MODULE 2
INTRODUCTION TO PYTHON PROGRAMMING

Syllabus : **Lists:** The List Data Type, Working with Lists, Augmented Assignment Operators, Methods, Example Program: Magic 8 Ball with a List, List-like Types: Strings and Tuples, References,**Dictionaries and Structuring Data:** The Dictionary Data Type, Pretty Printing, Using Data Structures to Model Real-World Things,
Textbook 1: Chapter 4-5

1. LISTS

A **list** is a type of value in Python that lets you store multiple items (or values) in an ordered sequence. It's like a container that holds multiple values, all arranged in a specific order. Lists can hold different types of data, including numbers, strings, and even other lists.

What does a list look like?

- A list starts and ends with square brackets [].
- Items inside the list are separated by commas.
- For example [1, 2, 3]
- ['cat', 'bat', 'rat', 'elephant']

What can a list hold?

- A list can contain:
  - Numbers: [1, 2, 3]
  - Strings: ['cat', 'bat']
  - A mix of different types: ['hello', 3.1415, True, None, 42]
  - Even another list: [1, 2, ['a', 'b'], 4]

What is an empty list?

- A list with no items in it is called an **empty list**. It looks like this: [].

Lists as variables:

- You can store a list in a variable. For example
- spam = ['cat', 'bat', 'rat', 'elephant']

Here, the variable `spam` is assigned a **single value**, which is the list itself. Inside the list, there are four items: `'cat'`, `'bat'`, `'rat'`, and `'elephant'`.

Examples

1. **Basic Lists:**

# A list of numbers

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

# A list of strings
animals = ['dog', 'cat', 'rabbit']

# A mixed list
mixed = ['hello', 10, 3.14, True]
```

2. Empty List:

```
empty_list = []
print(empty_list)
# Output: []
```

3. Accessing a List:

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

Summary

- A list is a way to store multiple items together in Python.
- It is flexible, can store any type of data, and is ordered (the order of items matters).
- Use square brackets [] to create a list and commas to separate the items.

Real-Time Example: Grocery Shopping List [for understanding purpose]

Imagine you're going grocery shopping, and you want to keep track of the items you need to buy. A **list** in Python can represent your shopping list.

```
# Grocery shopping list
grocery_list = ['milk', 'bread', 'eggs', 'butter', 'fruits']

# Printing the list
print("My grocery list:", grocery_list)

# Adding a new item to the list
grocery_list.append('vegetables')
print("Updated grocery list:", grocery_list)

# Checking the first item on the list
print("First item I need to buy:", grocery_list[0])

# Removing an item after buying
grocery_list.remove('bread')
print("After buying bread, the list is:", grocery_list)

# Counting how many items are left
print("Total items left to buy:", len(grocery_list))
```

Step-by-Step Breakdown:

1. **Create a List:**
   - ['milk', 'bread', 'eggs', 'butter', 'fruits'] is your shopping list.
2. **Accessing Items:**
   - You can check specific items using their position (e.g., grocery_list[0] gives 'milk').
3. **Adding New Items:**
   - When you remember to buy vegetables, you can add it to the list using append().
4. **Removing Items:**
   - After buying bread, remove it from the list using remove().
5. **Counting Items:**
   - Use len() to find out how many items are still left on the list.

OUTPUT

```
My grocery list: ['milk', 'bread', 'eggs', 'butter', 'fruits']

Updated grocery list: ['milk', 'bread', 'eggs', 'butter', 'fruits', 'vegetables']

First item I need to buy: milk

After buying bread, the list is: ['milk', 'eggs', 'butter', 'fruits', 'vegetables']

Total items left to buy: 5
```

Conclusion: This real-life example of a grocery list helps to understand:

- Lists store multiple items in order.
- You can add, remove, and manage items dynamically.

2. LIST INDEXES IN PYTHON

A list index is a number that tells Python which specific value you want to access in a list. Every item in a list is assigned a position called an index, starting from 0.

For example: spam = ['cat', 'bat', 'rat', 'elephant']

Here are the indexes for each item:

- `'cat'` is at index `0`
- `'bat'` is at index `1`
- `'rat'` is at index `2`
- `'elephant'` is at index `3`

How Indexing Works

To get a value from a list, use the list name followed by the index in square brackets []

```
spam = ['cat', 'bat', 'rat', 'elephant']
```

print(spam[0]) # Output: 'cat'

print(spam[1]) # Output: 'bat'

print(spam[2]) # Output: 'rat'

print(spam[3]) # Output: 'elephant'

Using Indexes with Expressions

Indexes can be used in expressions or combined with strings.

Example: spam = ['cat', 'bat']

# Combining list values with strings

print("Hello, " + spam[0])

# Output: Hello, cat

print("The " + spam[1] + " ate the " + spam[0] + ".")

# Output: The bat ate the cat.

Common Errors with Indexes

1. IndexError: List index out of range

- This error happens when you try to access an index that doesn't exist.
- spam = ['cat', 'bat']
- print(spam[5]) # Error: IndexError

2. TypeError: List indices must be integers

- List indexes must be whole numbers (integers). You can't use decimals.
- spam = ['cat', 'bat']
- print(spam[1.0]) # Error: TypeError

3. Lists Inside Lists (Nested Lists)

Lists can contain other lists as their items. You can use **multiple indexes** to access the values inside these nested lists.

# Nested list
spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]

# Access the first list
print(spam[0]) # Output: ['cat', 'bat']

