Python Notes

Condensed Notes

May 4, 2025

Contents

1 Lists **4**

1.1 Basics of Lists . . . 4
- 1.1.1 Creating Lists . . . 4
- 1.1.2 List Syntax and Indexing . . . 4
- 1.1.3 List Data Types (Heterogeneous Elements) . . . 5
- 1.1.4 Nested Lists . . . 5

1.2 Accessing Elements . . . 5
- 1.2.1 Indexing (Positive and Negative) . . . 5
- 1.2.2 Slicing . . . 5
- 1.2.3 Iterating Through Lists . . . 5

1.3 Modifying Lists . . . 6
- 1.3.1 Changing Elements by Index . . . 6
- 1.3.2 Adding Elements . . . 6
- 1.3.3 Removing Elements . . . 6

1.4 List Operations . . . 7
- 1.4.1 Concatenation . . . 7
- 1.4.2 Repetition . . . 7
- 1.4.3 Membership Testing . . . 7
- 1.4.4 Length of List . . . 7

1.5 List Methods . . . 8
- 1.5.1 append(), extend(), insert() . . . 8
- 1.5.2 remove(), pop(), clear() . . . 8
- 1.5.3 index(), count() . . . 8
- 1.5.4 sort(), reverse(), copy() . . . 8

1.6 List Comprehensions . . . 8
- 1.6.1 Basic List Comprehensions . . . 8
- 1.6.2 Conditional List Comprehensions . . . 9
- 1.6.3 Nested Comprehensions . . . 9

1.7 Iterating and Looping . . . 9
- 1.7.1 for Loops . . . 9
- 1.7.2 enumerate() . . . 9
- 1.7.3 zip() with Multiple Lists . . . 9

1.8 Common Use Cases . . . 9
- 1.8.1 Filtering Items . . . 9
- 1.8.2 Mapping/Transformation . . . 10
- 1.8.3 Flattening Nested Lists . . . 10
- 1.8.4 Finding min, max, sum . . . 10

1.9 Copying and Cloning Lists . . . 10
- 1.9.1 Shallow Copy vs. Deep Copy . . . 10

1.9.2 Methods to Copy Lists 11
1.10 Lists vs Other Data Structures 11
1.10.1 Lists vs Tuples 11
1.10.2 Lists vs Sets 11
1.10.3 When to Use Lists 11
1.11 Key Takeaways 11

Chapter 1

Lists

1.1 Basics of Lists

1.1.1 Creating Lists

Lists in Python are versatile data structures that store ordered collections of items. They are mutable, allowing modifications after creation, and can contain duplicates. Lists are defined using square brackets `[]`, with elements separated by commas.

```
# Basic list creation
my_list = [1, 2, 3, 4, 5]
print(my_list)  # Output: [1, 2, 3, 4, 5]

# Using list() constructor
my_list = list((1, 2, 3, 4, 5))
print(my_list)  # Output: [1, 2, 3, 4, 5]

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

Key Takeaway: Lists are dynamic arrays that automatically adjust their size as elements are added or removed.

1.1.2 List Syntax and Indexing

Lists are zero-indexed, meaning the first element is at index 0. Negative indexing allows access from the end, with -1 referring to the last element.

```
my_list = [10, 20, 30, 40, 50]
print(my_list[0])   # Output: 10
print(my_list[2])   # Output: 30
print(my_list[-1])  # Output: 50
print(my_list[-2])  # Output: 40
```

Key Takeaway: Indexing is fundamental for accessing specific elements efficiently.

1.1.3 List Data Types (Heterogeneous Elements)

Lists can store elements of different data types, including integers, strings, floats, booleans, and other lists, making them highly flexible.

```
mixed_list = [1, "hello", 3.14, True, [10, 20]]
print(mixed_list)  # Output: [1, 'hello', 3.14, True, [10, 20]]
```

Key Takeaway: The ability to store heterogeneous elements makes lists suitable for diverse applications.

1.1.4 Nested Lists

Nested lists are lists within lists, often used to represent matrices or hierarchical data. Elements are accessed using multiple indices.

```
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(nested_list)        # Output: [[1, 2, 3], [4, 5, 6], [7, 8,
   9]]
print(nested_list[1][2])  # Output: 6
```

Key Takeaway: Nested lists enable complex data structures but require careful indexing.

1.2 Accessing Elements

1.2.1 Indexing (Positive and Negative)

Positive indexing starts at 0, while negative indexing starts at -1 from the end, providing flexible access to list elements.

