Common data structures in Python

Lists

- Ordered, mutable (can be changed after creation)
- Allow duplicate elements
- Can store mixed data types
- Commonly used for collections of items where order matters and data type may not be same for each elements

Common Methods:

`append()`, `extend()`, `insert()`, `remove()`, `pop()`, `sort()`, `reverse()`, `index()`, `count()`

Common Operators:

`+`, `*`, `in`, `not in`, `len()`, slicing (`[:]`)

In [54]:
```
# Creating lists
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
mixed = [1, "apple", 3.14, True]
```

In [55]:
```
# Accessing elements
print(numbers[0])      # 0 index refers to first element
print(numbers[-1])     # -1 index refers to the last element
```

```
1
10
```

In [56]:
```
len(numbers)
```

Out[56]: 10

Slicing lists

Python is a zero-indexed language. It means that in data structures, the first element is given the index 0, the second element is given the index 1, and so on.

- When slicing a part of a list in python, the following syntax is used: `lst1[a, b]`
- However, the element at the b^{th} index is not returned
- For example, `lst1[0, 5]` will return the elements upto index 4 (the fifth element) and will not include the element at index 5 (the sixth element)

In [57]:
```
numbers[1:3]    # returns elements at index 1 and 2 (the second and the third el
```

Out[57]: [2, 3]

In [58]:
```
# Can also mention a step size in the slicing operator
numbers[0:7:2]
```

Out[58]:
```
[1, 3, 5, 7]
```

In [59]:
```
# Can slice starting from the end
numbers[-1:-9:-2]
```

Out[59]:
```
[10, 8, 6, 4]
```

Adding-removing items

In [60]:
```
numbers.append(11)
numbers
```

Out[60]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

In [61]:
```
numbers.insert(2, 200)    # inserting 200 at index 2
numbers
```

Out[61]:
```
[1, 2, 200, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

In [62]:
```
numbers.append(200)    # adding another 200 at the end of the list
numbers
```

Out[62]:
```
[1, 2, 200, 3, 4, 5, 6, 7, 8, 9, 10, 11, 200]
```

In [63]:
```
numbers.count(200)
```

Out[63]:
```
2
```

In [64]:
```
numbers.remove(200)    # this removes the first occurence of 200 from the list
numbers
```

Out[64]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 200]
```

The `list.pop()` method

- When this method is called without an argument, it removes the last item from the list and returns that item to the caller
- When called with a index as the argument, it removes the item at the provided index and returns that item

In [65]:
```
popped_item = numbers.pop()
popped_item
```

Out[65]:
```
200
```

In [66]:
```
numbers.pop(5)    # removes and returns the item at index 5
```

Out[66]:
```
6
```

Reversing, sorting lists

In [67]:
```
numbers.reverse()
numbers
```

Out[67]: [11, 10, 9, 8, 7, 5, 4, 3, 2, 1]

In [68]:
```
list(reversed(numbers))
```

Out[68]: [1, 2, 3, 4, 5, 7, 8, 9, 10, 11]

In [69]:
```
numbers
```

Out[69]: [11, 10, 9, 8, 7, 5, 4, 3, 2, 1]

In [70]:
```
numbers.sort()
numbers
```

Out[70]: [1, 2, 3, 4, 5, 7, 8, 9, 10, 11]

In [71]:
```
numbers.reverse()
numbers
```

Out[71]: [11, 10, 9, 8, 7, 5, 4, 3, 2, 1]

In [72]:
```
sorted(numbers)
```

Out[72]: [1, 2, 3, 4, 5, 7, 8, 9, 10, 11]

In [73]:
```
numbers
```

Out[73]: [11, 10, 9, 8, 7, 5, 4, 3, 2, 1]

Concatenating and repeating lists items

In [74]:
```
numbers + [97, 98, 99]
```

Out[74]: [11, 10, 9, 8, 7, 5, 4, 3, 2, 1, 97, 98, 99]

In [75]:
```
numbers * 2
```

Out[75]: [11, 10, 9, 8, 7, 5, 4, 3, 2, 1, 11, 10, 9, 8, 7, 5, 4, 3, 2, 1]

The `list.copy()` method

When assigning a list to another variable name, the new variable only gets a reference to the old list, it does not create a new copy. Modifying the new variable will also modify the old one.