```
# Access the second item in the first list
print(spam[0][1])  # Output: 'bat'

# Access the fifth item in the second list
print(spam[1][4])  # Output: 50
```

Conclusion

1. **Indexes**: Start at 0 and are used to access specific items in a list.
2. **Combining Values**: You can combine list values with strings or other expressions.
3. **Errors**: Be careful not to use invalid indexes or non-integer types.
4. **Nested Lists**: Use multiple indexes to access values inside lists of lists.

3. NEGATIVE INDEXES IN LIST

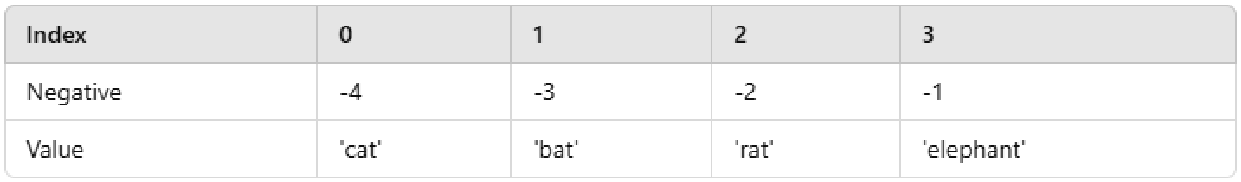
In Python, you can use **negative indexes** to access items in a list by counting backward from the end of the list. This is useful when you want to work with the last items in a list but don't know or want to count how many items are in it.

How Negative Indexes Work

1. The **last item** in the list is at index -1.
2. The **second-to-last item** is at index -2.
3. The **third-to-last item** is at index -3, and so on.

Ex: Here's a list to explain:

```
spam = ['cat', 'bat', 'rat', 'elephant']
```



Index	0	1	2	3
Negative	-4	-3	-2	-1
Value	'cat'	'bat'	'rat'	'elephant'

Example 1: Accessing Values

```
spam = ['cat', 'bat', 'rat', 'elephant']

# Access the last item
print(spam[-1])
 # Output: 'elephant'

# Access the second-to-last item
print(spam[-2])
# Output: 'rat'
# Access the third-to-last item
print(spam[-3])
# Output: 'bat'
```

Example 2: Combining Values with Strings

```
spam = ['cat', 'bat', 'rat', 'elephant']

# Use negative indexes in a sentence
sentence = 'The ' + spam[-1] + ' is afraid of the ' + spam[-3] + '.'
print(sentence)
# Output: 'The elephant is afraid of the bat.'
```

Why Use Negative Indexes?

Negative indexes are especially useful when:

1. You want to access the last few items in a list, no matter its length.
2. You don't want to calculate the exact index from the front of the list.

Conclusion

- Negative indexes start at -1 for the last item and count backward.
- They are a quick and flexible way to access the end of a list.

4. NEGATIVE INDEXES IN LIST

Slicing is a way to extract a portion of a list to create a new list. Instead of accessing just one item using a single index, you can use slicing to get multiple items by specifying a start and an end position.

How Slicing Works

A slice uses two numbers separated by a colon : inside square brackets:

```
list[start:end]
```

- start: The index where the slice begins (inclusive, meaning it includes this item).
- end: The index where the slice ends (exclusive, meaning it does **not** include this item).

The result of slicing is a **new list** that contains the items between the start and end indexes.

```
spam = ['cat', 'bat', 'rat', 'elephant']

# Slice from index 0 to 4
print(spam[0:4])
# Output: ['cat', 'bat', 'rat', 'elephant']
# Slice from index 1 to 3
print(spam[1:3])
# Output: ['bat', 'rat']