```
my_list = [10, 20, 30, 40, 50]
print(my_list[0])   # Output: 10
print(my_list[-1])  # Output: 50
```

1.2.2 Slicing

Slicing extracts a portion of a list using the syntax `list[start:stop:step]`. The `start` index is inclusive, `stop` is exclusive, and `step` defines the increment.

```
my_list = [10, 20, 30, 40, 50, 60]
print(my_list[1:4])   # Output: [20, 30, 40]
print(my_list[::2])   # Output: [10, 30, 50]
print(my_list[::-1])  # Output: [60, 50, 40, 30, 20, 10]
```

Key Takeaway: Slicing is powerful for extracting and manipulating sublists.

1.2.3 Iterating Through Lists

Lists can be iterated using `for` or `while` loops. `for` loops are more Pythonic and concise.

```
# Using for loop
for item in [1, 2, 3]:
    print(item)  # Output: 1, 2, 3

# Using while loop
i = 0
my_list = [1, 2, 3]
while i < len(my_list):
    print(my_list[i])
    i += 1
```

Key Takeaway: Choose `for` loops for simplicity unless index-based iteration is required.

1.3 Modifying Lists

1.3.1 Changing Elements by Index

List elements can be modified by assigning a new value to a specific index.

```
my_list = [10, 20, 30, 40]
my_list[1] = 25
print(my_list)  # Output: [10, 25, 30, 40]
```

1.3.2 Adding Elements

Lists support several methods to add elements:

- `append(x)`: Adds `x` to the end.
- `insert(i, x)`: Inserts `x` at index `i`.
- `extend(iterable)`: Adds all elements from `iterable` to the end.

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)  # Output: [1, 2, 3, 4]

my_list.insert(0, 0)
print(my_list)  # Output: [0, 1, 2, 3, 4]

my_list.extend([5, 6])
print(my_list)  # Output: [0, 1, 2, 3, 4, 5, 6]
```

1.3.3 Removing Elements

Lists provide multiple ways to remove elements:

- `remove(x)`: Removes the first occurrence of `x`.
- `pop([i])`: Removes and returns the element at index `i` (default: last).

- `del list[i]`: Deletes the element at index `i` or a slice.
- `clear()`: Removes all elements.

```
my_list = [1, 2, 3, 2]
my_list.remove(2)
print(my_list)  # Output: [1, 3, 2]

removed = my_list.pop(1)
print(removed, my_list)  # Output: 3 [1, 2]

del my_list[0]
print(my_list)  # Output: [2]

my_list.clear()
print(my_list)  # Output: []
```

1.4 List Operations

1.4.1 Concatenation

The + operator combines two lists into a new list.

```
list1 = [1, 2]
list2 = [3, 4]
combined = list1 + list2
print(combined)  # Output: [1, 2, 3, 4]
```

1.4.2 Repetition

The * operator repeats a list a specified number of times.

```
repeated = [1, 2] * 3
print(repeated)  # Output: [1, 2, 1, 2, 1, 2]
```

1.4.3 Membership Testing

The `in` and `not in` operators check if an element exists in a list.

```
my_list = [1, 2, 3]
print(2 in my_list)      # Output: True
print(4 not in my_list)  # Output: True
```

1.4.4 Length of List

The `len()` function returns the number of elements in a list.

```
my_list = [1, 2, 3]
print(len(my_list))  # Output: 3
```

1.5 List Methods

1.5.1 append(), extend(), insert()

These methods, covered in Section 3.2, facilitate adding elements to lists.

1.5.2 remove(), pop(), clear()

These methods, covered in Section 3.3, handle element removal.

1.5.3 index(), count()

- `index(x)`: Returns the index of the first occurrence of `x`.
- `count(x)`: Returns the number of occurrences of `x`.

```
my_list = [1, 2, 3, 2]
print(my_list.index(2))  # Output: 1
print(my_list.count(2))  # Output: 2
```

1.5.4 sort(), reverse(), copy()

- `sort()`: Sorts the list in place.
- `reverse()`: Reverses the list in place.
- `copy()`: Returns a shallow copy of the list.

```
my_list = [3, 1, 4, 2]
my_list.sort()
print(my_list)  # Output: [1, 2, 3, 4]

my_list.reverse()
print(my_list)  # Output: [4, 3, 2, 1]

copy_list = my_list.copy()
print(copy_list)  # Output: [4, 3, 2, 1]
```