In [76]:
```
n2 = numbers
n2.append(1000)
print(n2)
print(numbers)
```

```
[11, 10, 9, 8, 7, 5, 4, 3, 2, 1, 1000]
[11, 10, 9, 8, 7, 5, 4, 3, 2, 1, 1000]
```

In [77]:
```
n2 is numbers    # true because n2 and numbers refer to the same object in memor
```

Out[77]: True

In order to create a copy a list, use the `list.copy()` method or use the slice operator without any start and end index: `list[:]`

In [78]:
```
n3 = numbers.copy()
```

In [79]:
```
n3 is numbers    # false because n3 and numbers do not refer to the same objects
```

Out[79]: False

In [80]:
```
n3 == numbers
```

Out[80]: True

In [81]:
```
n3.append(1000)
print(n3)
print(numbers)
```

```
[11, 10, 9, 8, 7, 5, 4, 3, 2, 1, 1000, 1000]
[11, 10, 9, 8, 7, 5, 4, 3, 2, 1, 1000]
```

List comprehension

In [82]:
```
lst2=[x**2 for x in numbers]
lst2
```

Out[82]: [121, 100, 81, 64, 49, 25, 16, 9, 4, 1, 1000000]

Strings

- A string is an **immutable sequence** of Unicode characters.
- Defined using single (`'`), double (`"`) or triple quotes (`'''` or `"""`).
- Commonly used for text data, i.e., names, messages, file paths, etc.
- Being immutable means once created, the string's content cannot be changed.

Common Operations

- Indexing, slicing, concatenation, repetition
- Iteration and membership tests
- String methods for formatting, searching, and modification

Common Functions:

`len()` , `str()` , `format()` , `ord()` , `chr()`

Common Methods:

Category	Methods	Description

Category	Methods	Description
Case Conversion	`upper()`, `lower()`, `capitalize()`, `title()`, `swapcase()`	Change letter cases
Searching	`find()`, `rfind()`, `index()`, `count()`	Locate substrings
Testing	`startswith()`, `endswith()`, `isalpha()`, `isdigit()`, `isalnum()`, `isspace()`	Check properties
Modification	`replace()`, `strip()`, `lstrip()`, `rstrip()`	Clean or modify strings
Splitting & Joining	`split()`, `rsplit()`, `splitlines()`, `join()`	Split into lists or join lists into strings
Formatting	`format()`, f-strings, `%` formatting	Insert variables or values into strings

Operators:

`+` (concatenation), `*` (repetition), `in` / `not in` (membership), comparison operators (`==`, `<`, etc.), slicing (`[:]`)

In [83]:
```
# Creating strings
s1 = "Hello"
s2 = 'World'
multi = """This is
a multi-line string."""
```

In [84]:
```
print(s1, s2)
print(multi)
```

```
Hello World
This is
a multi-line string.
```

In [85]:
```
# Indexing and slicing
print(s1[0])       # First character
print(s1[-1])      # Last character
print(s1[1:4])     # Substring from index 1 to 3
```

```
H
o
ell
```

In [86]:
```
# Concatenation and repetition
print(s1 + " " + s2)
print(s1 * 3)
```

```
Hello World
HelloHelloHello
```

In [87]:
```
# Membership test
print("H" in s1)
print("x" not in s1)# Membership test
print("H" in s1)
print("x" not in s1)
```

```
True
True
True
True
```

In [88]:
```
# Common methods
text = "  python programming  "
print(text.upper())        # Convert to uppercase
print(text.strip())        # Remove leading/trailing spaces
print(text.replace("python", "java"))
print(text.title())        # Capitalize each word
```

```
  PYTHON PROGRAMMING
python programming
  java programming
  Python Programming
```

In [89]:
```
# Splitting and joining
words = text.strip().split()
print(words)
joined = "-".join(words)
print(joined)
```

```
['python', 'programming']
python-programming
```

In [90]:
```
# Searching
print(text.find("python"))    # Returns index of substring
print(text.count("p"))        # Count occurrences
```

```
2
2
```

In [91]:
```
# Validation checks
print("123".isdigit())
print("abc".isalpha())
print("abc123".isalnum())
```

```
True
True
True
```

Formatting strings

Can use the `.format()` method or `fstring`

In [92]:
```
# Formatting strings
name = "Alice"
age = 25
print("Name: {}, Age: {}".format(name, age))    # Using format()
print(f"Name: {name}, Age: {age}")    # Using f-string
```

```
Name: Alice, Age: 25
Name: Alice, Age: 25
```

Tuples

- Ordered, **immutable** (cannot be modified after creation)
- Allow duplicates

- Often used for fixed collections (coordinates, database records, etc.)