# Slice from index 0 to -1
print(spam[0:-1])
# Output: ['cat', 'bat', 'rat']
```

Shortcut Rules for Slicing

1. Leaving Out the Start Index:

1. If you don't specify the start index, Python assumes it to be 0 (the beginning of the list).
2. spam = ['cat', 'bat', 'rat', 'elephant']
3. print(spam[:2])
4. # Output: ['cat', 'bat']

2. Leaving Out the End Index:

- If you don't specify the end index, Python assumes it to be the length of the list (slice until the end).
- print(spam[1:]) # Output: ['bat', 'rat', 'elephant']

3. Leaving Out Both Start and End Indexes:

- If you leave out both, Python assumes you want the entire list.
- print(spam[:]) # Output: ['cat', 'bat', 'rat', 'elephant']

Conclusion

- **Indexes vs. Slices**:
  - spam[2] → Gets **one item** (the item at index 2).
  - spam[1:4] → Gets **multiple items** (from index 1 to 3).
- **Start and End Rules**:
  - The start index is included in the slice.
  - The end index is not included in the slice.
- **Shortcut**:
  - Leaving out start means "from the beginning."
  - Leaving out end means "until the end."
  - Leaving out both means "the whole list."

5. GETTING A LIST'S LENGTH WITH THE LEN() FUNCTION

The len() function is like a counting tool. It tells you how many items are inside something.

- If you give it a list (like a shopping list or a list of pets), it will count how many things are in the list.
- For example, if your list is ['cat', 'dog', 'moose'], there are 3 items in it. So, when you use len(spam), it gives you 3.

It's just like asking, "How many things are on my list?" and len() does the counting for you

6. CHANGING VALUES IN A LIST WITH INDEXES

A list is like a row of boxes where each box holds a value (like a word or number). You can change what's inside any box using its position (called its index). Here's how it works:

1. Changing an item at a specific position:

Each box in the list has a number (index), starting from 0 for the first box, 1 for the second box, and so on.For example:

spam = ['cat', 'bat', 'rat', 'elephant']

The box at index 1 (second item) holds 'bat'.

When you write

spam[1] = 'aardvark'

- It means, "Replace what's in box 1 with `'aardvark'`."

Now the list becomes: ['cat', 'aardvark', 'rat', 'elephant']

2. Using values from one box to change another:

If you want to copy a value from one box to another, you can do it like this:

spam[2] = spam[1]

This means, "Take the value in box 1 (`'aardvark'`) and put it in box 2."

Now the list looks like: ['cat', 'aardvark', 'aardvark', 'elephant']

3. Using negative indexes to count backward:

Negative indexes let you count from the end of the list. For example:

- -1 means the **last box**,
- -2 means the second-to-last box, and so on.

spam[-1] = 12345
It changes the **last box** to 12345.
Now the list becomes: ['cat', 'aardvark', 'aardvark', 12345]

Conclusion

- You can use an **index** to find or change a specific box in the list.
- Positive indexes count from the start (0, 1, 2, …).
- Negative indexes count backward from the end (-1, -2, -3, …).

It's like pointing to a specific box and saying, “Hey, I want to change what’s in here!

7. CHANGING VALUES IN A LIST WITH INDEXES

1. List Concatenation (+):

The + operator joins two lists together, just like gluing two pieces of paper.

For example: [1, 2, 3] + ['A', 'B', 'C']

- You take the first list [1, 2, 3].
- Add the second list ['A', 'B', 'C'] to it.

The result is a new list: [1, 2, 3, 'A', 'B', 'C']

Think of it as combining two shopping lists into one!

2. List Replication (*):

The * operator repeats a list multiple times, like making copies of something.

For example:['X', 'Y', 'Z'] * 3

This means “repeat the list ['X', 'Y', 'Z'] 3 times.”

The result is: ['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']

It’s like copying the same list again and again to make it longer.

3. Updating a list with +:

If you already have a list and use + to add more items to it, you can update the list.

For example:

```
spam = [1, 2, 3]

spam = spam + ['A', 'B', 'C']
```

- Start with spam = [1, 2, 3].
- Add ['A', 'B', 'C'] to it using +.
- Now, spam becomes [1, 2, 3, 'A', 'B', 'C']

Conclusion

- + joins lists together to make one big list.
- * repeats a list as many times as you want.
- You can update a list by using these operators.

It’s like playing with building blocks: you can combine or repeat blocks however you like

8. REMOVING VALUES FROM LISTS WITH DEL STATEMENTS

1. What does del do? The del statement is like an eraser. It removes something, either from a list or a variable.

2. Using del with a list: When you use del on a list with a specific index, it removes the value at that index.

For example: spam = ['cat', 'bat', 'rat', 'elephant']

del spam[2]

- Start with the list: ['cat', 'bat', 'rat', 'elephant'].
- del spam[2] means "delete the value at index 2" (which is 'rat').

After deleting 'rat', the list becomes:

['cat', 'bat', 'elephant']

3. What happens to the list? When you remove something, all the values after it move up to fill the gap. So, if you delete another value at index 2:

del spam[2]

The list becomes: ['cat', 'bat']

4. Using del with variables: You can also use del to completely remove a variable. For example:

x = 10

del x

Now, the variable x is erased. If you try to use x after this, Python will give you an error because it's gone.

Conclusion::

- del is like an eraser that removes specific values from a list or completely deletes a variable.
- After deleting something in a list, the other items shift to fill the empty spot.

It's like crossing out an item on a to-do list and moving everything else up

9. WORKING WITH LISTS

The Problem with Too Many Variables: When you have a lot of similar data (like the names of your cats), it's tempting to create a separate variable for each one:

catName1 = 'Zophie'

catName2 = 'Pooka'

catName3 = 'Simon'

But this approach has problems:

1. Limited flexibility: If you get more cats, you need to create more variables.
2. Repetitive code: You'll write the same kind of code over and over again.
3. Hard to manage: If you need to do something with all the names (like print them), you have to refer to each variable manually.

The Solution: Use a List

A list is like a container that can hold many values. Instead of creating separate variables, you put all the names into one list:

