Python Notes

Condensed Notes

May 6, 2025

Contents

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.

```python
# 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.

```python
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.

```python
# 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.

```python
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.

```python
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.

Chapter 2

Tuple

2.1 Introduction to Python Tuples

Python tuples are a fundamental data structure used to store ordered, immutable collections of items. They are versatile, supporting various data types and operations like indexing and slicing. Unlike lists, tuples cannot be modified after creation, making them ideal for data that should remain constant. This document provides detailed notes on tuples, covering their definition, creation, methods, and use cases, suitable for academic or professional study.

2.2 Definition and Syntax

A tuple is an ordered, immutable sequence of items, which can be of any data type, including duplicates. Tuples are defined using parentheses `()`, with items separated by commas. Parentheses are optional but recommended for clarity.

Listing 2.1: Basic Tuple Syntax

```
my_tuple = (1, 2, 3, "hello", True)
# Without parentheses
another_tuple = 1, 2, 3
```

Key Characteristics:

- **Ordered**: Items maintain a fixed order.
- **Immutable**: Items cannot be changed after creation.
- **Duplicates Allowed**: Multiple identical items are permitted.

2.3 Tuple Creation

Tuples can be created in several ways, depending on the number of items and data source.

2.3.1 Empty Tuples

Empty tuples are created using empty parentheses or the `tuple()` constructor.

Listing 2.2: Creating Empty Tuples

```
empty_tuple = ()
another_empty_tuple = tuple()
```

2.3.2 Single-Item Tuples

A single-item tuple requires a trailing comma to distinguish it from a regular value.

Listing 2.3: Single-Item Tuple

```
single_item_tuple = ("apple",)
# Without comma, it's a string
not_a_tuple = ("apple")  # Type: str
```

2.3.3 Multiple-Item Tuples

Multiple items are separated by commas, with or without parentheses.

Listing 2.4: Multiple-Item Tuples

```
fruits = ("apple", "banana", "cherry")
numbers = (1, 2, 3, 4, 5)
mixed = (1, "hello", 3.14, True)
```

2.3.4 Nested Tuples

Tuples can contain other tuples, creating nested structures.

Listing 2.5: Nested Tuples

```
nested_tuple = (1, (2, 3), (4, 5, 6))
```

2.3.5 Using the tuple() Constructor

The `tuple()` constructor converts iterables to tuples.

Listing 2.6: Using tuple() Constructor

```
list_to_tuple = tuple([1, 2, 3])
string_to_tuple = tuple("hello")
```

2.4 Accessing Tuple Elements

Tuple elements are accessed using indexing and slicing, similar to lists.

2.4.1 Indexing

Positive indices start at 0; negative indices start at -1 from the end.

Listing 2.7: Accessing Elements with Indexing

```
fruits = ("apple", "banana", "cherry")
print(fruits[0])   # Output: apple
print(fruits[-1])  # Output: cherry
```

2.4.2 Slicing

Slicing extracts a subset of the tuple using `[start:end:step]`.

Listing 2.8: Slicing Tuples

```
numbers = (0, 1, 2, 3, 4, 5)
print(numbers[1:4])  # Output: (1, 2, 3)
print(numbers[::2])  # Output: (0, 2, 4)
```

2.5 Tuple Immutability

Tuples are immutable, meaning their elements cannot be modified after creation.

Listing 2.9: Immutability Example

```
fruits = ("apple", "banana", "cherry")
# fruits[0] = "orange"  # Raises TypeError
```

2.5.1 Tuples with Mutable Elements

If a tuple contains mutable objects (e.g., lists), those objects can be modified.

Listing 2.10: Mutable Elements in Tuples

```
nested = (1, [2, 3], 4)
nested[1].append(5)
print(nested)  # Output: (1, [2, 3, 5], 4)
```

2.6 Nested Tuples

Nested tuples allow complex data structures, accessed using multiple indices.

Listing 2.11: Accessing Nested Tuples

```
nested_tuple = (1, (2, 3), (4, 5, 6))
print(nested_tuple[1])     # Output: (2, 3)
print(nested_tuple[2][1])  # Output: 5
```

2.7 Tuple Packing and Unpacking

Packing and unpacking are powerful features for working with tuples.

2.7.1 Packing

Packing creates a tuple by assigning multiple values to a variable.

Listing 2.12: Tuple Packing

```
packed_tuple = 1, 2, 3  # Output: (1, 2, 3)
```

2.7.2 Unpacking

Unpacking assigns tuple elements to multiple variables.

Listing 2.13: Tuple Unpacking

```
a, b, c = packed_tuple
print(a, b, c)  # Output: 1 2 3
```

2.7.3 Use in Functions

Unpacking is common with functions returning multiple values.

Listing 2.14: Unpacking Function Return Values

```
def get_min_max(numbers):
    return min(numbers), max(numbers)

min_val, max_val = get_min_max([1, 2, 3, 4, 5])
print(min_val, max_val)  # Output: 1 5
```

2.8 Using Tuples as Dictionary Keys

Tuples, being immutable, can serve as dictionary keys if all elements are hashable.

Listing 2.15: Tuples as Dictionary Keys

