String indexing allows you to access individual characters in a string. You can do this by using square brackets and the location, or index, of the character you want to access. It's important to remember that Python starts indexes at 0. So to access the first character in a string, you would use the index [0]. If you try to access an index that's larger than the length of your string, you'll get an **IndexError**. This is because you're trying to access something that doesn't exist! You can also access indexes from the end of the string going towards the start of the string by using negative values. The index [-1] would access the last character of the string, and the index [-2] would access the second-to-last character.

You can also access a portion of a string, called a slice or a substring. This allows you to access multiple characters of a string. You can do this by creating a range, using a colon as a separator between the start and end of the range, like [2:5]. This range is similar to the range() function we saw previously. It includes the first number, but goes to one less than the last number. You can also easily reference the start or end of the string by leaving one value blank. For example [:5] would give us all characters from the start to the fourth character in the string. We can also use [5:] to get the characters from the fourth character to the end of the string.

In Python, strings are immutable. This means that they can't be modified. So if we wanted to fix a typo in a string, we can't simply modify the wrong character. We would have to create a new string with the typo corrected. We can also assign a new value to the variable holding our string.

If we aren't sure what the index of our typo is, we can use the string method *index* to locate it and return the index. Let's imagine we have the string **"lions tigers and bears"** in the variable **animals**. We can locate the index that contains the letter **g** using *animals.index("g")*, which will return the index; in this case 8. We can also use substrings to locate the index where the substring begins. *animals.index("bears")* would return 17, since that's the start of the substring. If there's more than one match for a substring, the index method will return the first match. If we try to locate a substring that doesn't exist in the string, we'll receive a **ValueError** explaining that the substring was not found.

We can avoid a ValueError by first checking if the substring exists in the string. This can be done using the ***in*** keyword. We saw this keyword earlier when we covered *for* loops. In this case, it's a conditional that will be either True or False. If the substring is found in the string, it will be True. If the substring is not found in the string, it will be False. Using our previous variable **animals**, we can do **"horses" in animals** to check if the substring "horses" is found in our variable. In this case, it would evaluate to False, since horses aren't included in our example string. If we did **"tigers" in animals**, we'd get True, since this substring is contained in our string.

We've covered a bunch of String class methods already, so let's keep building on those and run down some more advanced methods.

The string method **lower** will return the string with all characters changed to lowercase. The inverse of this is the **upper** method, which will return the string all in uppercase. Just like with previous methods, we call these on a string using dot notation, like **"this is a string".upper()**. This would return the string **"THIS IS A STRING"**. This can be super handy when checking user input, since someone might type in all lowercase, all uppercase, or even a mixture of cases.

You can use the **strip** method to remove surrounding whitespace from a string. Whitespace includes spaces, tabs, and newline characters. You can also use the methods **lstrip** and **rstrip** to remove whitespace only from the left or the right side of the string, respectively.

The method **count** can be used to return the number of times a substring appears in a string. This can be handy for finding out how many characters appear in a string, or counting the number of times a certain word appears in a sentence or paragraph.

If you wanted to check if a string ends with a given substring, you can use the method **endswith**. This will return True if the substring is found at the end of the string, and False if not.

The **isnumeric** method can check if a string is composed of only numbers. If the string contains only numbers, this method will return True. We can use this to check if a string contains numbers before passing the string to the **int()** function to convert it to an integer, avoiding an error. Useful!

We took a look at string concatenation using the plus sign, earlier. We can also use the **join** method to concatenate strings. This method is called on a string that will be used to join a list of strings. The method takes a list of strings to be joined as a parameter, and returns a new string composed of each of the strings from our list joined using the initial string. For example, **" ".join(["This","is","a","sentence"])** would return the string **"This is a sentence"**.

The inverse of the join method is the **split** method. This allows us to split a string into a list of strings. By default, it splits by any whitespace characters. You can also split by any other characters by passing a parameter.

You can use the **format** method on strings to concatenate and format strings in all kinds of powerful ways. To do this, create a string containing curly brackets, **{}**, as a placeholder, to be replaced. Then call the format method on the string using *.format()* and pass variables as parameters. The variables passed to the method will then be used to replace the curly bracket placeholders. This method automatically handles any conversion between data types for us.

If the curly brackets are empty, they'll be populated with the variables passed in the order in which they're passed. However, you can put certain expressions inside the curly brackets to do even more powerful string formatting operations. You can put the name of a variable into the curly brackets, then use the names in the parameters. This allows for more easily readable code, and for more flexibility with the order of variables.