```
catNames = ['Zophie', 'Pooka', 'Simon', 'Lady Macbeth', 'Fat-tail', 'Miss Cleo']
```

Now, the list holds all your cat names in one place, and you can easily add, remove, or work with them.

Example 1: Old Way (Repetitive Code)

Here's how the program looks if you don't use a list:

```
print('Enter the name of cat 1:')
catName1 = input()
print('Enter the name of cat 2:')
catName2 = input()
print('Enter the name of cat 3:')
catName3 = input()
# And so on...
print('The cat names are:')
print(catName1 + ' ' + catName2 + ' ' + catName3)
```

Problems:

- Lots of repetitive code.
- If you have more cats, you need to add more variables and lines of code.

Example 2: Using a List (Better Way)

Here's how the program looks with a list:

```
catNames = []  # Start with an empty list
while True:
   print('Enter the name of cat ' + str(len(catNames) + 1) + ' (Or enter nothing to stop.):')
   name = input()  # Get a name from the user
   if name == '':  # If the user enters nothing, stop the loop
      break
   catNames = catNames + [name]  # Add the name to the list
print('The cat names are:')
for name in catNames:  # Go through each name in the list
   print(' ' + name)
```

Why is the List Better?

1. Flexible: You can store any number of names without creating new variables.
2. Less repetitive code: The same code works whether you have 2 cats or 200 cats.
3. Easy to manage: The list makes it simple to loop through all the names and do something with them (like print them).

Conclusion: Instead of using separate variables for each piece of similar data, use a list. It makes your code shorter, cleaner, and more flexible. Lists are like boxes that can hold as many items as you want, and you can easily add, remove, or process them

10. USING FOR LOOPS WITH LISTS

What is a for loop?

A for loop is a way to repeat code for each item in a list or a sequence.

For example:

```
for i in range(4):

   print(i)
```

This means:

- Start with i = 0 and print it.
- Then i = 1, print it.
- Then i = 2, print it.
- Finally, i = 3, print it.

The result:

```
0

1

2

3
```

Using a List Directly: You can also loop directly through a list instead of using range:

```
for i in [0, 1, 2, 3]:

   print(i)
```

This works the same way: it goes through each number in the list and prints it.

Using range(len(someList)): If you want to loop through a list with indexes, use range(len(someList)). For example:

```
supplies = ['pens', 'staplers', 'flamethrowers', 'binders']

for i in range(len(supplies)):

    print('Index ' + str(i) + ' in supplies is: ' + supplies[i])
```

This does two things:

1. Gives you the index (i).
2. Lets you use the value at that index (supplies[i]).

The output:

```
Index 0 in supplies is: pens

Index 1 in supplies is: staplers

Index 2 in supplies is: flamethrowers

Index 3 in supplies is: binders
```

Why use range(len())?

It's useful when you need both:

1. The index (position in the list).
2. The value (item at that position).

This method works no matter how many items are in the list

11. THE IN AND NOT IN OPERATORS

What Do in and not in Do?

- in checks if a value is present in a list (or another sequence).
- not in checks if a value is not present in a list.

They return True or False, depending on the result.

Examples:Using in:
'howdy' in ['hello', 'hi', 'howdy', 'heyas']

  - Check if 'howdy' is in the list.
  - Returns True because 'howdy' is in the list.

Using not in:
'cat' not in ['hello', 'hi', 'howdy', 'heyas']

  - Checks if 'cat' is not in the list.

- Returns True because 'cat' is not in the list.

Real-Life Example: Here's how you can use this in a program:

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']
print('Enter a pet name:')
name = input()  # User enters a name
if name not in myPets:
   print('I do not have a pet named ' + name)
else:
   print(name + ' is my pet.')
```

How it works:

1. The user types a name (e.g., Footfoot).
2. The program checks if the name is in the list myPets.

If it's not in the list, it says:I do not have a pet named Footfoot

If it's in the list, it says: Zophie is my pet.

Why Use This?

- It's a quick way to check if something exists in a list.
- Makes your code simple and easy to read.

Think of it as asking, "Is this item on my shopping li**st**

12. THE MULTIPLE ASSIGNMENT TRICK

Instead of assigning values to variables one by one, you can assign multiple variables at the same time using a list (or tuple).

Example: cat = ['fat', 'gray', 'loud']

# Traditional way (one by one):

size = cat[0]

color = cat[1]

disposition = cat[2]

# Shortcut using multiple assignment: size, color, disposition = cat

Now:

- size gets 'fat'
- color gets 'gray'
- disposition gets 'loud'

Important Rule: The number of variables must match the number of values in the list:

If there are too few or too many values, Python will give an error:

```
size, color, disposition, name = cat
```

```
# Error: not enough values to unpack (expected 4, got 3)
```

Why Use It?

- It’s faster and makes your code cleaner.
- Great when you know exactly how many values you’re working with

13. USING THE ENUMERATE() FUNCTION WITH LISTS

What Does enumerate() Do?

The enumerate() function gives you:

1. The index (position) of each item in a list.
2. The item itself.

This happens in a single step, so you don’t need to use range(len(list)).

