewer numbers from the user and add them together

while count <6:

    i = int(input("Choose a number:"))

    sum += i

    count +=1

#print the sum of the numbers for the user to see.

print (sum)

Vlookup list

animal\_list = ["bear","cat","dog","elephant"]

farm\_animal\_list = ["donkey","cow","chicken","cat","dog"]

#Store a list to hold values for matching and unmatching

matching\_list = []

unmacthing\_list = []

#common items in the two list (vlookup)

for animal in animal\_list:

    if animal in farm\_animal\_list:

        print (animal)

#append list of all common animals

for animal in animal\_list:

    if animal in farm\_animal\_list:

        matching\_list.append(animal)

print(matching\_list)

#compare unmatched list A compared to list B (print a line for each)

for animal in animal\_list:

    if animal not in farm\_animal\_list:

        print (f"this animal is not in the farm animal list: {animal}")

#compare unmatched list A compared to list B (store in a new list)

for animal in animal\_list:

    if animal not in farm\_animal\_list:

        unmacthing\_list.append(animal)

print(unmacthing\_list)

#create a list of only unique values in one list

unique\_list = []

list\_one = [1,2,2,3,3,4,5,5,6,6,7,7,7,8,8,8,9,9,9]

for i in list\_one:

    if i not in unique\_list:

        unique\_list.append(i)

print(unique\_list)

#create a list of duplicates in the single list

list\_two=[1,2,3,4,4,5,5,6,1]

dup\_list = set([x for x in list\_two if list\_two.count(x)>1])

print(dup\_list)

#concatenate two list

combined\_list = []

for i in animal\_list:

    combined\_list.append(i)

for i in farm\_animal\_list:

    combined\_list.append(i)

print (combined\_list)

unique\_animals = set(combined\_list) #set makes the list unquie

print(unique\_animals)

print (sorted(unique\_animals)) #prints list sorted without modifiying the list

unique\_animals.sort()

print(unique\_animals)

**Undup list**

Method: Using List Comprehension

One of the simplest and efficient ways to find the unduplicated elements between two lists is by using list comprehension. List comprehension allows you to create a new list by iterating through one list and filtering elements based on their presence in the other list.

unduplicated\_list = [element for element in list1 if element not in list2]

list1: The first list, from which unduplicated elements will be extracted.

list2: The second list, used to check for duplicate elements.

unduplicated\_list: The new list that will contain elements that are unique to list1.

Example:

# Create two lists with elements

list1 = [1, 2, 3, 4, 5]

list2 = [4, 5, 6, 7, 8]

# Find the unduplicated elements between the two lists

unduplicated\_list = [element for element in list1 if element not in list2]

# Display the unduplicated elements

print(unduplicated\_list)  # Output: [1, 2, 3, 6, 7, 8]

In this example, we have two lists, list1 and list2, with some duplicate elements. By using list comprehension, we create a new list called unduplicated\_list, which contains elements that are unique to list1 and not present in list2.

Method: Using Set Operations

Another way to achieve unduplication between two lists is by converting the lists to sets and using set operations like difference().

set1 = set(list1)

set2 = set(list2)

unduplicated\_list = list(set1.difference(set2))

list1: The first list, from which unduplicated elements will be extracted.

list2: The second list, used to check for duplicate elements.

set1, set2: Sets created from list1 and list2.

unduplicated\_list: The new list that will contain elements that are unique to list1.

Example:

list1 = [1, 2, 3, 4, 5]

list2 = [4, 5, 6, 7, 8]

# Find the unduplicated elements between the two lists using sets

set1 = set(list1)

set2 = set(list2)

unduplicated\_list = list(set1.difference(set2))

# Display the unduplicated elements

print(unduplicated\_list)  # Output: [1, 2, 3]

In this example, we create sets set1 and set2 from list1 and list2, respectively. We then use the difference() method to find elements that are unique to set1, which represents the unduplicated elements between the two lists.

**Conclusion**

Congratulations! You now know how to achieve unduplication between two lists in Python. Whether you use list comprehension or set operations, you can confidently find elements unique to both lists and create a new list with those unduplicated elements. With this knowledge, you can effectively handle scenarios where you need to compare and process data from two lists. Happy coding!

**Section 21.0**

**Packing and unpacking list**

**Video Resource:**<https://www.youtube.com/watch?v=ElK1sCg79zo>

**Packing Lists**

Method 1: Creating Lists

In Python, you can create a list by enclosing multiple values within square brackets [ ].

