Briefing Doc: Python

Unit 1: Introduction and Overview

Introduction

- **Python** stands out as a high-level programming language, celebrated for its simplicity and versatility.
- Python accommodates multiple programming paradigms, including procedural, object-oriented, and functional styles.
- Its extensive standard library and vibrant community have propelled its widespread adoption across domains like web development, data science, and artificial intelligence.

Comments

- Comments are integral to code documentation. While they are disregarded by the compiler or interpreter during execution, comments play a crucial role in explaining the code, enhancing its readability, and aiding fellow developers in understanding its purpose.
- **Single Line Comments**: Single-line comments start with `#`
- **Multi-Line Comments**: multi-line comments are enclosed in triple quotes (''' or """).

Keyword

- **Keywords** in a programming language hold specific predefined meanings and are reserved, preventing their use as variable names or identifiers.
- These keywords are fixed and case-sensitive, providing specific functionalities or signifying actions within the code. Examples of Python keywords include **"if", "else", "while", "for", "def", "class", and "return".**

Identifiers

- **Identifiers** are names employed to uniquely identify variables, functions, classes, or other entities within a program.
- Adhering to specific rules, identifiers must commence with a letter (A-Z or a-z) or an underscore (_), cannot clash with keywords, are case-sensitive, and should meaningfully represent the element they denote.

Variables

- **Variables** serve as containers for storing data values. Python's dynamic typing eliminates the need for explicit type declarations. The type of a variable is determined at runtime.
- **Assignment statements** are used to assign values to variables. This is achieved using the **"=" operator.**

Data Types

- Python offers a rich set of standard data types to represent various kinds of data. These include:
    - **Numeric Types:** *int*, *float*, *complex*
    - **Sequence Types:** *list*, *tuple*, *range*
    - **Text Type:** *str*
    - **Boolean Type:** *bool*
- **Internal types**, employed internally by the Python interpreter, include *code objects*, *stack frames*, and more. Generally not directly manipulated by users, these types are fundamental to Python's internal workings, often enabling functionalities for higher-level types and operations.
- Examples of internal types include:
    - *NoneType*: Represents the absence of a value (None).
    - *Frame Type*: Holds the execution context for functions, crucial for debugging.
    - *Function Type*: Represents all functions and lambdas.

Operators and Functions

Operators

- **Operators** are symbols or keywords that perform operations on values (operands). They facilitate data manipulation, calculations, comparisons, and more. Python provides a variety of operators, classified as:
    - **Arithmetic Operators:** +, -, , _/, %,_ *, //
    - **Comparison Operators:** ==, !=, >, <, >=, <=
    - **Assignment Operators:** =, +=, -=, _=, /=, %=,_ *=, //=
    - **Logical Operators:** and, or, not

- **Bitwise Operators:** &, |, ^, ~, <<, >>
- **Membership Operators:** in, not in
- **Identity Operators:** is, is not

Built-in Functions

- **Built-in functions** are pre-defined functions in Python, readily available for use. Designed for handling common tasks, these functions streamline coding by providing efficient solutions for data manipulation, type conversion, mathematical calculations, input/output operations, and more. Some commonly used built-in functions include:
    - **Input and Output Functions:** *print()*, *input()*
    - **Type Conversion Functions:** *int()*, *float()*, *str()*
    - **Mathematical Functions:** *abs()*, *pow()*, *round()*

Numbers and Sequences

- **Numbers** are fundamental data types used to represent numeric values. Python supports three main types of numbers:
    - **Integers (*int*):** Whole numbers without fractional parts.
    - **Floating-Point Real Numbers (*float*):** Numbers with decimal points.
    - **Complex Numbers:** Numbers with both real and imaginary parts, expressed as *a + bj*, where *a* and *b* are real numbers, and *j* is the imaginary unit (√-1).
- **Sequences** are ordered collections of items, where each item occupies a specific position (index). Python supports several types of sequences:
    - **Strings:** Sequences of characters, commonly used to represent text.
    - **Lists:** Ordered and mutable collections that can hold items of different data types.
    - **Tuples:** Similar to lists but immutable, meaning their elements cannot be changed after creation.
    - **Ranges:** Sequences of numbers, often used in loops.