Example:

```
supplies = ['pens', 'staplers', 'flamethrowers', 'binders']

for index, item in enumerate(supplies):

    print('Index ' + str(index) + ' in supplies is: ' + item)
```

Output:

```
Index 0 in supplies is: pens

Index 1 in supplies is: staplers

Index 2 in supplies is: flamethrowers

Index 3 in supplies is: binders
```

Why Use enumerate()?

- It’s cleaner and easier than using range(len(list)).
- You can access both the index and the value in the same loop.

It’s perfect when you need to work with both

14. USING THE RANDOM.CHOICE() AND RANDOM.SHUFFLE() FUNCTIONS WITH LISTS

random.choice()

- What it does: Picks one random item from a list.

Example:

```
import random
pets = ['Dog', 'Cat', 'Moose']
print(random.choice(pets))  # Output might be 'Dog' or 'Cat' or 'Moose'
```

- When to use: When you want to pick something randomly (e.g., selecting a winner).

random.shuffle()

- What it does: Randomly rearranges the items in a list (changes the list directly).

Example:

```
import random
people = ['Alice', 'Bob', 'Carol', 'David']
random.shuffle(people)
print(people)  # Output might be ['Carol', 'David', 'Alice', 'Bob']
```

- When to use: When you want to mix up the order of items (e.g., shuffling a deck of cards).

Key Points:

- random.choice() gives one random item from the list.
- random.shuffle() rearranges the whole list randomly.

These functions are handy for adding randomness to your programs.

14. AUGMENTED ASSIGNMENT OPERATORS

Here's a simple explanation of augmented assignment (+=):

What is +=?

- += is a shortcut for adding a value to a variable and assigning the result back to the variable.

Example:

```
spam = 42
spam += 1  # Same as spam = spam + 1
print(spam)  # Output: 43
```

- Instead of writing spam = spam + 1, you use spam += 1.

Why use it? It simplifies code and makes it more concise

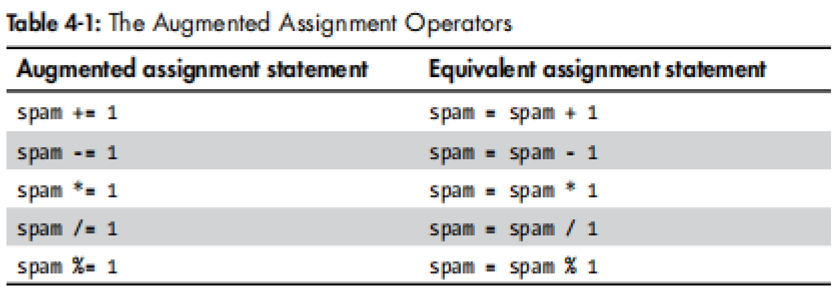


Table 4-1: The Augmented Assignment Operators

Augmented assignment statement	Equivalent assignment statement
`spam += 1`	`spam = spam + 1`
`spam -= 1`	`spam = spam - 1`
`spam *= 1`	`spam = spam * 1`
`spam /= 1`	`spam = spam / 1`
`spam %= 1`	`spam = spam % 1`

+= Operator: Used to concatenate strings or lists.

Example:

```
spam = 'Hello,'
spam += ' world!'
print(spam)  # Output: 'Hello world!'
```

*= Operator: Used to replicate strings or lists.

Example:

```
bacon = ['Zophie']
bacon *= 3
print(bacon)  # Output: ['Zophie', 'Zophie', 'Zophie']
```

Why Use These? They make concatenation and replication easier and more concise.

15. METHODS

What is the index() method?

- The index() method is used to find the position of a value in a list.
- It returns the first occurrence of the value.
- If the value isn't in the list, it raises a ValueError.

Example:

```
spam = ['hello', 'hi', 'howdy', 'heyas']

print(spam.index('hello'))  # Output: 0

print(spam.index('heyas'))  # Output: 3

# If the value is not in the list:

# print(spam.index('howdy howdy howdy'))  # Raises ValueError
```

Duplicates: If a value appears more than once, index() returns the first occurrence.

```
spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']

print(spam.index('Pooka'))  # Output: 1
```

### Why Use `index()`?

- It helps locate where a value is within a list

16. ADDING VALUES TO LISTS WITH THE APPEND() AND INSERT() METHODS

append() Method: Adds a value to the end of the list.

Example : spam = ['cat', 'dog', 'bat']

spam.append('moose')

print(spam)

# Output: ['cat', 'dog', 'bat', 'moose']

insert() Method: Inserts a value at a specific index in the list.

Example: spam = ['cat', 'dog', 'bat']

spam.insert(1, 'chicken')

print(spam)

# Output: ['cat', 'chicken', 'dog', 'bat']

Important Notes:

- Neither append() nor insert() return the modified list; they modify the list **in place**.
- **Cannot** use append() or insert() with other data types like strings or integers.

Example with Error:

eggs = 'hello'

# eggs.append('world') # Error: 'str' object has no attribute 'append'

bacon = 42

# bacon.insert(1, 'world') # Error: 'int' object has no attribute 'insert'

17. REMOVING VALUES FROM LISTS WITH THE REMOVE() METHOD

remove() Method:

Removes the first occurrence of a specified value from the list.

