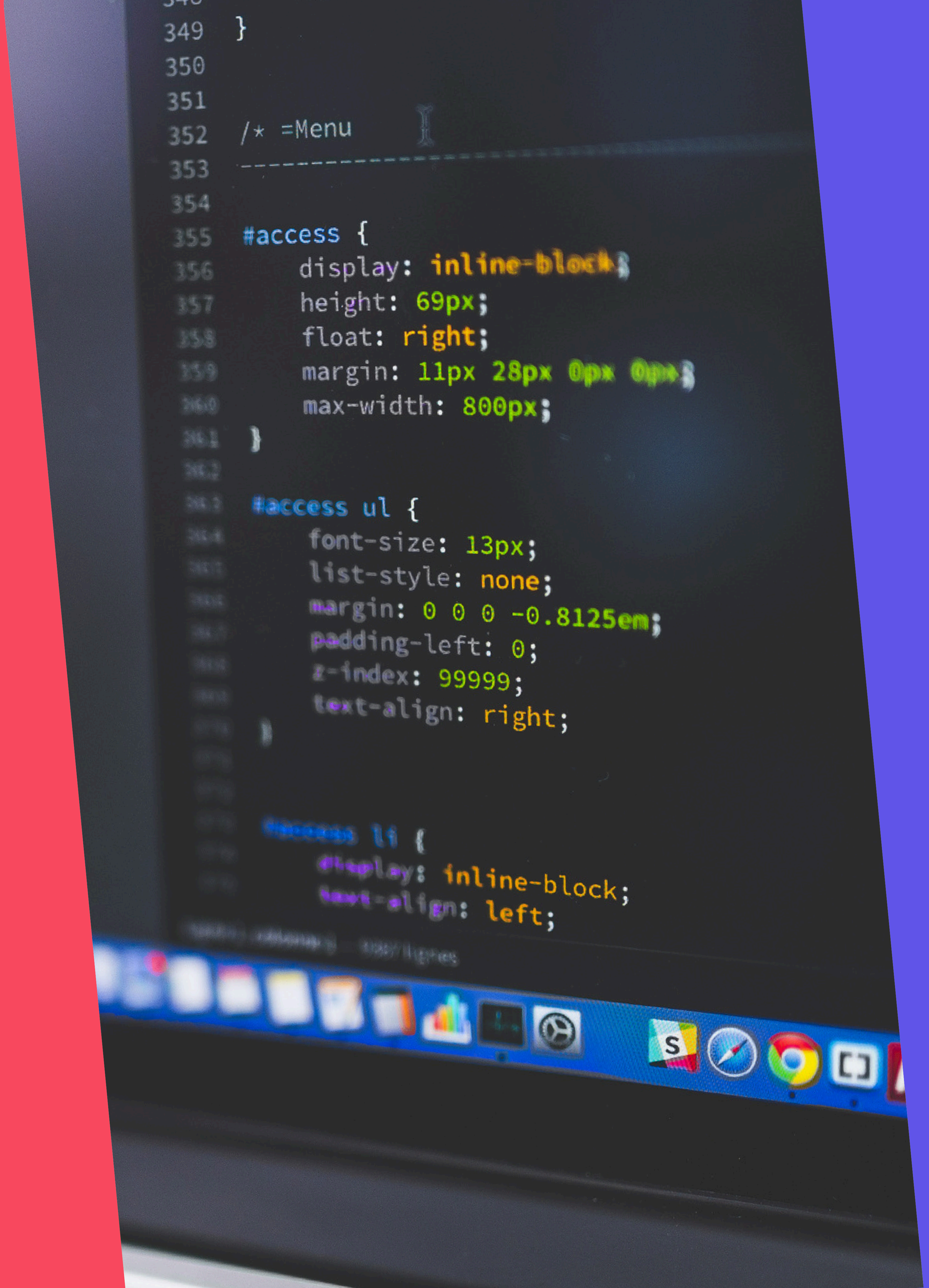


```
}


/* =Menu
---------------------------------

#access {
    display: inline-block;
    height: 69px;
    float: right;
    margin: 11px 28px 0px 0px;
    max-width: 800px;
}

#access ul {
    font-size: 13px;
    list-style: none;
    margin: 0 0 0 -0.8125em;
    padding-left: 0;
    z-index: 99999;
    text-align: right;
}


#access li {
    display: inline-block;
    text-align: left;
```

WBS CODING SCHOOL

Collections of Data

WBS CODING SCHOOL

Agenda

- What is a Collection?
- Lists
- Tuples
- Sets
- Dictionaries

WBS CODING SCHOOL

Learning outcomes

Skills you will acquire:

1.Understand what is a Collection

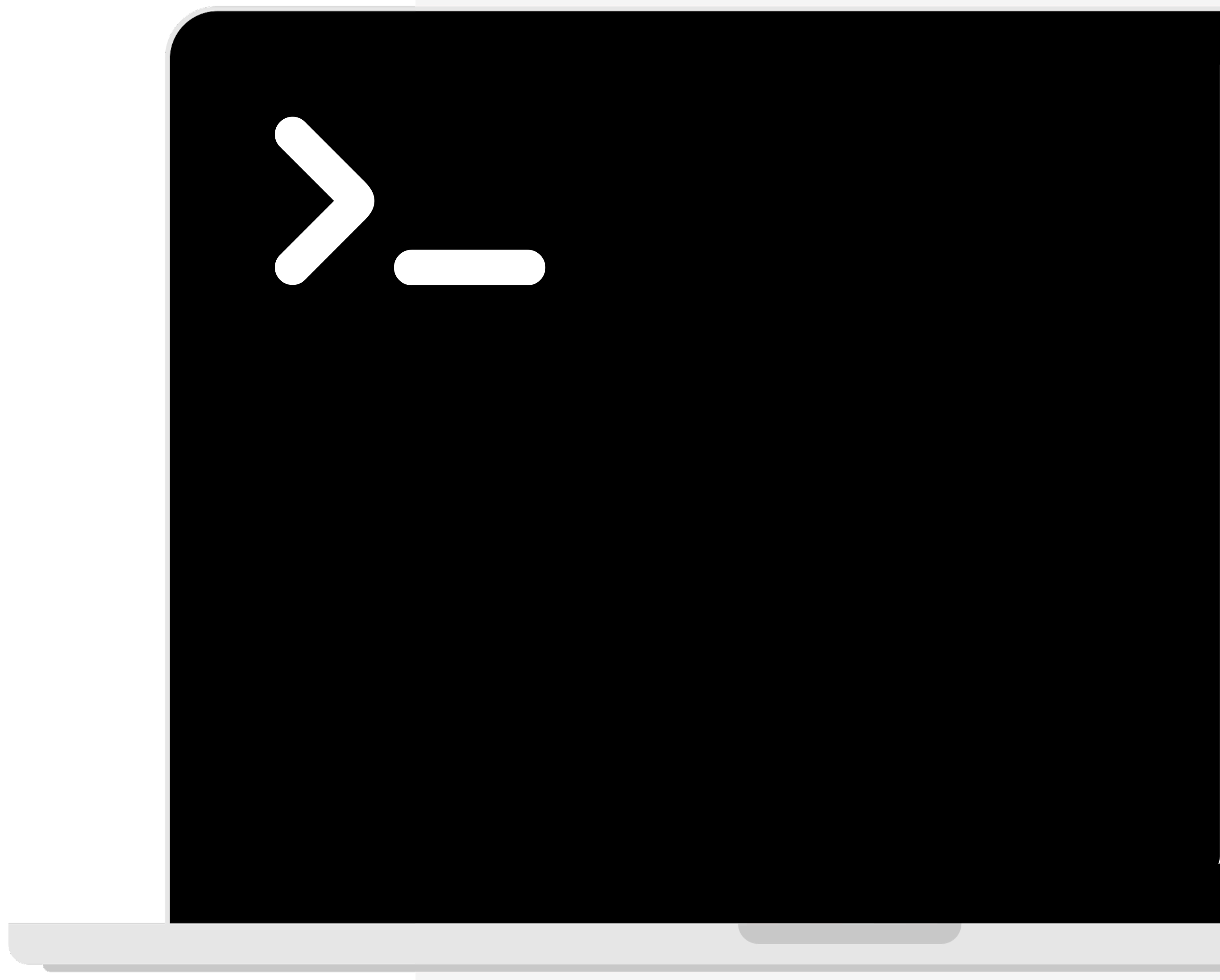
2. Outline the types of Collections

3. Know the characteristis of each Collection

4. Understand how to interact with each type of Collection

What is a Collection?

Types And Uses



What is a Collection?

A collection in Python is a container used to store multiple items in a single variable.

Think of a collection like a box that can hold many things inside — like numbers, words, or even other boxes! These are helpful when you want to group related data together.