Text

Description automatically generated

Example:

# Packing integers into a list

numbers = [1, 2, 3, 4, 5]

# Packing strings into a list

fruits = ['apple', 'banana', 'orange'

]

Method 2: Using list() Function

You can also use the list() function to pack values into a list.

Text

Description automatically generated with medium confidence

# Pack a string into a list using the list() function

characters = list("hello")

Unpacking Lists

Method 1: Assigning to Variables

In Python, you can assign list elements to individual variables using unpacking.

Text

Description automatically generated

Example:

# Unpacking integers into separate variables

a, b, c = [1, 2, 3]

# Unpacking strings into separate variables

x, y, z = ['apple', 'banana', 'orange']

Method 2: Using \* Operator

The \* operator (asterisk) allows you to capture multiple elements as a list during unpacking. It is particularly useful when the number of elements is unknown.

Text

Description automatically generated

Example:

# Using \* operator to capture multiple elements during unpacking

first, \*rest, last = [1, 2, 3, 4, 5]

**Conclusion**

Congratulations! You now know how to pack and unpack lists in Python. Packing allows you to combine multiple values into a single list, while unpacking enables you to extract elements from a list into separate variables. These techniques are valuable when working with data structures and handling multiple values efficiently. With this knowledge, you can confidently use packing and unpacking in your Python projects to manipulate lists and variables effectively. Happy coding!

Example 1:

Dictionary:

sales\_staff = {

"Weylin":278,

"Cocoa":279,

"Cheaters":170

}

product\_list ={

278:["radio","TV","Fishtank"],

279:["plush toy","treats","smartee"],

170:["drugs","weapons"]

}

print(sales\_staff)

print(sales\_staff.keys())

print(sales\_staff.values())

print(sales\_staff.items())

Extra Material:

**Section 10 Python coding examples:**

**Section 10.1**

**Final project example 1:**

import csv

#import pandas as pd

#indexes

num=0

name= 1

DOB= 2

SSN=3

MA\_Num=4

approval\_date=5

billing\_date=6

billing\_code=7

qty=8

approved\_amt=9

billed\_amt=10

red\_font = '\u001b[31;22m'

green\_font = '\u001b[32;22m'

yellow\_font = '\u001b[33;22m'

reset\_font = '\u001b[39;22m'

def main():

    first\_file = first\_month\_file("Final\_Project.csv")

    #print\_list(first\_file)

    billed\_amount(first\_file) #Print billed amount greater than approved amount

    #running\_total\_person(first\_file)

    check\_MA\_num(first\_file)

    check\_SSN(first\_file)

    check\_billing\_amount(first\_file)

    check\_billing\_code(first\_file)

def first\_month\_file(file\_name):

    first\_month\_file = []

    with open (file\_name, "rt") as csv\_file:

        #reads the file

        reader = csv.reader(csv\_file)

        #skip header

        next(reader)

        #loop through the read to get all lines

        for row in reader:

            first\_month\_file.append(row)

    return first\_month\_file

def print\_list(first\_month\_file):

    for line in first\_month\_file:

        print(line)

def billed\_amount(first\_month\_file):

    '''

    function to test if the provider billed more than the approved

    amount.  Returns Name, date, approved amount, billed amount and difference

    '''

    for line in first\_month\_file:

        check\_num = line[num]

        person\_name = line[name]

        quanity = line[qty]

        billing\_date\_violation = line[billing\_date]

        approved\_amount = float(line[approved\_amt])

        billed\_amount = float(line[billed\_amt])\*int(quanity)

        difference = billed\_amount - approved\_amount

        if billed\_amount > approved\_amount:

            print (f"{red\_font}Error!{yellow\_font} line: {check\_num}.{reset\_font} Over budget: {person\_name}, {billing\_date\_violation} approved\_amt: ${approved\_amount:.2f}; billed: ${billed\_amount:.2f} you went over ${difference:.2f}")

def running\_total\_person(first\_month\_file):

    '''

    Prints the running total for each person in the list for the given month

    Not working as designed yet!!!