Example: spam = ['cat', 'bat', 'rat', 'elephant']

spam.remove('bat')

```
print(spam)  # Output: ['cat', 'rat', 'elephant']
```

Important Notes: If the value is not in the list, a ValueError will be raised.

```
spam.remove('chicken')  # Raises ValueError
```

If a value appears multiple times, only the first instance is removed.

```
spam = ['cat', 'bat', 'rat', 'cat', 'hat', 'cat']

spam.remove('cat')

print(spam)  # Output: ['bat', 'rat', 'cat', 'hat', 'cat']
```

Comparison with del Statement:

- Use del when you know the index of the value you want to remove.
- Use remove() when you know the value you want to remove.

The remove() method is simple and useful when you want to eliminate a known value from a list.

18. SORTING THE VALUES IN A LIST WITH THE SORT() METHOD

sort() Method: Purpose: To sort lists of numbers or strings in ascending order.

Example:

```
spam = [2, 5, 3.14, 1, -7]

spam.sort()

print(spam)  # Output: [-7, 1, 2, 3.14, 5]
```

1. In-Place Sorting:

sort() sorts the list directly and doesn't return a new list. So, don't write spam = spam.sort() because it will return None.

Example:

```
spam = [3, 1, 2]

spam.sort()

print(spam)  # Output: [1, 2, 3]
```

2. Sorting Order: By default, sort() sorts in ascending order.

3. Sorting in Reverse: Use reverse=True to sort in descending order.

```
spam.sort(reverse=True)
```

print(spam) # Output: [5, 3.14, 2, 1, -7]

4. Sorting with Different Types: You can't sort a list with mixed types (e.g., numbers and strings) because Python can't compare them.

Example:

spam = [1, 3, 2, 'Alice', 'Bob']

spam.sort() # Raises TypeError

6. Sorting Strings: sort() uses ASCIIbetical order, so uppercase letters are sorted before lowercase letters.

Example:
spam = ['Alice', 'bob', 'Zebra']

spam.sort()

print(spam) # Output: ['Alice', 'Zebra', 'bob']

7. Case Insensitive Sorting: To sort strings ignoring the case, use key=str.lower.

spam = ['a', 'Z', 'B', 'z']

spam.sort(key=str.lower)

print(spam) # Output: ['a', 'B', 'z', 'Z']

19. REVERSING THE VALUES IN A LIST WITH THE REVERSE() METHOD

The reverse() method is used to reverse the order of elements in a list. Here's a simple explanation with an example:

reverse() Method:

Purpose: To reverse the order of elements in a list.

Example:

spam = ['cat', 'dog', 'moose']

spam.reverse()

print(spam) # Output: ['moose', 'dog', 'cat']

How it Works:

The reverse() method reverses the list in place. It does not create a new list, but modifies the original list directly.

Example:

```
spam = ['cat', 'dog', 'moose']

spam.reverse()  # The list order is reversed

print(spam)  # Output: ['moose', 'dog', 'cat']
```

Points to Note: In-Place Operation: The list is modified directly, so spam = spam.reverse() won't work. You need to write spam.reverse() to reverse the list. This method is useful when you need the elements of a list in the opposite order quickly

20. EXAMPLE PROGRAM: MAGIC 8 BALL WITH A LIST

The Magic 8 Ball program uses a list of possible answers, which makes the code cleaner and easier to manage. Here's a simple explanation:

Program Overview:

1. List of Messages:

```
        messages = ['It is certain',
              'Yes definitely',
              'Reply hazy try again',
              'Ask again later',
              'My reply is no',
              'Outlook not so good',
              'Very doubtful']
```

2. Random Selection:A random message is selected from the list using:

```
import random

random.choice(messages)
```

Example:

```
import random
messages = ['It is certain', 'Yes definitely', 'Reply hazy try again', 'Ask again later', 'My reply is no',
'Outlook not so good', 'Very doubtful']
print(messages[random.randint(0, len(messages) - 1)])
```

3. Flexibility: Adding or removing messages is easier because you just modify the list without needing to change the code logic.

This method simplifies the Magic 8 Ball program by reducing repetitive `if-elif` statements into a concise list

21. SEQUENCE DATA TYPES

Python provides several types of sequence data, including lists, strings, and tuples. Here's a simple explanation of how these work:

1. Lists vs Strings:

- Lists are mutable: You can change, add, or remove items from a list.
- Strings are immutable: Once created, they cannot be changed.

2. Indexing & Slicing:

Both lists and strings allow indexing and slicing.