Common Methods:

`count()`, `index()`

Common Operators:

`+`, `*`, `in`, `len()`, unpacking

In [93]:
```
# Creating tuples
coords = (10, 20)
single = (5,)    # single-element tuple needs a comma
```

In [94]:
```
coords[0]
```

Out[94]: 10

In [95]:
```
# Tuple unpacking
x, y = coords
print(x, y)
```

```
10 20
```

In [96]:
```
# Concatenating
coords + (30, 40)
```

Out[96]: (10, 20, 30, 40)

Sets

- Unordered, **mutable**, and store **unique** elements
- Useful for membership tests, removing duplicates, and set operations

Common Methods:

`add()`, `remove()`, `discard()`, `update()`, `union()`, `intersection()`, `difference()`, `symmetric_difference()`

Common Operators:

`|` (union), `&` (intersection), `-` (difference), `^` (symmetric difference)

In [97]:
```
# Creating sets
nums = {1, 2, 3, 3}
print(nums)   # duplicates removed automatically
```

```
{1, 2, 3}
```

In [98]:
```
# Adding and removing
nums.add(4)
nums.discard(2)
print(nums)
```

```
{1, 3, 4}
```

In [99]:
```python
# Set operations
a = {1, 2, 3}
b = {3, 4, 5}
print(a | b)   # union
print(a & b)   # intersection
print(a - b)   # difference
```

```
{1, 2, 3, 4, 5}
{3}
{1, 2}
```

Dictionary

- Stores data as **key-value pairs**
- Keys are unique and immutable (strings, numbers, tuples)
- Values can be any data type

Common Methods:

`get()`, `keys()`, `values()`, `items()`, `update()`, `pop()`, `clear()`

Common Operators:

`in`, `not in`, `len()`

In [100…
```python
# Creating a dictionary
student = {"name": "Alice", "age": 20, "grade": "A"}
student
```

Out[100…
```
{'name': 'Alice', 'age': 20, 'grade': 'A'}
```

In [101…
```python
# Accessing values
print(student["name"])
print(student.get("grade"))
```

```
Alice
A
```

In [102…
```python
# Adding and modifying
student["age"] = 21
student["city"] = "Dhaka"
```

In [103…
```python
# Removing
student.pop("grade")
```

Out[103…
```
'A'
```

In [104…
```python
# Iterating
for key in student:
    print(key, ":", student[key])
```

```
name : Alice
age : 21
city : Dhaka
```

The `.items()` method

Reuturns a list of tuples where each tuple contains a key (at index 0) and its corresponding value (at index 1).

In [105…
```
for key, value in student.items():
    print(key, ":", value)
```

```
name : Alice
age : 21
city : Dhaka
```

The `.get()` method

This method works like the `[key]` operator, but one can mention a default value to return when `.get()` is run with a key that does not exist in the dictionary

In [106…
```
# Example: counting word frequency
sentence = "apple banana apple orange apple"
word_count = {}
for word in sentence.split():
    word_count[word] = word_count.get(word, 0) + 1

print(word_count)
```

```
{'apple': 3, 'banana': 1, 'orange': 1}
```

The `collections` module

The `collections` module provides specialized container data types that extend Python's built-in data structures for more efficient or expressive handling of data.

Data Structure	Description	Key Features / Use Cases
`deque`	Double-ended queue supporting fast appends and pops from both ends	Ideal for implementing queues and stacks efficiently (`append()` , `appendleft()` , `pop()` , `popleft()`)
`Counter`	Subclass of `dict` for counting hashable objects	Automatically counts element frequencies; useful for word counts, histograms, etc.
`defaultdict`	Like a `dict` but provides a default value for missing keys	Simplifies handling of missing keys (e.g., `defaultdict(list)` for grouping items)
`namedtuple`	Factory function for creating tuple subclasses with named fields	Makes tuples more readable by accessing elements by name instead of index