    '''

    df = pd.read\_csv('work\_test\_oct')

    df[billed\_amount].sum()

def check\_MA\_num(first\_month\_file):

    for line in first\_month\_file:

        check\_num = line[num]

        MA\_num\_check = line[MA\_Num]

        if len(MA\_num\_check) > 10:

            print (f'{red\_font}Error!{yellow\_font} line: {check\_num}. {reset\_font}{MA\_num\_check} has too many digits')

        elif len(MA\_num\_check) < 10:

            print (f'{red\_font}Error!{yellow\_font} line: {check\_num}. {reset\_font}{MA\_num\_check} does not have enough digits')

        if MA\_num\_check.isdigit() ==False:

            print (f"{red\_font}Error!{yellow\_font} line: {check\_num}. {reset\_font}{MA\_num\_check} must only use numbers")

def check\_SSN(first\_month\_file):

    for line in first\_month\_file:

        check\_num = line[num]

        SSN\_check = str(line[SSN])

        segments=SSN\_check.split("-")

        if len(segments)==3:

            if len(segments[0])!=3 or len(segments[1])!=2 or len(segments[2])!=4:

                print (f"{red\_font}Error!{yellow\_font} line: {check\_num}. {reset\_font}{SSN\_check} incorrect")

            if (segments[0]).isdigit()==False or (segments[1]).isdigit()==False or (segments[2]).isdigit()==False:

                print(f"{red\_font}Error!{yellow\_font} line: {check\_num}. {reset\_font}{SSN\_check} must only be numbers")

        else:

            print(f"{red\_font}Error!{yellow\_font} line: {check\_num}. {reset\_font}{SSN\_check} segment issue")

def check\_billing\_amount(first\_month\_file):

    ''''

    This checks to ensure the billing amount does not exceed $10,000.

    '''

    for line in first\_month\_file:

        check\_num = line[num]

        billing\_amount\_check = line[billed\_amt]

        quanity = line[qty]

        total\_billing = float(billing\_amount\_check) \* int(quanity)

        if total\_billing > 10000:

            print(f"{red\_font}Error!{yellow\_font} line: {check\_num}. {reset\_font}You cannot bill more than $10,000")

def check\_billing\_code(first\_month\_file):

    '''

    this checks to esnure the billing code is on the approved list

    '''

    for line in first\_month\_file:

        check\_num = line[num]

        check\_billing\_code = line[billing\_code]

        if check\_billing\_code not in ["S5165","S9986","T1013","T1016","T1019","T1999","T2028","T2038","A0100",\

            "A0120","T2002","T2004","A0090","S0215","A0100","A0110","A0130","S0209"]:

            print (f"{red\_font}Error!{yellow\_font} line: {check\_num}. {reset\_font}Contains an improper billing code {check\_billing\_code}!")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Example 2**

**Tic-Tac-Toe Game:**

board = [["1","2","3"],["4","5","6"],["7","8","9"]]

#Font Colors

black\_font = '\u001b[30;22m'

red\_font = '\u001b[31;22m'

green\_font = '\u001b[32;22m'

yellow\_font = '\u001b[33;22m'

blue\_font = '\u001b[34;22m'

purple\_font = '\u001b[35;22m'

cyan\_font = '\u001b[36;22m'

white\_font = '\u001b[37;22m'

reset\_font = '\u001b[39;22m'

def main():

    while True:

        player\_moves()

def print\_board():

    slot1=board[0][0]

    slot2=board[0][1]

    slot3=board[0][2]

    slot4=board[1][0]

    slot5=board[1][1]

    slot6=board[1][2]

    slot7=board[2][0]

    slot8=board[2][1]

    slot9=board[2][2]

    print("\tWelcome to the game of")

    print("\tTic\tTac\tToe")

    print()

    print(f"\t{slot1}\t{slot2}\t{slot3}")

    print(f"\t{slot4}\t{slot5}\t{slot6}")

    print(f"\t{slot7}\t{slot8}\t{slot9}")

    print()

def player\_moves():

    print\_board()

    player\_choice = input(f"{cyan\_font}Player 1(X) Enter a number 1-9: {reset\_font}")

    if player\_choice == "1":

        board[0][0] = "X"

        check\_victory()

        check\_draw()

    elif player\_choice == "2":

        board[0][1] = "X"

        check\_victory()

        check\_draw()

    elif player\_choice == "3":

        board[0][2] = "X"

        check\_victory()

        check\_draw()

    elif player\_choice == "4":

        board[1][0] = "X"

        check\_victory()

        check\_draw()

    elif player\_choice == "5":

        board[1][1] = "X"

        check\_victory()

        check\_draw()

    elif player\_choice == "6":

        board[1][2] = "X"

        check\_victory()

        check\_draw()

    elif player\_choice == "7":

        board[2][0] = "X"

        check\_victory()

        check\_draw()

    elif player\_choice == "8":

        board[2][1] = "X"

        check\_victory()

        check\_draw()

    elif player\_choice == "9":

        board[2][2] = "X"

        check\_victory()

        check\_draw()