Indexing: Accessing individual elements by position.

name = 'Zophie'

name[0] # Output: 'Z'

name[-2] # Output: 'i'

Slicing: Extracting a portion of the sequence.

name[0:4] # Output: 'Zoph'

Using in and not in operators:

'Zo' in name # Output: True

'z' in name   # Output: False

3. For Loops with Sequence Data:

You can iterate through both lists and strings with a for loop:

```
for i in name:
    print('* * * ' + i + ' * * *')
```

Output:

- Z * * *
- o * * *
- p * * *
- h * * *
- i * * *
- e * * *

4. Mutable vs Immutable:

Mutable: Can be changed.

```
my_list = [1, 2, 3]

my_list[0] = 4  # Changes the first element

# Mutable list

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

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

# Modifying the list

fruits[0] = 'blueberry'

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

# Adding a new element

fruits.append('orange')

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

# Removing an element

fruits.remove('banana')

print(fruits)  # Output: ['blueberry', 'cherry', 'orange']
```

Immutable: Cannot be changed directly.

```
name = 'Zophie'

name[0] = 'T'  # Raises TypeError
```

5. Changing an Immutable String:

To "mutate" a string, you create a new string:

```
name = 'Zophie a cat'

newName = name[0:7] + 'the' + name[8:]

# newName becomes 'Zophie the cat'
```

Original string name remains unchanged.

6. Replacing a List:

When you assign a new list to a variable, it replaces the old list:

eggs = [1, 2, 3]

eggs = [4, 5, 6] # Replaces [1, 2, 3] with [4, 5, 6]

However, to modify the existing list, you would use methods like del or append:

eggs = [1, 2, 3]

del eggs[0] # Removes the first element

eggs.append(4) # Adds 4 to the end

Final eggs: [2, 3, 4]

This covers the main differences between mutable and immutable types, as well as how indexing, slicing, and loops work with sequences

22. TUPLE DATA TYPES

A **tuple** is like a list, but with some important differences:

Syntax: Tuple is created with parentheses () instead of square brackets [].

- Example: eggs = ('hello', 42, 0.5)

Immutability: Tuples are **immutable**, meaning you **cannot** change, add, or remove values once they are created

- eggs = ('hello', 42, 0.5)
- eggs[1] = 99 # This will raise a TypeError

Single Value Tuple:

- If a tuple has only one value, you need to add a comma at the end.
  - Example
  - type(('hello',)) # This is a tuple
  - type(('hello'))   # This is a string

Benefits: Since tuples are immutable, they are faster than lists and can be used safely when you don't want values to change.

Converting:You can convert a tuple to a list (or vice versa) using functions like `list()` and `tuple()`.

- Example:
- tuple(['cat', 'dog', 5]) # Converts list to tuple
- list(('cat', 'dog', 5))   # Converts tuple to list

Example

Creating a tuple:

eggs = ('apple', 'banana', 'cherry')

print(eggs[0]) # 'apple'

Trying to modify a tuple:

eggs[1] = 'orange' # TypeError: 'tuple' object does not support item assignment

CHAPTER 5

1. The Dictionary Data Type

A dictionary in Python is a collection of key-value pairs, where each key is unique and maps to a specific value.

- Example: myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}. Here, 'size', 'color', and 'disposition' are keys, and 'fat', 'gray', and 'loud' are their corresponding values.
- Unlike lists, dictionaries are unordered, meaning the order of key-value pairs doesn't matter.
- You access values using keys: myCat['size'] returns 'fat'.
- If a key doesn't exist, you get a KeyError.
- Dictionaries are useful for organizing data in flexible ways, such as storing friends' birthdays with their names as keys.
- Example: birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12'} stores birthdays, which can be updated easily.
- Dictionaries allow you to add new key-value pairs and check if a key exists using in.

A dictionary in Python is like a box where you can store pairs of keys and values.

- Keys are like labels, and values are the information or data that go with those labels.
- Unlike lists, which are ordered, dictionaries aren't in a specific order.

For example:

- {'name': 'Alice', 'age': 30, 'city': 'Wonderland'}

Here:

- 'name' is a key, and 'Alice' is its value.
- 'age' is a key, and 30 is its value.

You can access the value using the key:

- my_dict['name'] will give 'Alice'.

If a key doesn't exist, like trying my_dict['height'], it will return a KeyError because there's no value for that key.

In short, dictionaries are a flexible way to organize and look up information quickly using keys

1. **keys(), values(), and items()**:
   - keys(): Returns all the keys in a dictionary.
   - values(): Returns all the values in a dictionary.

   - items(): Returns both keys and values as pairs (tuples).
2. **Ordered Dictionaries**:
   - Starting from Python 3.7, dictionaries keep track of the order in which key-value pairs are added. In older versions (e.g., Python 3.5), this order is not guaranteed.
3. **Using in and not in**:
   - key in dict.keys(): Checks if a key exists.
   - value in dict.values(): Checks if a value exists.
4. **get() Method**:
   - Safely get a value from a dictionary. If the key doesn't exist, a default value is returned.
5. **setdefault() Method**:
   - Sets a default value for a key if it doesn't already exist.
6. **Example Program**:
   - setdefault() can help count occurrences of characters in a string without raising errors when keys don't exist.

In summary:

- keys(), values(), and items() are methods for accessing parts of a dictionary.
- Dictionaries are ordered in newer versions, keeping track of added order.
- in and get() are useful for checking/extracting values safely.
- setdefault() ensures a key exists with a default value.

