DATA TYPES AND STRUCTURE IN PYTHON

FOCUS ASTU EXCELLENCY 2023

Adama, Ethiopia

Contents

Introduction to Data Types and Data Structures

In the world of programming, understanding data types and data structures is fundamental. Data types define the nature of data and how it can be manipulated, while data structures provide a way to organize and store collections of data. In Python, a versatile and widely used programming language, there are various data types and data structures available.

This introduction serves as a primer for beginners, providing an overview of key concepts related to data types and data structures. We will explore different data types such as integers, floating-point numbers, complex numbers, strings, and booleans. Additionally, we will delve into essential data structures like lists, tuples, dictionaries, and sets.

Each section will cover the basic properties, operations, and use cases of the respective data type or data structure. You will gain an understanding of how to create and access these entities, perform operations on them, and make informed decisions on when to use each one based on their characteristics.

Furthermore, we will explore the concepts of mutability and immutability, which have significant implications on data structures. Understanding the difference between mutable and immutable objects helps in choosing the appropriate data structure for specific scenarios and understanding how they behave when modified.

By the end of this lecture, you will have a solid foundation in Python data types and data structures. This knowledge will enable you to handle different types of data effectively, choose the right data structure for specific tasks, and perform common operations on them with confidence. Let's embark on this journey to explore the world of data types and data structures in Python!

What are data types?

Data types in programming languages define the type of data that a variable can hold. They specify the kind of values that can be assigned to variables, the operations that can be performed on those values, and the memory space required to store them. In Python, data types categorize values into different classes, which determine the type of operations that can be performed on them.

Data types in Python include numeric types (*integers*, *floating-point numbers*, *complex numbers*), text type (strings), Boolean type (True or False), as well as more complex data structures like lists, tuples, dictionaries, and sets.

Importance of data types in programming:

Data types are essential in programming for the following reasons:

a. **Memory Allocation**: Different data types require different amounts of memory to store values. Understanding data types helps optimize memory usage, especially in large-scale programs or when dealing with large datasets.
b. **Data Integrity**: Data types help enforce rules and constraints on the values that can be assigned to variables. They ensure that the program operates with the correct type of data, preventing errors and unexpected behavior.
c. **Operations and Manipulations**: Each data type has associated operations and functions that can be performed on it. Understanding data types allows programmers to manipulate and transform data effectively, perform arithmetic operations, concatenate strings, access elements in lists, and much more.
d. **Code Readability and Maintainability**: By explicitly declaring the data type of variables, code becomes more readable and easier to understand for both the programmer and other collaborators. It also helps in maintaining and debugging the code in the long run.
e. **Interoperability**: When working with different libraries or modules, understanding data types enables seamless integration and interoperability between different parts of a program.

Data types are fundamental building blocks in programming. They define how data is represented, stored, and manipulated, ensuring data integrity, optimizing memory usage, and enabling efficient operations on variables.

Numeric Data Types

Integers (int):

Integers are whole numbers without any decimal points. They can be positive or negative numbers.

Examples of integers: -3, 0, 42, 1000, etc.

Operations on integers include addition (+), subtraction (-), multiplication (*), division (/), modulus (%), and exponentiation (**).

Python provides built-in functions such as ***abs()***, ***min()***, ***max()***, and ***pow()*** to perform common operations on integers.

Floating-point numbers (float):

Floating-point numbers, also known as floats, represent real numbers with decimal points.

Examples of floats: 3.14, -0.5, 2.0, etc.

Operations on floats are similar to integers and include arithmetic operations such as addition (+), subtraction (-), multiplication (*), division (/), modulus (%), and exponentiation (**).

Python provides additional math functions in the math module, such as ***math.sqrt()***, ***math.sin()***, ***math.cos()***, etc., to perform mathematical operations on floats.

Complex numbers (complex):

Complex numbers are numbers in the form of a + bj, where a and b are real numbers, and j represents the imaginary unit (√-1).

Examples of complex numbers: 2 + 3j, -1 + 2j, etc.

Operations on complex numbers include addition, subtraction, multiplication, division, and conjugation.

Python provides built-in functions such as ***complex()***, ***abs()***, ***real()***, and ***imag()*** to perform operations on complex numbers.

It's important to note that Python automatically infers the data type based on the value assigned to a variable. However, you can explicitly convert between numeric data types using casting functions like ***int(), float(), and complex().***

Text Data Type

Strings (str):

Strings are used to represent text in Python. They are enclosed in either single quotes (') or double quotes ("").