    #code for player 2 moves

    print("Player 2 turn")

    print\_board()

    player\_choice = input(f"{green\_font}Player 2(O): Enter a number 1-9: {reset\_font}")

    if player\_choice == "1":

        board[0][0] = "O"

        check\_victory()

        check\_draw()

    elif player\_choice == "2":

        board[0][1] = "O"

        check\_victory()

        check\_draw()

    elif player\_choice == "3":

        board[0][2] = "O"

        check\_victory()

        check\_draw()

    elif player\_choice == "4":

        board[1][0] = "O"

        check\_victory()

        check\_draw()

    elif player\_choice == "5":

        board[1][1] = "O"

        check\_victory()

        check\_draw()

    elif player\_choice == "6":

        board[1][2] = "O"

        check\_victory()

        check\_draw()

    elif player\_choice == "7":

        board[2][0] = "O"

        check\_victory()

        check\_draw()

    elif player\_choice == "8":

        board[2][1] = "O"

        check\_victory()

        check\_draw()

    elif player\_choice == "9":

        board[2][2] = "O"

        check\_victory()

        check\_draw()

def check\_victory():

    if board[0][0]=="X" and board[0][1]=="X" and board[0][2]=="X": #top vert line win

        print (f"{red\_font}Player 1 wins. Congrats!{reset\_font}")

    elif board[1][0]=="X" and board[1][1]=="X" and board[1][2]=="X": #mid vert line win

        print (f"{red\_font}Player 1 wins. Congrats!{reset\_font}")

    elif board[2][0]=="X" and board[2][1]=="X" and board[2][2]=="X": #bottom vert line win

        print (f"{red\_font}Player 1 wins. Congrats!{reset\_font}")

    elif board[0][0]=="X" and board[1][0]=="X" and board[2][0]=="X": #left hort line win

        print (f"{red\_font}Player 1 wins. Congrats!{reset\_font}")

    elif board[0][1]=="X" and board[1][1]=="X" and board[2][1]=="X": #middle hort line win

        print (f"{red\_font}Player 1 wins. Congrats!{reset\_font}")

    elif board[0][2]=="X" and board[1][2]=="X" and board[2][2]=="X": #right hort line win

        print (f"{red\_font}Player 1 wins. Congrats!{reset\_font}")

    elif board[0][0]=="X" and board[1][1]=="X" and board[2][2]=="X": #left to right dia win

        print (f"{red\_font}Player 1 wins. Congrats!{reset\_font}")

    elif board[0][2]=="X" and board[1][1]=="X" and board[2][0]=="X": #right to left dia win

        print (f"{red\_font}Player 1 wins. Congrats!{reset\_font}")

    elif board[0][0]=="O" and board[0][1]=="O" and board[0][2]=="O": #top vert line win

        print (f"{red\_font}Player 2 wins. Congrats!{reset\_font}")

    elif board[1][0]=="O" and board[1][1]=="O" and board[1][2]=="O": #mid vert line win

        print (f"{red\_font}Player 2 wins. Congrats!{reset\_font}")

    elif board[2][0]=="O" and board[2][1]=="O" and board[2][2]=="O": #bottom vert line win

        print (f"{red\_font}Player 2 wins. Congrats!{reset\_font}")

    elif board[0][0]=="O" and board[1][0]=="O" and board[2][0]=="O": #left hort line win

        print (f"{red\_font}Player 2 wins. Congrats!{reset\_font}")

    elif board[0][1]=="O" and board[1][1]=="O" and board[2][1]=="O": #middle hort line win

        print (f"{red\_font}Player 2 wins. Congrats!{reset\_font}")

    elif board[0][2]=="O" and board[1][2]=="O" and board[2][2]=="O": #right hort line win

        print (f"{red\_font}Player 2 wins. Congrats!{reset\_font}")

    elif board[0][0]=="O" and board[1][1]=="O" and board[2][2]=="O": #left to right dia win

        print (f"{red\_font}Player 2 wins. Congrats!{reset\_font}")

    elif board[0][2]=="O" and board[1][1]=="O" and board[2][0]=="O": #right to left dia win