2. PRETTY PRINTING

- Pretty Printing:
  - The pprint module helps make dictionaries look nicer and easier to read by formatting them in a clean, organized way.
- pprint.pprint():
  - This function prints a dictionary in a more readable format, especially useful when dealing with large or complex dictionaries.
- pprint.pformat():
  - Instead of printing, pprint.pformat() returns a formatted string of the dictionary. You can use this to store or display the formatted dictionary as text.

Example Program:

Importing pprint:

```
import pprint
```

Creating a Dictionary:

```
message = 'It was a bright cold day in April, and the clocks were striking thirteen.'

count = {}

for character in message:

    count.setdefault(character, 0)

    count[character] += 1
```

Pretty Printing:

pprint.pprint(count) prints the dictionary in a neat way.

pprint.pformat(count) returns a string of the formatted dictionary.

Output:

- Using pprint.pprint(count):

```
{' ': 13,
',': 1,
'.': 1,
'A': 1,
'I': 1,
--snip--
't': 6,
'w': 2,
'y': 1}
```

Using pprint.pformat(count) returns the same formatted content as a string.

In short:

- pprint.pprint() displays a nicely formatted dictionary.
- pprint.pformat() returns a string of the formatted dictionary for use in other places.

Conclusion : Pretty printing is the process of formatting data (like dictionaries) in a way that makes it easier to read and understand. It organizes the data into a cleaner, more structured view, especially when dealing with large or complex data structures.

Imagine you have a messy list of numbers, like this:

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

Now, if you simply print it using print():

```
print(numbers)
```

The output will look like this:

```
[1, 5, 3, 7, 2, 8, 4]
```

But if you want to see it in a cleaner, more organized way, you can use **pretty printing** with pprint:

```
import pprint

pprint.pprint(numbers)
```

The output will look like this:

```
[1, 2, 3, 4, 5, 7, 8]
```

Pretty printing helps you visualize the data in a more readable format, especially when working with larger or more complex data structures like dictionaries or nested lists.

3.USING DATA STRUCTURES TO MODEL REAL-WORLD THINGS

Chessboard and Coordinates:

- A chessboard is made up of squares.
- Each square has a unique address made up of a letter (A to H) and a number (1 to 8). Example: A1, B2, C3, etc.

Using Data Structures:

- To keep track of a chess game, we need a way to store and represent the board's state and moves.
- A simple data structure like a **list of lists** can represent the chessboard:
  - Each sublist represents a row of the board.
  - Each item in the sublist represents a square (or space) on the board.

Example:

- A simple chessboard could be represented as a 2D list:

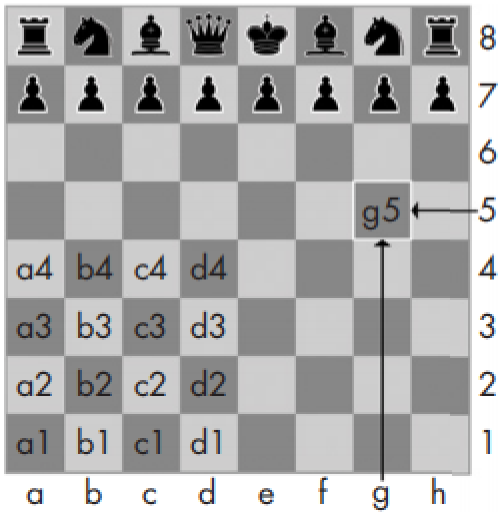


Figure 5-1: The coordinates of a chessboard in algebraic chess notation

Scenario: Managing a School Library

Imagine you're managing a school library, and you need to keep track of books, their authors, and the number of copies available. Using data structures like dictionaries and lists makes this easier.

Example Using Data Structures:

- Books are represented by a dictionary where:
  - The key is the book title.
  - The value is another dictionary containing the author and the number of copies available.

```
library = {

   'To Kill a Mockingbird': {'author': 'Harper Lee', 'copies': 3},

   '1984': {'author': 'George Orwell', 'copies': 5},

   'Moby Dick': {'author': 'Herman Melville', 'copies': 2}

}
```

Each book has a title as a key, and the value is a dictionary with the author and how many copies are available.

Functions to Manage the Library

You can create functions to help manage the library:

Adding a new book:

```
def addBook(library, title, author, copies):

   library[title] = {'author': author, 'copies': copies}
```

Checking the number of copies:

```
def checkCopies(library, title):

   return library.get(title, {}).get('copies', 0)
```

Calculating total copies for all books:

```
def totalCopies(library):

   total = 0

   for book in library.values():

      total += book['copies']

   return total
```

Example Usage:

Adding a new book:

addBook(library, 'The Great Gatsby', 'F. Scott Fitzgerald', 4)

Checking how many copies of '1984' are available:

print(checkCopies(library, '1984')) # Output: 5

Calculating total copies:

print(totalCopies(library)) # Output: 14

Why This Works:

- Nested dictionaries are used to represent complex data (books with authors and copies).
- Functions help manage and manipulate this data efficiently, even if the library grows to include thousands of books.

In this way, data structures simplify managing real-world systems like libraries