Types of Collections in Python

There are four built-in collection types in Python. Each works in a slightly different way:

Type	Description	Example
List	An **ordered**, **changeable** collection. Allows **duplicates**.	["apple", "banana", "apple"]
Tuple	An **ordered**, **unchangeable** collection. Allows **duplicates**.	("apple", "banana", "apple")
Set	An **unordered**, **unchangeable** (but you can add/remove), **no duplicates**.	{"apple", "banana"}
Dictionary (dict)	A collection of **key-value pairs**, **unordered** in older versions, **ordered** in Python 3.7+.	{"name": "Alice", "age": 30}

Types of Collections in Python

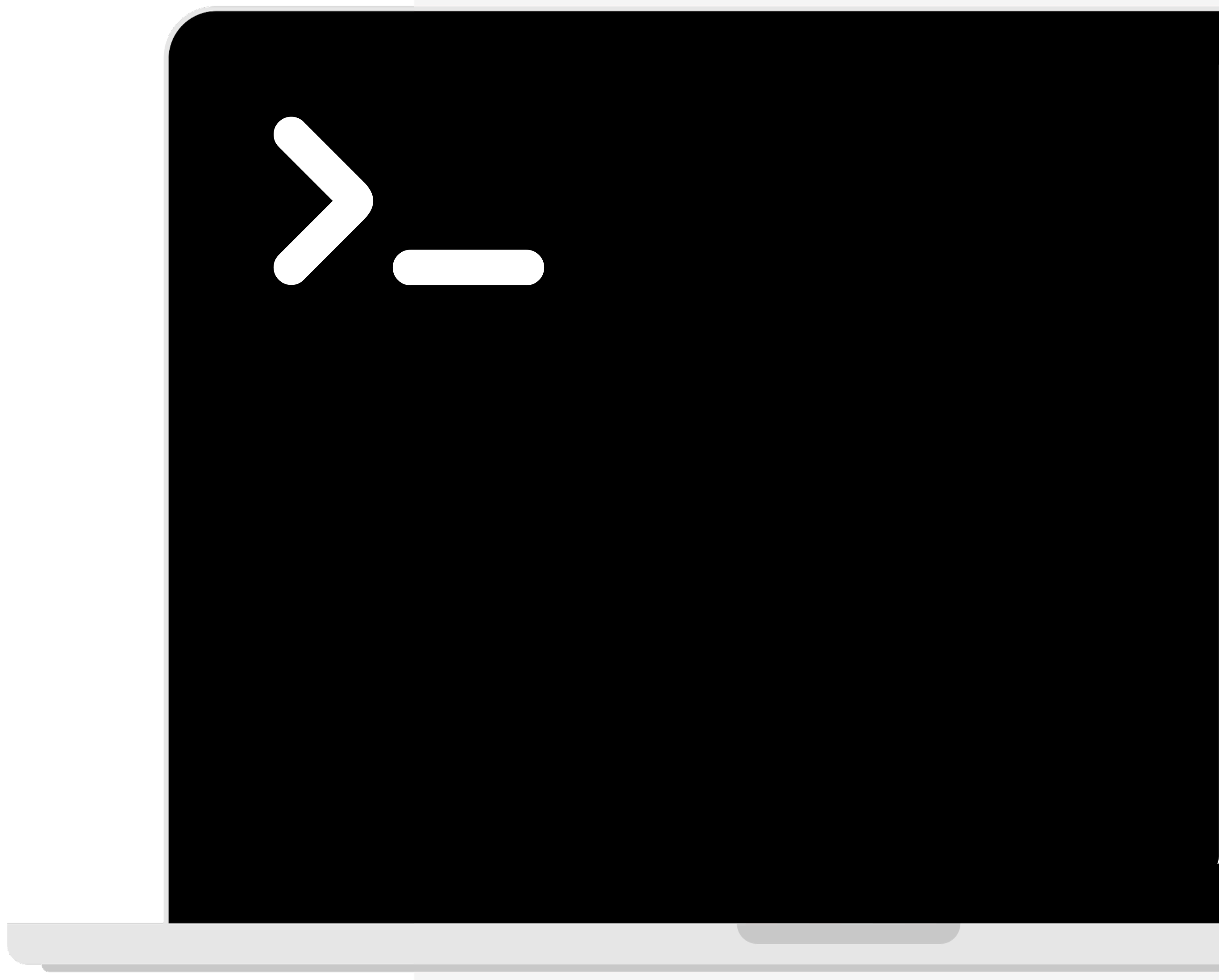
Feature	List	Tuple	Set	Dictionary
Ordered	✅	✅	❌	✅ (3.7+)
Changeable	✅	❌	❌*	✅
Duplicates	✅	✅	❌	✅ (keys: ❌)
Key Access	❌	❌	❌	✅

Types of Collections in Python

```python
# List
fruits = ["apple", "banana", "cherry"]

# Tuple
colors = ("red", "green", "blue")

# Set
unique_numbers = {1, 2, 3}

# Dictionary
person = {"name": "John", "age": 25}
```

Lists

Mutable Collections of Data



List are...