1.6 List Comprehensions

1.6.1 Basic List Comprehensions

List comprehensions provide a concise way to create lists using a single line of code.

```
squares = [x**2 for x in range(5)]
print(squares)  # Output: [0, 1, 4, 9, 16]
```

1.6.2 Conditional List Comprehensions

Conditions can be added to filter elements during list creation.

```
even_squares = [x**2 for x in range(10) if x % 2 == 0]
print(even_squares)  # Output: [0, 4, 16, 36, 64]
```

1.6.3 Nested Comprehensions

Nested comprehensions handle complex list transformations, such as flattening or transposing.

```
matrix = [[1, 2], [3, 4]]
flattened = [item for sublist in matrix for item in sublist]
print(flattened)  # Output: [1, 2, 3, 4]
```

1.7 Iterating and Looping

1.7.1 for Loops

Covered in Section 2.3, `for` loops are ideal for iterating over lists.

1.7.2 enumerate()

The `enumerate()` function provides both index and value during iteration.

```
for index, value in enumerate([10, 20, 30]):
    print(f"Index {index}: {value}")
# Output:
# Index 0: 10
# Index 1: 20
# Index 2: 30
```

1.7.3 zip() with Multiple Lists

The `zip()` function iterates over multiple lists in parallel.

```
names = ["Alice", "Bob"]
ages = [25, 30]
for name, age in zip(names, ages):
    print(f"{name} is {age} years old")
# Output:
# Alice is 25 years old
# Bob is 30 years old
```

1.8 Common Use Cases

1.8.1 Filtering Items

List comprehensions or `filter()` can select elements based on conditions.

```
numbers = [1, 2, 3, 4, 5]
even_numbers = [x for x in numbers if x % 2 == 0]
print(even_numbers)  # Output: [2, 4]
```

1.8.2 Mapping/Transformation

List comprehensions or `map()` transform each element.

```
numbers = [1, 2, 3]
doubled = [x * 2 for x in numbers]
print(doubled)  # Output: [2, 4, 6]
```

1.8.3 Flattening Nested Lists

Nested comprehensions can flatten nested lists into a single list.

```
nested = [[1, 2], [3, 4]]
flattened = [item for sublist in nested for item in sublist]
print(flattened)  # Output: [1, 2, 3, 4]
```

1.8.4 Finding min, max, sum

Built-in functions `min()`, `max()`, and `sum()` operate on lists.

```
numbers = [10, 20, 30]
print(min(numbers))  # Output: 10
print(max(numbers))  # Output: 30
print(sum(numbers))  # Output: 60
```

1.9 Copying and Cloning Lists

1.9.1 Shallow Copy vs. Deep Copy

- **Shallow Copy**: Copies references to elements, so changes to nested objects affect both lists.
- **Deep Copy**: Creates new copies of all elements, including nested objects.

```
import copy
original = [[1, 2], [3, 4]]
shallow_copy = original[:]
deep_copy = copy.deepcopy(original)
original[0][0] = 10
print(original)      # Output: [[10, 2], [3, 4]]
print(shallow_copy)  # Output: [[10, 2], [3, 4]]
print(deep_copy)     # Output: [[1, 2], [3, 4]]
```

1.9.2 Methods to Copy Lists

- Slicing: `new_list = old_list[:]`
- `list()` constructor: `new_list = list(old_list)`
- `copy()` method: `new_list = old_list.copy()`

1.10 Lists vs Other Data Structures

1.10.1 Lists vs Tuples

- **Lists**: Mutable, defined with `[]`, suitable for dynamic data.
- **Tuples**: Immutable, defined with `()`, ideal for fixed data.

1.10.2 Lists vs Sets

- **Lists**: Ordered, allow duplicates, defined with `[]`.
- **Sets**: Unordered, no duplicates, defined with {}.

1.10.3 When to Use Lists

- When order matters.
- When duplicates are allowed.
- When the collection needs modification.

Table 1.1: Comparison of Lists, Tuples, and Sets

Feature	List	Tuple	Set
Mutability	Mutable	Immutable	Mutable
Order	Ordered	Ordered	Unordered
Duplicates	Allowed	Allowed	Not allowed
Syntax	`[]`	`()`	{}
Use Cases	General use	Fixed data	Unique items

1.11 Key Takeaways

- Lists are versatile for storing and manipulating ordered collections.
- Indexing, slicing, and methods enable efficient data manipulation.
- List comprehensions offer concise solutions for list creation and transformation.
- Understanding shallow vs. deep copies is critical for nested lists.