You can also put a formatting expression inside the curly brackets, which lets you alter the way the string is formatted. For example, the formatting expression **{:.2f}** means that you'd format this as a float number, with two digits after the decimal dot. The colon acts as a separator from the field name, if you had specified one. You can also specify text alignment using the greater than operator: **>**. For example, the expression **{:>3.2f}** would align the text three spaces to the right, as well as specify a float number with two decimal places. String formatting can be very handy for outputting easy-to-read textual output.

String Reference Cheat Sheet

In Python, there are a lot of things you can do with strings. In this cheat sheet, you'll find the most common string operations and string methods.

String operations

- len(string) Returns the length of the string
- for character in string Iterates over each character in the string
- if substring in string Checks whether the substring is part of the string
- string[i] Accesses the character at index i of the string, starting at zero
- string[i:j] Accesses the substring starting at index i, ending at index j-1. If i is omitted, it's 0 by default. If j is omitted, it's len(string) by default.

String methods

- string.lower() / string.upper() Returns a copy of the string with all lower / upper case characters
- string.lstrip() / string.rstrip() / string.strip() Returns a copy of the string without left / right / left or right whitespace
- string.count(substring) Returns the number of times substring is present in the string
- string.isnumeric() Returns True if there are only numeric characters in the string. If not, returns False.
- string.isalpha() Returns True if there are only alphabetic characters in the string. If not, returns False.

- string.split() / string.split(delimiter) Returns a list of substrings that were separated by whitespace / delimiter
- string.replace(old, new) Returns a new string where all occurrences of old have been replaced by new.
- delimiter.join(list of strings) Returns a new string with all the strings joined by the delimiter

Check out the official documentation for all available String methods.

Formatting Strings Cheat Sheet

Python offers different ways to format strings. In the video, we explained the format() method. In this reading, we'll highlight three different ways of formatting strings. For this course you only need to know the format() method. But on the internet, you might find any of the three, so it's a good idea to know that the others exist.

Using the format() method

The format method returns a copy of the string where the {} placeholders have been replaced with the values of the variables. These variables are converted to strings if they weren't strings already. Empty placeholders are replaced by the variables passed to format in the same order.

```
# "base string with {} placeholders".format(variables)
example = "format() method"
formatted_string = "this is an example of using the {} on a string".format
  (example)
print(formatted_string)
"""Outputs:
this is an example of using the format() method on a string
"""
```

If the placeholders indicate a number, they’re replaced by the variable corresponding to that order (starting at zero).

```
# "{0} {1}".format(first, second)
first = "apple"
second = "banana"
third = "carrot"
formatted_string = "{0} {2} {1}".format(first, second, third)
print(formatted_string)
"""Outputs:
apple carrot banana
```

the placeholders indicate a field name, they're replaced by the variable corresponding to that field name. This means that parameters to format need to be passed indicating the field name.

```
"{:exp1} {:exp2}".format(value1, value2)
```

If the placeholders include a colon, what comes after the colon is a formatting expression. See below for the expression reference.

Official documentation for [the format string syntax]

```
# {:d} integer value

'{:d}'.format(10.5) → '10'
```

Formatting expressions

Expr	Meaning	Example
{:d}	integer value	'{:d}'.format(10.5) → '10'
{:.2f}	floating point with that many decimals	'{:.2f}'.format(0.5) → '0.50'
{:.2s}	string with that many characters	'{:.2s}'.format('Python') → 'Py'
{:<6s}	string aligned to the left that many spaces	'{:<6s}'.format('Py') → 'Py '
{:>6s}	string aligned to the right that many spaces	'{:<6s}'.format('Py') → ' Py'
{:^6s}	string centered in that many spaces	'{:<6s}'.format('Py') → ' Py '

Check out the official documentation for [all available expressions].

Old string formatting (Optional)

The format() method was introduced in Python 2.6. Before that, the % (modulo) operator could be used to get a similar result. While this method is **no longer recommended** for new code, you might come across it in someone else's code. This is what it looks like:

"base string with %s placeholder" % variable

The % (modulo) operator returns a copy of the string where the placeholders indicated by % followed by a formatting expression are replaced by the variables after the operator.

"base string with %d and %d placeholders" % (value1, value2)

To replace more than one value, the values need to be written between parentheses. The formatting expression needs to match the value type.

"%(var1) %(var2)" % {var1:value1, var2:value2}

Variables can be replaced by name using a dictionary syntax (we'll learn about dictionaries in an upcoming video).

"Item: %s - Amount: %d - Price: %.2f" % (item, amount, price)

The formatting expressions are mostly the same as those of the format() method.

Check out the official documentation for old string formatting.