- **Collections** - you can use them to store multiple values in one variable.
- **Ordered** - each item has a specific position or index:
  - The indexes start on 0.
  - You can use the index to access an item or a sequence of items.
- **Changeable** - You can add, change, or remove items.
- **Allows duplicates** - You can have repeated items.

```
shopping_list = ["milk", "eggs", "bread"]
```

Creating a List

- Use the square brackets []
- Add a coma between each item
- You can use any data type, and mix different data types within the list.
- You can create a list from another list with a for loop or using list methods.

```
fruits = ["apple", "banana", "cherry"]

upper_fruits = [fruit.upper() for fruit in fruits]

lower_fruits = [fruit.lower() for fruit in upper_fruits if "a" in fruit]
```

List Comprehension

- It helps to create a list from another list with modification. Basic syntax:
  - expression
  - for loop
  - condition (optional)

```python
fruits = ["apple", "banana", "cherry"]

upper_fruits = [fruit.upper() for fruit in fruits]

lower_fruits = [fruit.lower() for fruit in upper_fruits if "a" in fruit]
```

Reference to a List

By giving a variable the value of another variable with a list as its value:

- Creates a reference to that list
- Any changes done to one list will affect both lists.

```
fruits = ["apple", "banana", "cherry"]

fruits_2 = fruits # creates a reference to fruits, so changes will affect both lists
```

Accessing Items in a List

To acces the items of a list, use the indexes:

```python
shopping_list = ["milk", "eggs", "honey", "bread"]

shopping_list[0] # selects the first item (milk)
shopping_list[-1] # selects the last item (bread)
shopping_list[1:3] # selects the items from index 1 to 2 (eggs and honey)
shopping_list[:3] # selects from the first item to item with index 2 (milk and eggs)
shopping_list[3:] # selects item with index 3 till the last item (honey and bread)
```

Modifying Items in a List

- You can use the index or a range of index to update the value of an item in the list:

```
shopping_list = ["milk", "eggs", "honey", "bread"]

shopping_list[2] = "apples" # updates the list to ["milk", "eggs", "apples", "bread"]

shopping_list[1:3] = ["apples", "juice"] # updates the list to ["milk", "apples", "juice", "bread"]
```

Built-in List Methods

List methods are built-in functions in Python that you can use to work with lists – like **adding**, **removing**, **sorting**, or **copying** items.

Think of them as tools you can use to manipulate your list.

Lists can change, unlike strings, so some of this methods **will update the original list**.

Built-in List Methods

Method	What it does	Example
append(item)	Adds item to the end	my_list.append(5)
extend(list)	Adds the items of another list at the end	my_list.extend(my_other_list)
insert(i, item)	Inserts item at index i	my_list.insert(0, "hello")
remove(item)	Removes first occurrence of item	my_list.remove("apple")
pop([i])	Removes item at index i or last	my_list.pop()
clear()	Removes all items	my_list.clear()
index(item)	Returns index of item	my_list.index("banana")
count(item)	Counts how many times item appears	my_list.count("apple")
sort()	Sorts the list in ascending order	my_list.sort()
reverse()	Reverses the order of the list	my_list.reverse()
copy()	Returns a copy of the list	new_list = my_list.copy()

Built-in List Methods

```
fruits = ["apple", "banana", "cherry"]

fruits.append("orange")       # ['apple', 'banana', 'cherry', 'orange']
fruits.insert(1, "mango")     # ['apple', 'mango', 'banana', 'cherry', 'orange']
fruits.remove("banana")       # ['apple', 'mango', 'cherry', 'orange']
fruits.pop()                  # removes 'orange'
fruits.sort()                 # sorts alphabetically
fruits.reverse()              # reverses order
copy_fruits = fruits.copy()   # creates a copy
```

Useful Built-in Functions

Function	What It Does	Example
len(list)	Returns the **number of items**	len(fruits)
sum(list)	Adds up all numbers in a list	sum([1, 2, 3]) → 6
min(list)	Returns the **smallest** item	min([5, 2, 8]) → 2
max(list)	Returns the **largest** item	max([5, 2, 8]) → 8
sorted(list)	Returns a **new sorted list** (doesn't change original)	sorted([3, 1, 2]) → [1, 2, 3]
list()	Converts another data type into a list	list("abc") → ['a','b','c']
reversed()	Returns a **reversed iterator**. Used with **list()** can create a reversed copy of a list	list(reversed([1,2,3])) → [3,2,1]