Working with Strings

- **Strings** are sequences of characters, enclosed in single ('...') or double ("...") quotes. Python provides numerous built-in methods to manipulate strings efficiently, facilitating tasks like formatting, searching, or replacing text. Some common string methods include:
    - **len():** Returns the number of characters in the string.

  - **lower() and upper():** Convert a string to all lowercase or all uppercase, respectively.
  - **strip():** Removes leading and trailing whitespace from the string.
  - **replace():** Replaces occurrences of a substring with another string.
  - **split():** Splits a string into a list of substrings based on a delimiter.
  - **find():** Returns the index of the first occurrence of a substring, returning -1 if not found.

- **Immutability:** Strings in Python are immutable, meaning their values cannot be changed after they are created. Any operation that appears to modify a string actually creates a new string with the changes.
- **String Slicing:** Strings can be "sliced" to extract substrings using the syntax *string[start : end : step]*. This allows for flexible manipulation of string portions.

Conditionals and Loops

- **Conditional statements** are essential for controlling the flow of a program by executing specific blocks of code based on given conditions. Python supports several types of conditional statements:
  - **if Statement:** Executes a block of code only if a specified condition evaluates to True.
  - **else Statement:** Provides an alternative code block to execute if the preceding *if* condition is False.
  - **elif Statement:** Allows testing additional conditions when the initial *if* condition is False.
- **Loops** are constructs that allow a block of code to be executed repeatedly as long as a certain condition is met. They are crucial for automating repetitive tasks and iterating over sequences. Python offers two primary loop types:
  - **while Loop:** Repeats a code block as long as its associated condition remains True.
  - **for Loop:** Iterates over a sequence (like a list, tuple, or string) or a range of numbers, executing the code block for each item in the sequence.
- **Loop Control Statements** offer finer control over the execution of loops:
  - **break Statement:** Terminates the loop prematurely, regardless of the loop's condition.
  - **continue Statement:** Skips the remaining code in the current loop iteration and moves to the next iteration.
  - **pass Statement:** Acts as a placeholder in the loop, indicating that no action should be taken in that iteration.

Unit 2: Lists and Dictionaries

Lists

- **Lists** are ordered, mutable collections, allowing for modifications after creation. They are versatile, capable of storing elements of different data types, including other lists (nested lists). Some key features of lists include:
  - **Ordered:** Elements maintain their order.
  - **Mutable:** Elements can be added, removed, or modified.
  - **Heterogeneous:** Lists can contain items of various data types.
  - **Dynamic:** Lists can grow or shrink as needed.
  - **Indexed:** Each element can be accessed using its index, starting from 0 for the first element.

Tuples

- **Tuples**, unlike lists, are immutable sequences. Once created, their elements cannot be changed. They are often used when you need to ensure data integrity, preventing accidental modifications. Defining characteristics of tuples are:
  - **Immutable:** Elements cannot be altered after creation.
  - **Ordered:** Elements retain their specified order.
  - **Allow Duplicates:** Tuples can contain identical elements.
  - **Heterogeneous:** Can accommodate different data types within a single tuple.

Dictinoaries

- **Dictionaries** are data structures that store data in key-value pairs, allowing for efficient retrieval of values based on their associated keys. Some notable features of dictionaries are:
  - **Key-Value Pairs:** Data is organized as key-value pairs.
  - **Mutable:** Dictionaries can be modified by adding, updating, or removing key-value pairs.
  - **Keys Must Be Immutable:** Dictionary keys must be of immutable data types like strings, numbers, or tuples (as long as they contain only immutable elements), as this ensures their uniqueness and hashability for efficient lookup.
  - **Unordered (Up to Python 3.6):** Dictionaries did not maintain the order of insertion prior to Python 3.7. From Python 3.7 onwards, insertion order is preserved.
  - **Fast Lookups:** Retrieving a value using its key is a very fast operation in dictionaries due to their hash-based implementation.