```
locations = {
    ("New York", "USA"): "Big Apple",
    ("London", "UK"): "Big Ben",
}
print(locations[("New York", "USA")])  # Output: Big Apple
```

2.9 Tuple Methods

Tuples have two built-in methods: `count()` and `index()`.

2.9.1 count()

Returns the number of occurrences of a value.

Listing 2.16: Using count() Method

```
numbers = (1, 2, 2, 3, 2)
print(numbers.count(2))  # Output: 3
```

2.9.2 index()

Returns the index of the first occurrence of a value.

Listing 2.17: Using index() Method

```
fruits = ("apple", "banana", "cherry")
print(fruits.index("banana"))  # Output: 1
```

2.10 Iteration over Tuples

Tuples can be iterated using loops.

Listing 2.18: Iterating over a Tuple

```
fruits = ("apple", "banana", "cherry")
for fruit in fruits:
    print(fruit)
```

2.11 Membership Testing

The `in` and `not in` operators check for item existence.

Listing 2.19: Membership Testing

```
fruits = ("apple", "banana", "cherry")
print("banana" in fruits)      # Output: True
print("orange" not in fruits) # Output: True
```

2.12 Tuple vs List Comparison

Tuples and lists are both sequences but differ in key ways.

2.13 Conversion Between Tuples and Other Data Structures

Tuples can be converted to and from other data structures.

Listing 2.20: Converting Data Structures

```
# List to tuple
my_list = [1, 2, 3]
```

Table 2.1: Comparison of Tuples and Lists

Feature	Tuple	List
Syntax	`()`	`[]`
Mutability	Immutable	Mutable
Performance	Faster	Slower
Use Case	Fixed data	Dynamic data
Methods	`count()`, `index()`	Many (e.g., `append()`, `sort()`)

```
list_to_tuple = tuple(my_list)

# Tuple to list
my_tuple = (1, 2, 3)
tuple_to_list = list(my_tuple)

# String to tuple
string_to_tuple = tuple("hello")

# Tuple to string (if elements are strings)
tuple_to_string = "".join(("h", "e", "l", "l", "o"))
```

2.14 Memory Efficiency of Tuples

Tuples are more memory-efficient than lists due to their immutability.

Listing 2.21: Memory Usage Comparison

```
import sys
my_list = [1, 2, 3]
my_tuple = (1, 2, 3)
print(sys.getsizeof(my_list))    # Typically larger
print(sys.getsizeof(my_tuple))   # Typically smaller
```

2.15 Common Use Cases of Tuples

Tuples are used in various scenarios, including:

- Returning multiple values from functions.
- Serving as dictionary keys.
- Storing fixed, immutable data.
- Passing variable arguments to functions.

Listing 2.22: Function Returning Tuple

```
def get_coordinates():
    return 10, 20

```

```
x, y = get_coordinates()
print(x, y)  # Output: 10 20
```

2.16 Tuple Comprehensions via Generator Expressions

Tuples do not support comprehensions directly, but generator expressions can create tuples.

Listing 2.23: Generator Expression for Tuple

```
gen_exp = (x**2 for x in range(5))
my_tuple = tuple(gen_exp)
print(my_tuple)  # Output: (0, 1, 4, 9, 16)
```

2.17 When to Use Lists Instead of Tuples

Lists are preferred when:

- Data needs to be modified (e.g., appending, removing).
- Working with homogeneous data requiring operations like sorting.
- The collection size may change dynamically.

2.18 Advanced Topics

2.18.1 Named Tuples

Named tuples, from the `collections` module, enhance readability by assigning names to tuple fields.

Listing 2.24: Using Named Tuples

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
p = Point(10, 20)
print(p.x, p.y)  # Output: 10 20
```

2.18.2 Tuples in String Formatting

Tuples can provide multiple arguments for string formatting.

Listing 2.25: String Formatting with Tuples

```
name = "Alice"
age = 30
print("Name: %s, Age: %d" % (name, age))
```

2.18.3 Tuples as Function Arguments

Tuples can pass variable arguments using the * operator.

Listing 2.26: Variable Arguments with Tuples

```
def func(*args):
    for arg in args:
        print(arg)

func(1, 2, 3)  # Outputs: 1 2 3
```

2.19 Common Mistakes and Gotchas

- **Forgetting the comma in single-item tuples:**

```
not_a_tuple = ("apple")   # String
a_tuple = ("apple",)      # Tuple
```

- **Attempting to modify a tuple:**

```
my_tuple = (1, 2, 3)
# my_tuple[0] = 4   # Raises TypeError
```

- **Using mutable objects in dictionary keys:**

```
invalid_key = ([1, 2], 3)
# my_dict[invalid_key] = "value"   # Raises TypeError
```

2.20 Conclusion

Python tuples are a powerful, efficient data structure for storing immutable, ordered collections. Their immutability ensures data integrity, while their memory efficiency and performance make them suitable for various applications. By mastering tuples, you can write more robust and efficient Python code.