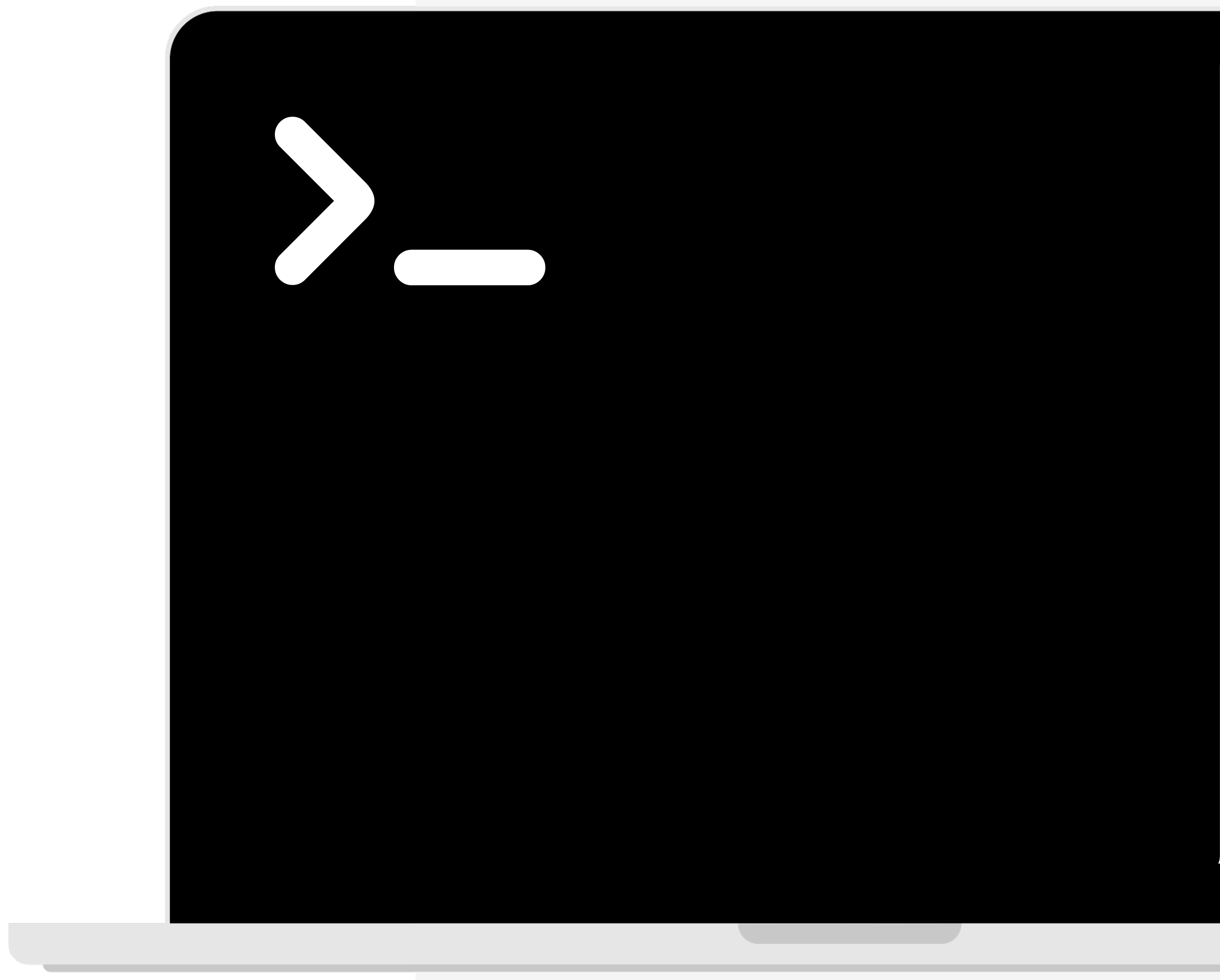
Modules

There are also some modules that can be used with lists that can be quite helpfull:

Module	What It's For	Example Use
copy	Copying lists (shallow or deep)	copy.copy(list)
random	Random items and shuffling	random.shuffle(list)
statistics	Averages, medians, stats	statistics.mean(list)
itertools	Combinations, loops, advanced tools	itertools.product(list1, list2)
collections	Counting, advanced data structures	Counter(list)
json	Saving/loading lists as text	json.dumps(list)
math	Math operations on numbers	math.sqrt(x)

Tuples

Immutable Collections of Data



Tuples are...

- **Collections** - you can use them to store multiple values in one variable.
- **Ordered** - each item has a specific position or index:
    - The indexes start on 0.
    - You can use the index to access an item or a sequence of items.
- **Immutable** - You cannot modify them once created.
- **Allows duplicates** - You can have repeated items.

```
my_tuple = ("apple", "banana", "cherry")
```

Accessing Items in a Tuple

You use indexing, just like lists:

```
fruits = ("apple", "banana", "cherry")

print(fruits[0])  # 'apple'
print(fruits[-1]) # 'cherry'
print(fruits[1:3])  # ('banana', 'cherry')
```

Unpacking a Tuple

Tuple unpacking means splitting a tuple into individual variables.