Formatted string literals (Optional)

This feature was added in Python 3.6 and isn't used a lot yet. Again, it's included here in case you run into it in the future, but it's not needed for this or any upcoming courses.

A formatted string literal or f-string is a string that starts with 'f' or 'F' before the quotes. These strings might contain {} placeholders using expressions like the ones used for format method strings.

The important difference with the format method is that it takes the value of the variables from the current context, instead of taking the values from parameters.

Examples:

```
>>> name = "Micah"

>>> print(f'Hello {name}')

Hello Micah

>>> item = "Purple Cup"
```

```
>>> amount = 5

>>> price = amount * 3.25

>>> print(f'Item: {item} - Amount: {amount} - Price: {price:.2f}')

Item: Purple Cup - Amount: 5 - Price: 16.25
```

Check out the official documentation for f-strings.

Lists in Python are defined using square brackets, with the elements stored in the list separated by commas: **list = ["This", "is", "a", "list"]**. You can use the **len()** function to return the number of elements in a list: **len(list)** would return **4**. You can also use the **in** keyword to check if a list contains a certain element. If the element is present, it will return a True boolean. If the element is not found in the list, it will return False. For example, **"This" in list** would return True in our example. Similar to strings, lists can also use indexing to access specific elements in a list based on their position. You can access the first element in a list by doing **list[0]**, which would allow you to access the string **"This"**.

In Python, lists and strings are quite similar. They're both examples of sequences of data. Sequences have similar properties, like (1) being able to iterate over them using **for loops**; (2) support indexing; (3) using the **len** function to find the length of the sequence; (4) using the plus operator **+** in order to concatenate; and (5) using the **in** keyword to check if the sequence contains a value. Understanding these concepts allows you to apply them to other sequence types as well.

While lists and strings are both sequences, a big difference between them is that lists are mutable. This means that the contents of the list can be changed, unlike strings, which are immutable. You can add, remove, or modify elements in a list.

You can add elements to the end of a list using the **append** method. You call this method on a list using dot notation, and pass in the element to be added as a parameter. For example, **list.append("New data")** would add the string "New data" to the end of the list called list.

If you want to add an element to a list in a specific position, you can use the method **insert**. The method takes two parameters: the first specifies the index in the list, and the second is the element to be added to the list. So **list.insert(0, "New data")** would add the string "New data" to the front of the list. This wouldn't overwrite the existing element at the start of the list. It would just shift all the other elements by one. If you specify an index that's larger than the length of the list, the element will simply be added to the end of the list.

You can remove elements from the list using the **remove** method. This method takes an element as a parameter, and removes the first occurrence of the element. If the element isn't found in the list, you'll get a **ValueError** error explaining that the element was not found in the list.

You can also remove elements from a list using the **pop** method. This method differs from the remove method in that it takes an index as a parameter, and returns the element that was removed. This can be useful if you don't know what the value is, but you know where it's located. This can also be useful when you need to access the data and also want to remove it from the list.

Finally, you can change an element in a list by using indexing to overwrite the value stored at the specified index. For example, you can enter **list[0] = "Old data"** to overwrite the first element in a list with the new string "Old data".

As we mentioned earlier, strings and lists are both examples of sequences. Strings are sequences of characters, and are immutable. Lists are sequences of elements of any data type, and are mutable. The third sequence type is the tuple. Tuples are like lists, since they can contain elements of any data type. But unlike lists, tuples are immutable. They're specified using parentheses instead of square brackets.

You might be wondering why tuples are a thing, given how similar they are to lists. Tuples can be useful when we need to ensure that an element is in a certain position and will not change. Since lists are mutable, the order of the elements can be changed on us. Since the order of the elements in a tuple can't be changed, the position of the element in a tuple can have meaning. A good example of this is when a function returns multiple values. In this case, what gets returned is a tuple, with the return values as elements in the tuple. The order of the returned values is important, and a tuple ensures that the order isn't going to change. Storing the elements of a tuple in separate variables is called unpacking. This allows you to take multiple returned values from a function and store each value in its own variable.

When we covered *for* loops, we showed the example of iterating over a list. This lets you iterate over each element in the list, exposing the element to the for loop as a variable. But what if you want to access the elements in a list, along with the index of the element in question? You can do this using the **enumerate()** function. The enumerate() function takes a list as a parameter and returns a tuple for each element in the list. The first value of the tuple is the index and the second value is the element itself.

Lists and Tuples Operations Cheat Sheet

Lists and tuples are both sequences, so they share a number of sequence operations. But, because lists are mutable, there are also a number of methods specific just to lists. This cheat sheet gives you a run down of the common operations first, and the list-specific operations second.