Examples of strings: "Hello", 'Python', "42", 'This is a string!', etc.

Strings can contain any combination of letters, numbers, symbols, and whitespace characters.

Python provides various string operations to manipulate and work with strings, such as concatenation, indexing, slicing, length determination, and more.

String operations:

Concatenation: Combining two or more strings together using the + operator.

Indexing: Accessing individual characters in a string by their position using square brackets ([ ]). Python uses 0-based indexing, so the first character is at index 0.

Slicing: Extracting a portion of a string by specifying a range of indices. The syntax for slicing is string[start:end:step].

Length determination: Finding the length of a string using the len() function.

String methods: Python provides numerous built-in methods for string manipulation, such as ***lower(), upper(), strip(), split(), replace()***, and many more.

Boolean Data Type:

The Boolean data type represents truth values, which can be either True or False.

Booleans are typically used in conditional statements and logical operations to control the flow of the program.

Examples: ***True, False***

Boolean operations:

and: Returns True if both operands are True, otherwise returns False.

or: Returns True if at least one of the operands is True, otherwise returns False.

not: Returns the opposite Boolean value of the operand. If the operand is True, not returns False, and vice versa.

Understanding strings and Booleans is crucial for working with textual data, user input, conditionals, and logical operations in Python.

Lists

Creating and Accessing Lists:

Lists in Python are used to store multiple values in an ordered sequence. They can contain elements of different data types.

To create a list, enclose the elements in square brackets ([]), separated by commas.

For example: `my_list = [1, 2, 3, 'four', 5.0].`

Lists can be assigned to variables and accessed using the variable name. Elements in a list are indexed starting from 0.

Example1:

```
my_list = [1, 2, 3, 'four', 5.0]

print(my_list[0])  # Output: 1

print(my_list[3])  # Output: 'four'
```

List Operations:

Appending: Adding an element to the end of a list using the append() method or the + operator.

Example2:

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

my_list.append(4)

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

Inserting: Adding an element at a specific index in a list using the insert() method.

Example3:

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

my_list.insert(1, 'new')

print(my_list)  # Output: [1, 'new', 2, 3]
```

Removing: Removing an element from a list using the remove() method or the del keyword.

Example4:

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

my_list.remove('new')

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

del my_list[0]

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

List Indexing and Slicing:

Indexing: Accessing individual elements in a list by their index. Positive indices start from 0 (the first element), and negative indices start from -1 (the last element).

Example5:

```
my_list = [1, 2, 3, 4, 5]

print(my_list[0])   # Output: 1

print(my_list[-1])  # Output: 5
```

Slicing: Extracting a portion of a list by specifying a range of indices using the slice notation start:end:step.

Example6:

```
my_list = [1, 2, 3, 4, 5]

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

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

print(my_list[::2])  # Output: [1, 3, 5]
```

Lists are versatile and widely used in Python to store and manipulate collections of data. Understanding list creation, element access, and various operations like appending, inserting, removing, indexing, and slicing is essential for working with lists effectively.

Tuples

Creating and Accessing Tuples:

Tuples in Python are similar to lists but are immutable, meaning they cannot be modified once created.

To create a tuple, enclose the elements in parentheses (()), separated by commas.

For example: `my_tuple = (1, 2, 3, 'four', 5.0).`

Tuples can be assigned to variables and accessed using the variable name. Elements in a tuple are indexed starting from 0.

Example7:

```
my_tuple = (1, 2, 3, 'four', 5.0)

print(my_tuple[0])  # Output: 1

print(my_tuple[3])  # Output: 'four'
```

Immutable Nature of Tuples:

Tuples are immutable, which means their values cannot be changed after creation. Once a tuple is created, its elements cannot be added, removed, or modified.

Trying to modify a tuple will result in a TypeError.

Example8:

```
my_tuple = (1, 2, 3)

my_tuple[0] = 10  # Raises TypeError: 'tuple' object does not support item assignment
```

Tuple Packing and Unpacking:

Tuple Packing: Creating a tuple by grouping multiple values together, separated by commas. The values are automatically packed into a tuple.

Example9:

```
my_tuple = 1, 2, 'three'  # Tuple packing

print(my_tuple)  # Output: (1, 2, 'three')
```

Tuple Unpacking: Assigning the elements of a tuple to multiple variables simultaneously.

Example10:

```
my_tuple = (1, 2, 3)
a, b, c = my_tuple  # Tuple unpacking
print(a, b, c)  # Output: 1 2 3
```

Tuples are useful when you need to store a collection of values that should not be modified. They can be used to return multiple values from a function, as keys in dictionaries, or as elements in sets. Understanding tuple creation, element access, immutability, and packing/unpacking is important for working with tuples effectively.

Dictionaries

Creating and Accessing Dictionaries:

Dictionaries in Python are unordered collections of key-value pairs.

To create a dictionary, enclose the key-value pairs in curly braces ({}) with each pair separated by a colon (:). For example: my_dict = {'key1': value1, 'key2': value2}.

Keys in a dictionary must be unique, and they are typically strings or numbers. Values can be of any data type.

To access the value associated with a specific key, use square brackets ([]).

Example11:

```
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}

print(my_dict['name'])  # Output: 'John'

print(my_dict['age'])   # Output: 30
```

Key-Value Pairs:

Dictionaries store data as key-value pairs, where each key is associated with a value.

Keys are used to access the corresponding values in a dictionary.

Example12:

```
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
```

Dictionary Operations:

Adding or Updating Values: Assign a value to a new key or an existing key in the dictionary using the assignment operator (=).

Example13:

```
my_dict = {'name': 'John', 'age': 30}

my_dict['city'] = 'New York'  # Adding a new key-value pair

my_dict['age'] = 31  # Updating an existing value
```

Removing Values: Use the del keyword to remove a key-value pair from a dictionary.

Example14:

```
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}

del my_dict['age']  # Removing a key-value pair
```

Dictionary Methods: Python provides built-in methods such as keys(), values(), and items() to retrieve keys, values, and key-value pairs respectively.

Example15:

```
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}

keys = my_dict.keys()    # Get all keys

values = my_dict.values()  # Get all values

items = my_dict.items()   # Get all key-value pairs
```

Dictionaries are widely used for storing and accessing data based on unique keys. Understanding dictionary creation, accessing values using keys, and performing operations like adding, removing, and updating key-value pairs is essential for working with dictionaries effectively.

Sets

Creating and Accessing Sets:

Sets in Python are unordered collections of unique elements.

To create a set, enclose the elements in curly braces ({}) or use the set() function.

For example: my_set = {1, 2, 3} or my_set = set([1, 2, 3]).

Sets can contain elements of different data types like numbers, strings, or even other sets.

Accessing elements in a set is not done through indexing because sets are unordered. Instead, you can check for membership using the in keyword.

Example16:

```
my_set = {1, 2, 3}

print(1 in my_set)   # Output: True

print(4 in my_set)   # Output: False
```

Set Operations:

Union: Combining two sets to create a new set that contains all unique elements from both sets. The union operation can be performed using the | operator or the union() method.

Intersection: Finding the common elements between two sets. The intersection operation can be performed using the & operator or the intersection() method.

Difference: Finding the elements that are present in one set but not in another. The difference operation can be performed using the - operator or the difference() method.

Example17:

```
set1 = {1, 2, 3}

set2 = {3, 4, 5}

union_set = set1 | set2  # or set1.union(set2)

intersection_set = set1 & set2  # or set1.intersection(set2)

difference_set = set1 - set2  # or set1.difference(set2)
```

Set Methods:

Adding Elements: Use the add() method to add a single element to a set. If the element already exists in the set, it will not be added again.

Removing Elements: Use the remove() method to remove a specific element from a set. If the element doesn't exist, it raises a KeyError. Alternatively, the discard() method can be used, which removes the element if present, but doesn't raise an error if the element is not found.

Example18:

```
my_set = {1, 2, 3}

my_set.add(4)

my_set.remove(2)

my_set.discard(5)  # No error raised if element doesn't exist
```

Sets are useful when you want to store a collection of unique elements and perform operations like union, intersection, and difference. Understanding set creation, element access, set operations, and set methods is important for working with sets effectively.

Type Conversion (Casting)

Type conversion, also known as casting, is the process of converting one data type to another in a programming language. In Python, you can perform type conversion using various built-in functions or by using the constructor functions of the desired data type.

Converting between Different Data Types:

Integers to Floats: Use the float() function to convert an integer to a floating-point number.

Example19:

```
x = 5
y = float(x)  # y will be 5.0 (a float)
```

Floats to Integers: Use the int() function to convert a float to an integer. This will truncate the decimal part.

Example20:

```
x = 5.7
y = int(x)  # y will be 5 (an integer)
```

Strings to Integers or Floats: Use the int() or float() function to convert a string to an integer or float, respectively. The string should contain a valid numerical representation.

Example21:

```
x = "10"
y = int(x)  # y will be 10 (an integer)
z = "3.14"
w = float(z)  # w will be 3.14 (a float)
```

Integers or Floats to Strings: Use the str() function to convert an integer or float to a string.

Example22:

```
x = 42
```

```
y = str(x)  # y will be "42" (a string)
z = 3.14
w = str(z)  # w will be "3.14" (a string)
```

Implicit vs Explicit Type Conversion:

Implicit Type Conversion: Python automatically performs implicit type conversion when it is safe to do so. For example, when performing arithmetic operations between different numeric data types, Python will automatically convert the operands to a common data type.

Example23:

```
x = 5
y = 2.5
z = x + y  # z will be 7.5 (float) - implicit conversion of x to float
```

Explicit Type Conversion: Explicit type conversion, also known as casting, is performed by the programmer using the appropriate type conversion functions. This allows you to explicitly convert one data type to another.

Example24:

```
x = "10"
y = int(x)  # y will be 10 (an integer) - explicit conversion from string to int
z = 3.14
w = int(z)  # w will be 3 (an integer) - explicit conversion from float to int
```

Understanding type conversion and when to use it is important for manipulating and working with different data types in Python. Implicit type conversion is performed automatically by Python, while explicit type conversion requires the use of appropriate conversion functions to convert between data types.

Data Structures Comparison

Understanding the differences between lists, tuples, dictionaries, and sets

Choosing the appropriate data structure for different scenarios

Lists, **tuples**, **dictionaries**, and **sets** are all data structures in Python, but they differ in their properties and use cases. Here's a comparison of these data structures:

Lists:

- Ordered collection of elements that can be modified (mutable).
- Elements can be of different data types.
- Accessed using indexing (e.g., `my_list[0]`) and slicing (e.g., `my_list[1:3]`).
- Useful for storing and manipulating data when the order and mutability of elements are important.
- Use a list when you need to add, remove, or modify elements frequently and maintain their order.

Tuples:

- Ordered collection of elements that cannot be modified (immutable).
- Elements can be of different data types.
- Accessed using indexing (e.g., `my_tuple[0]`) and slicing (e.g., `my_tuple[1:3]`).
- Useful for representing a group of related values that should remain unchanged.
- Use a tuple when you want to store a collection of values that should not be modified.

Dictionaries:

- Unordered collection of key-value pairs.
- Keys are unique and associated with values.
- Accessed using keys (e.g., `my_dict['key']`).
- Useful for storing and accessing data based on unique keys.
- Use a dictionary when you have a mapping between keys and values and need fast access to values based on their keys.

Sets:

- Unordered collection of unique elements.
- Does not allow duplicate values.
- Accessed using membership operators (e.g., element in my_set).
- Useful for storing unique values and performing set operations like union, intersection, and difference.
- Use a set when you want to store a collection of unique elements or perform set operations efficiently.

Choosing the appropriate data structure depends on the specific requirements of your scenario. Consider the following factors:

- **Order**: If the order of elements matters, use a list or a tuple. If the order doesn't matter, use a dictionary or a set.
- **Mutability**: If you need to modify the elements, use a list. If the elements should be immutable, use a tuple, dictionary, or set depending on other requirements.
- **Key-Value Mapping**: If you need to associate values with unique keys, use a dictionary.
- **Uniqueness**: If you need to store unique elements and perform set operations, use a set.
- **Performance**: Consider the operations you'll perform frequently (access, modification, searching) and choose a data structure that optimizes those operations.

Understanding the differences and characteristics of these data structures helps you select the most appropriate one for your specific use case.

Mutable vs Immutable Data Types

Understanding the concept of mutability in Python

How mutability affects data structures

In Python, mutability refers to whether an object can be modified after it is created. Mutable objects can be changed, while immutable objects cannot be modified once created. Understanding mutability is important because it affects how data structures behave and how they are manipulated.

Mutable Data Types:

Lists, dictionaries, and sets are mutable data types in Python.

Mutable objects can have their values modified, added, or removed without creating a new object.

For example, you can change the value of an element in a list or add a new key-value pair to a dictionary.

Immutable Data Types:

Tuples, strings, and numbers (integers, floats, etc.) are immutable data types in Python.

Immutable objects cannot be changed once they are created. Any operation that appears to modify an immutable object actually creates a new object with the updated value.

For example, when you concatenate two strings, a new string object is created with the combined value.

Effects of Mutability on Data Structures:

Lists vs. Tuples:

- Lists are mutable, so you can change their elements, add or remove elements, or reorder the elements.
- Tuples are immutable, so their elements cannot be changed after creation. If you want to modify a tuple, you need to create a new tuple with the desired changes.
- Lists are preferred when you need a data structure that can be modified. Tuples are used when you want to represent a collection of values that should remain unchanged.

Dictionaries:

- Dictionaries are mutable, allowing you to modify, add, or remove key-value pairs.
- The keys of a dictionary must be immutable objects, such as strings or tuples, to ensure hashability.

Sets:

- Sets are mutable, allowing you to add or remove elements.
- However, since sets contain only unique elements, modifying a set will not change the existing elements.

Understanding mutability is crucial because it impacts how you work with and manipulate data structures. Mutable data types are useful when you need to modify the data or change its structure, while immutable data types provide safety and predictability by ensuring that the data remains unchanged once created.

Common Data Structure Operations

Finding the Length of a Data Structure:

Use the ***len()*** function to determine the length of a data structure.

Example25:

```
my_list = [1, 2, 3, 4, 5]

length = len(my_list)  # length will be 5
```

Checking for the Presence of an Element:

Use the ***in*** keyword to check if an element is present in a data structure.

Example26:

```
my_tuple = (1, 2, 3, 4, 5)

is_present = 3 in my_tuple  # is_present will be True
```

Iterating Over Elements in a Data Structure:

Use loops, such as for or while, to iterate over elements in a data structure.

Example27:

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

for element in my_set:

    print(element)  # Prints each element of the set

my_dict = {'a': 1, 'b': 2, 'c': 3}

for key, value in my_dict.items():

    print(key, value)  # Prints each key-value pair of the dictionary
```

These common operations are applicable to various data structures in Python. By utilizing these operations, you can work with the length of a data structure, check for the presence of specific elements, and iterate over elements to perform desired tasks or operations.

Summary

In this lecture for Python beginners, we covered various topics related to data types and data structures. We started by introducing the concept of data types and highlighting their importance in programming. We discussed numeric data types such as integers, floating-point numbers, and complex numbers, along with their characteristics and usage.

Moving on, we explored text data types and focused on strings, including string operations like concatenation, indexing, and slicing. We also touched upon the boolean data type, which represents true and false values, and discussed boolean operations like and, or, and not.

Next, we delved into different data structures and their functionalities. We covered lists, which are ordered collections of elements, and learned about list creation, access, and various list operations. We then explored tuples, which are similar to lists but immutable, and discussed tuple creation, access, and packing/unpacking

The lecture also covered dictionaries, which store key-value pairs, and explained dictionary creation, access, and operations. Finally, we discussed sets, which store unique elements, and examined set creation, access, and set operations

Additionally, we covered the concept of type conversion (casting), including converting between different data types and implicit vs. explicit type conversion.

By understanding these concepts, Python beginners will have a solid understanding of data types and data structures, allowing them to handle different types of data and choose the appropriate data structure for specific tasks.

Reference

1. **Python Official Documentation**: The official documentation for Python provides comprehensive information about data types, data structures, and their usage in Python. It covers detailed explanations, examples, and usage guidelines. You can access it at: https://docs.python.org/

2. Python Tutorial on **W3Schools**: W3Schools offers a beginner-friendly tutorial on Python that covers data types, data structures, and various operations. It provides interactive examples and exercises to reinforce your learning. You can find it at: https://www.w3schools.com/python/

3. Python Data Structures - **GeeksforGeeks**: GeeksforGeeks is a popular platform for programming tutorials and resources. They have a section dedicated to Python data structures, covering topics such as lists, tuples, dictionaries, sets, and more. You can explore it at: https://www.geeksforgeeks.org/python-data-structures/

4. Python Data Structures and Algorithms - **Real Python**: Real Python is an online platform that offers in-depth tutorials, articles, and resources for Python developers. They have a comprehensive guide on Python data structures and algorithms, providing practical examples and explanations. You can access it at: https://realpython.com/python-data-structures/

5. Python for Data Structures, Algorithms, and Interviews - **Udemy**: Udemy is an online learning platform that offers a wide range of Python courses. "Python for Data Structures, Algorithms, and Interviews" is a course specifically focused on data structures and algorithms in Python. It covers various data structures and their implementations. You can find it at: https://www.udemy.com/course/python-for-data-structures-algorithms-and-interviews/