- If you try to unpack into the wrong number of variables, you'll get an error
- When you only want some values, you can use * to grab the resting values

```python
person = ("Alice", 25, "Paris")

name, age, city = person

print(name)  # Alice
print(age)   # 25
print(city)  # Paris
```

```python
data = (1, 2, 3, 4, 5)

first, *middle, last = data

print(first)    # 1
print(middle)   # [2, 3, 4]
print(last)     # 5
```

Built-in Tuple Methods

Tuples have only two methods:

Method	Description	Example
.count(x)	Counts how many times x appears	my_tuple.count("apple")
.index(x)	Returns the index of the first occurrence	my_tuple.index("banana")

Useful Built-in Functions with Tuples

Function	What It Does	Example
len()	Number of items in the tuple	len(my_tuple)
sum()	Adds all numbers in the tuple	sum((1, 2, 3)) → 6
min()	Smallest item	min((5, 2, 9)) → 2
max()	Largest item	max((5, 2, 9)) → 9
tuple()	Converts another data type to a tuple	tuple("abc") → ('a','b','c')

Useful Modules with Tuples

Module	**Use Case Example**
collections	namedtuple lets you create tuple-like objects with named fields
itertools	Useful for looping, combining, and processing tuples
operator	Contains functions for sorting and comparing tuples

Joining Tuples

There are two ways of joining Tuples:

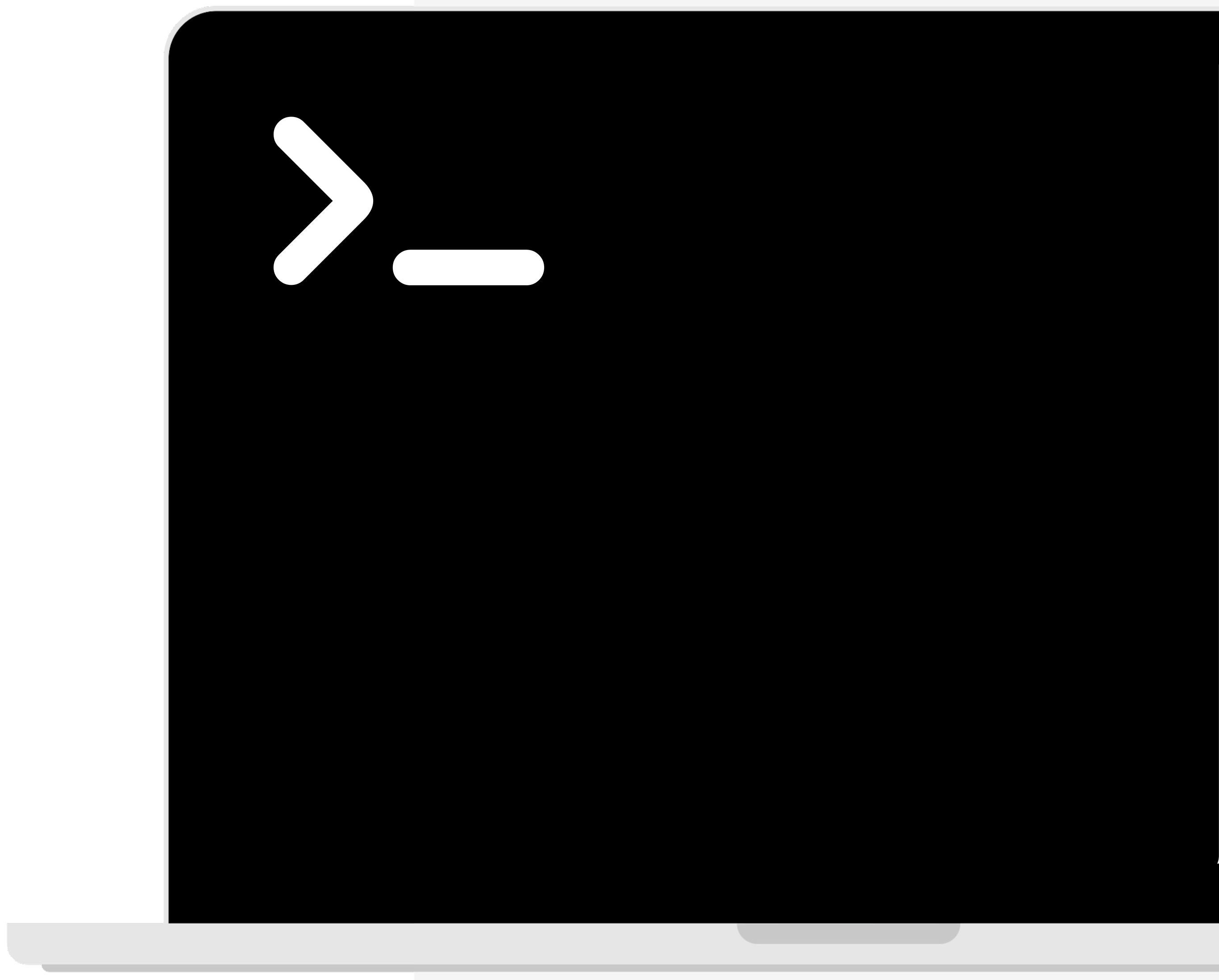
- **Addition operator (+)** - creates a new tuple containing elements from each original tuple in order.