Common sequence operations

- len(sequence) Returns the length of the sequence
- for element in sequence Iterates over each element in the sequence
- if element in sequence Checks whether the element is part of the sequence
- sequence[i] Accesses the element at index i of the sequence, starting at zero
- sequence[i:j] Accesses a slice starting at index i, ending at index j-1. If i is omitted, it's 0 by default. If j is omitted, it's len(sequence) by default.
- for index, element in enumerate(sequence) Iterates over both the indexes and the elements in the sequence at the same time

Check out the official documentation for sequence operations.

List-specific operations and methods

- list[i] = x Replaces the element at index i with x
- list.append(x) Inserts x at the end of the list
- list.insert(i, x) Inserts x at index i
- list.pop(i) Returns the element a index i, also removing it from the list. If i is omitted, the last element is returned and removed.
- list.remove(x) Removes the first occurrence of x in the list
- list.sort() Sorts the items in the list
- list.reverse() Reverses the order of items of the list
- list.clear() Removes all the items of the list
- list.copy() Creates a copy of the list
- list.extend(other_list) Appends all the elements of other_list at the end of list

Most of these methods come from the fact that lists are mutable sequences. For more info, see the official documentation for mutable sequences and the list specific documentation.

List comprehension

- [expression for variable in sequence] Creates a new list based on the given sequence. Each element is the result of the given expression.

- [expression for variable in sequence if condition] Creates a new list based on the given sequence. Each element is the result of the given expression; elements only get added if the condition is true.

Dictionaries are another data structure in Python. They're similar to a list in that they can be used to organize data into collections. However, data in a dictionary isn't accessed based on its position. Data in a dictionary is organized into pairs of keys and values. You use the key to access the corresponding value. Where a list index is always a number, a dictionary key can be a different data type, like a string, integer, float, or even tuples.

When creating a dictionary, you use curly brackets: **{}**. When storing values in a dictionary, the key is specified first, followed by the corresponding value, separated by a colon. For example, **animals = { "bears":10, "lions":1, "tigers":2 }** creates a dictionary with three key value pairs, stored in the variable animals. The key "bears" points to the integer value 10, while the key "lions" points to the integer value 1, and "tigers" points to the integer 2. You can access the values by referencing the key, like this: **animals["bears"]**. This would return the integer 10, since that's the corresponding value for this key.

You can also check if a key is contained in a dictionary using the **in** keyword. Just like other uses of this keyword, it will return True if the key is found in the dictionary; otherwise it will return False.

Dictionaries are mutable, meaning they can be modified by adding, removing, and replacing elements in a dictionary, similar to lists. You can add a new key value pair to a dictionary by assigning a value to the key, like this: **animals["zebras"] = 2**. This creates the new key in the animal dictionary called zebras, and stores the value 2. You can modify the value of an existing key by doing the same thing. So **animals["bears"] = 11** would change the value stored in the bears key from 10 to 11. Lastly, you can remove elements from a dictionary by using the **del** keyword. By doing **del animals["lions"]** you would remove the key value pair from the animals dictionary.

You can iterate over dictionaries using a *for* loop, just like with strings, lists, and tuples. This will iterate over the sequence of keys in the dictionary. If you want to access the corresponding values associated with the keys, you could use the keys as indexes. Or you can use the **items** method on the dictionary, like **dictionary.items()**. This method returns a tuple for each element in the dictionary, where the first element in the tuple is the key and the second is the value.

If you only wanted to access the keys in a dictionary, you could use the **keys()** method on the dictionary: **dictionary.keys()**. If you only wanted the values, you could use the **values()** method: **dictionary.values()**.

Dictionary Methods Cheat Sheet

Definition

x = {key1:value1, key2:value2}

Operations

- len(dictionary) - Returns the number of items in the dictionary
- for key in dictionary - Iterates over each key in the dictionary
- for key, value in dictionary.items() - Iterates over each key,value pair in the dictionary
- if key in dictionary - Checks whether the key is in the dictionary
- dictionary[key] - Accesses the item with key key of the dictionary
- dictionary[key] = value - Sets the value associated with key
- del dictionary[key] - Removes the item with key key from the dictionary

Methods

- dict.get(key, default) - Returns the element corresponding to key, or default if it's not present
- dict.keys() - Returns a sequence containing the keys in the dictionary
- dict.values() - Returns a sequence containing the values in the dictionary
- dict.update(other_dictionary) - Updates the dictionary with the items coming from the other dictionary. Existing entries will be replaced; new entries will be added.
- dict.clear() - Removes all the items of the dictionary

Check out the official documentation for dictionary operations and methods.