        print (f"{red\_font}Player 2 wins. Congrats!{reset\_font}")

def check\_draw():

    if ((board[0][0] =="X" or board[0][0] =="O") and (board[0][1] =="X" or board[0][1] =="O") and (board[0][2] =="X" or board[0][2] =="O")\

    and (board[1][0] =="X" or board[1][0] =="O") and (board[1][1] =="X" or board[1][1] =="O") and (board[1][2] =="X" or board[1][2] =="O")\

    and (board[2][0] =="X" or board[2][0] =="O") and (board[2][1] =="X" or board[2][1] =="O") and (board[2][2] =="X" or board[2][2] =="O")):

        print (f"{yellow\_font}Draw.  Good Game!{reset\_font}")

if \_\_name\_\_=="\_\_main\_\_":

    main()

**Example 3:**

**Tic-Tac\_Toe BYU Solution:**

'''

Tic-Tac-Toe: A Solution

Author: Bro. Manley

'''

def main():

    player = next\_player("")

    board = create\_board()

    while not (has\_winner(board) or is\_a\_draw(board)):

        display\_board(board)

        make\_move(player, board)

        player = next\_player(player)

    display\_board(board)

    print("Good game. Thanks for playing!")

def create\_board():

    board = []

    for square in range(9):

        board.append(square + 1)

    return board

def display\_board(board):

    print()

    print(f"{board[0]}|{board[1]}|{board[2]}")

    print('-+-+-')

    print(f"{board[3]}|{board[4]}|{board[5]}")

    print('-+-+-')

    print(f"{board[6]}|{board[7]}|{board[8]}")

    print()

def is\_a\_draw(board):

    for square in range(9):

        if board[square] != "x" and board[square] != "o":

            return False

    return True

def has\_winner(board):

    return (board[0] == board[1] == board[2] or

            board[3] == board[4] == board[5] or

            board[6] == board[7] == board[8] or

            board[0] == board[3] == board[6] or

            board[1] == board[4] == board[7] or

            board[2] == board[5] == board[8] or

            board[0] == board[4] == board[8] or

            board[2] == board[4] == board[6])

def make\_move(player, board):

    square = int(input(f"{player}'s turn to choose a square (1-9): "))

    board[square - 1] = player

def next\_player(current):

    if current == "" or current == "o":

        return "x"

    elif current == "x":

        return "o"

if \_\_name\_\_ == "\_\_main\_\_":

    main()

Creating Classes:

The Main file calls to the Director Class twice (D1 and D2):

Table

Description automatically generated with low confidence

The Director class (D1) class creates 4 new parameters through the use of a constructor :

Text

Description automatically generated

The function D2 is called for the Main.py:

Graphical user interface, text, application

Description automatically generated

D2’s job has 3 jobs to do called: 1,2,3.

Graphical user interface, application

Description automatically generated

Job 1 is a function:

Graphical user interface, text, application

Description automatically generated

Job 2 is a function:

Graphical user interface, text, application

Description automatically generated

Job 3 is a function:

Graphical user interface, text, application, email

Description automatically generated

Functions 2 (do\_updates is connected to the die.py

Timeline

Description automatically generated with medium confidence

Graphical user interface, table

Description automatically generated

And B is connected:

Graphical user interface

Description automatically generated with medium confidence

Table

Description automatically generated with low confidence

Class example:

class MIChoice\_employee:

    residence = "Michigan"

    supervisor = "Elizabeth"

    work\_location = "Home"

    def \_\_init\_\_(self,name,gender,employee\_num,software\_skill):

        self.name = name

        self.gender = gender,

        self.employee\_num = employee\_num

        self.software\_skill = software\_skill

    def skill\_type(self):

        if self.software\_skill == "Access":

            print (f"{self.name} is a query writer!")

        elif self.software\_skill =="SQL":

            print (f"{self.name} is a query writer!")

        elif self.software\_skill =="BIQ":

            print (f"{self.name} is a query writer!")

        else:

            print (f"{self.name} is data entry")

employee\_1 = MIChoice\_employee("Amanda","F","1076541","Access")

employee\_2 = MIChoice\_employee("Kevin","M","1071235","BIQ")

employee\_3 = MIChoice\_employee("Dorothy","F","1077895","Bridges")

employee\_4 = MIChoice\_employee("Angela","F","1075654","Excel")

print(employee\_3.skill\_type())