- **Multiplication operator (*)** - creates a new tuple that contains repeated sequences of the original tuple's items.

```python
a = (1, 2, 3)
b = (4, 5)

joined = a + b
print(joined)  # (1, 2, 3, 4, 5)

repeat = a * 2
print(repeat)  # (1, 2, 3, 1, 2, 3)
```

Sets

Unordered Collections of Data



Sets are...

- **Collections** - you can use them to store multiple values in one variable, however they can only hold immutable data types as its items.
- **Unordered** - because of this, access by indexing is not possible.
- **Changeable** - You can add and remove items, but not modify the items themselves.
- **Doesn't allow duplicates** - You cannot have repeated items.

```python
fruits = {"apple", "banana", "cherry"}
```

Accessing Items in a Set

You can’t access by index (like in lists or tuples), but you can **loop**:

```
for fruit in fruits:
    print(fruit)
```

Or use the keyword **in** to check if an item is in a set:

```
if "banana" in fruits:
    print("Banana is in the set")
```

Built-in Sets Methods

Method	Description	Example
.add(x)	Adds an item	fruits.add("orange")
.update(x)	Adds a sequence of items from anoter iterable	fruits.update(fruits_1)
.remove(x)	Removes an item (error if not found)	fruits.remove("apple")
.discard(x)	Removes an item (no error if missing)	fruits.discard("kiwi")
.pop()	Removes a random item	fruits.pop()
.clear()	Removes all items	fruits.clear()
.copy()	Returns a shallow copy of the set	new_set = fruits.copy()

Built-in Sets Methods - Joining Sets

Method	Description	Example
.update(x)	Adds a sequence of items from anoter iterable	fruits.update(tropical_fruits)
.union(x)	Combines two sets removing duplicates	fruits.union(tropical_fruits)
.intersection(x)	Returns elements common to two or more sets	fruits.intersection(tropical_fruits)
.difference(x)	Returns elements in one set but not in another	fruits.difference(tropical_fruits)
.symmetric_difference(x)	Returns elements in either set but not in both (excludes common items).	fruits.symmetric_difference(tropical_fruits)

Useful Built-in Functions with Sets

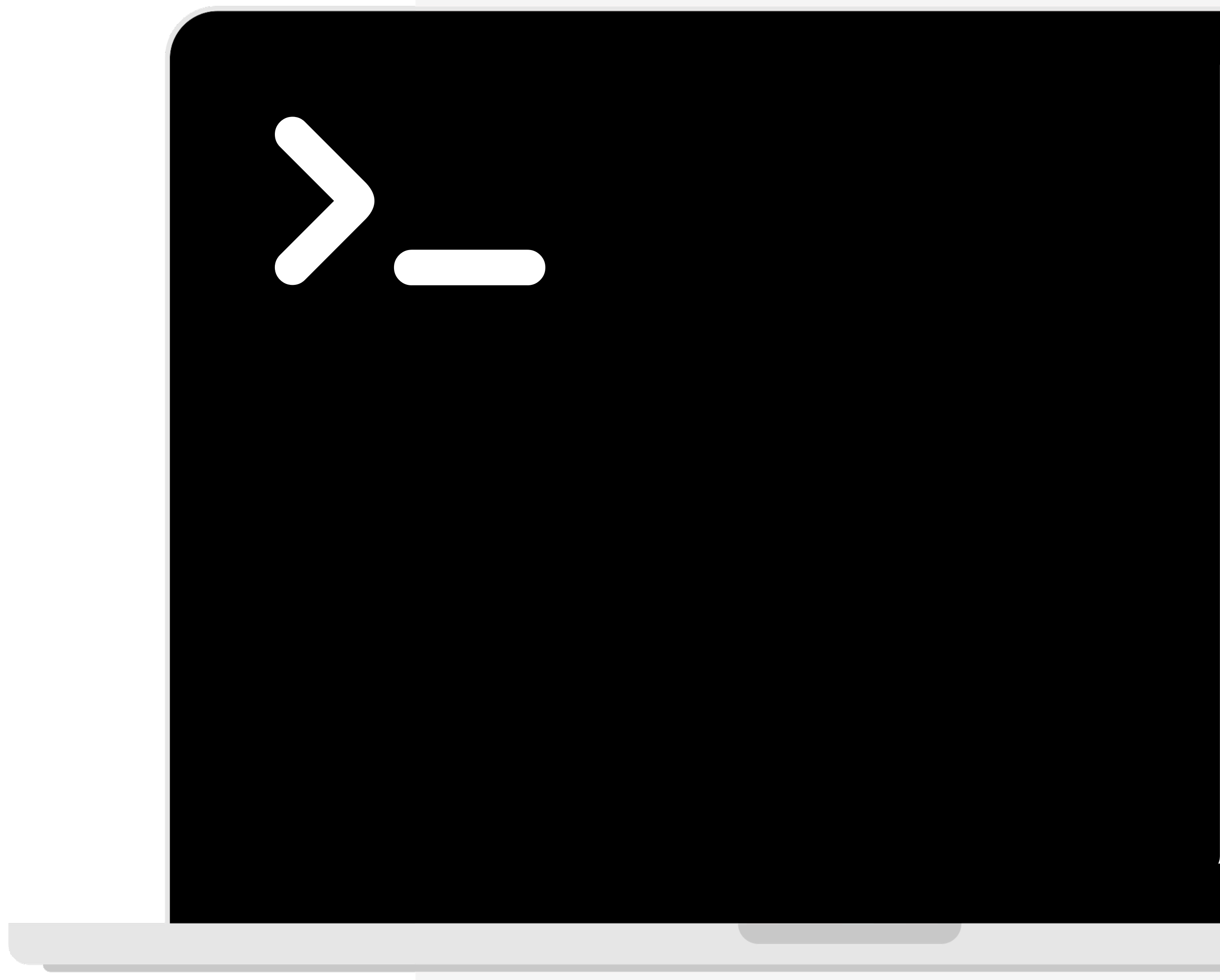
Function	**What It Does**	**Example**
len()	Number of items	len(fruits)
set()	Converts list or tuple to a set	set([1, 2, 2, 3]) → {1, 2, 3}
sorted()	Returns a sorted list version of the set	sorted(fruits)

Useful Modules with Sets

Module	**Use Case Example**
collections	Has Counter, which can be useful before converting to a set
itertools	Helps with advanced combinations and operations on sets
math	For set operations involving numbers (indirectly useful)

Dictionaries

Unique Collections of Data



Dictionaries are...

- **Collections** of key-value pairs. Each key has a value associated with it.
- **Ordered** - since python 3.7, however indexing is done by keys, not position.
- **Changeable** - You can add and remove items. You can change the value of a key, but keys are immutable and must be an immutable data type.
- **Doesn’t allow duplicates** - You cannot have repeated keys.

```
person = {
    "name": "Alice",
    "age": 25,
    "city": "Paris"
}
```

Accessing Items in a Dictionary

Use square notation with the key to get the value:

```
print(person["name"])  # Alice
```

You can also use .get() to safely access a key:

```
print(person.get("age"))            # 25
print(person.get("height", "N/A"))  # "N/A" if not found
```

Adding or Updating Items in a Dictionary

Use square notation with the key you want to create or update and use = to assign a value to it:

```
person["age"] = 26  # Update
person["country"] = "France"  # Add new key-value pair
```

Removing Items from a Dictionary

You can use the **pop()** or the **clear()** method or the keyword **del**:

```
person.pop("city")          # Removes "city"
del person["age"]           # Also removes "age"
person.clear()              # Removes everything
```

Built-in Dictionaries Methods

Method	Description
.get(key)	Returns value, or default if missing
.keys()	Returns all keys
.values()	Returns all values
.items()	Returns all key-value pairs (as tuples)
.update()	Adds or updates items from another dict
.pop(key)	Removes the key and returns its value
.clear()	Empties the dictionary
.copy()	Makes a shallow copy

Useful Built-in Functions with Dictionaries

Function	**Description**	**Example**
len()	Number of key-value pairs	len(person)
str()	Converts dictionary to a string	str(person)
type()	Shows the type of the object	type(person)
dict()	Creates a dictionary from tuples/lists	dict([("a", 1), ("b", 2)])

WBS CODING SCHOOL

Lesson completed