Sets in Python

- **Sets** are unordered, mutable collections that strictly enforce the uniqueness of their elements. No duplicate values are permitted within a set. Sets are often used for tasks requiring distinct values and for performing mathematical set operations. Key characteristics of sets include:
    - **Unordered:** The order of elements in a set is arbitrary and not guaranteed to remain consistent.
    - **Unique:** Only one instance of each distinct element is allowed within a set.
    - **Mutable:** Elements can be added or removed from a set, but the elements themselves must be immutable (e.g., numbers, strings, tuples) to ensure their hashability.
- **Set Operations**: Include adding and removing elements, common set methods, and mathematical set operations like union, intersection, and difference.
- **Set Comprehensions**: Concise way to create sets using a syntax similar to list comprehensions
    - `{expression for item in iterable if condition}`

Unit 3: Regular Expression and Exception Handling

Regular Expressions

- **Regular Expressions (REs)** are potent tools for text processing. They are used to define patterns for searching, matching, and manipulating strings based on these patterns. Applications of REs include input validation, data extraction from text, and text transformation tasks.
- **Special Symbols:** Used in regular expressions, including `., ^, $, *, +, ?, {n}, {n,},` `` [], `, \d, \w, and \s. ``
- Python's re Module: Provides functions like `re.match()`, `re.search()`, `re.findall()`, `re.sub()`, and `re.split()`.

Exception Handling

- **Exception Handling** is a crucial mechanism for dealing with runtime errors that can disrupt the normal flow of a program. These errors, known as exceptions, can arise from various situations, such as invalid input, attempts to access nonexistent files, or logical errors in the code. Python's *try-except* structure provides a robust way to catch and handle exceptions gracefully, preventing the program from crashing.

- **Raising Exceptions**: Using the `raise` keyword to throw exceptions manually.
- **Assertions**: Debugging aids for testing assumptions in code using the assert keyword
- **Standard Exceptions**: Built-in exception classes like `IndexError`, `KeyError`, `ValueError`, and `ZeroDivisionError`

Unit 4: OOPs in Python

Classes

- **Classes** are the foundation of object-oriented programming (OOP). A class acts as a blueprint or template for creating objects (instances). Objects are entities that have both data (attributes) and behaviors (methods). Classes encapsulate these attributes and methods, providing a structured way to represent real-world concepts or entities in code.

OOPs (Object Oriented Programming)

- **Object-oriented programming** is a programming paradigm built upon the concept of "objects," which encapsulate data and the procedures that operate on that data. Python supports four key principles of OOP:

    - **Encapsulation:** Bundling data (attributes) and the methods that operate on the data within a class. This concept restricts direct access to some of the object's components, promoting data integrity and maintainability.
    - **Abstraction:** Hiding the complex implementation details of a class and presenting only essential information and functionalities to the user. This simplifies interactions with objects, allowing users to focus on what an object does without needing to understand its internal workings.
    - **Inheritance:** Allows a class (child class) to inherit attributes and methods from another class (parent class). This fosters code reusability and allows for the creation of specialized classes based on more general ones, establishing a hierarchical relationship among classes.
    - **Polymorphism:** Enables objects of different classes to be treated as objects of a common type. This means that a single function or method can operate on objects of various classes, providing flexibility and reducing code duplication.

The `__init()__` method

- In Python classes, the `__init__()` method serves as a constructor. It is automatically invoked when a new object of the class is created. Its primary role is to initialize the object's attributes, setting their initial values.

Instance Methods

- **Instance Methods:** These are functions defined within a class, designed to operate on individual objects (instances) of that class. They have access to the object's attributes and can modify them. The first parameter of an instance method is always *self*, referring to the instance itself, allowing the method to access and manipulate its data.

Class Variables

- **Class Variables:** While instance variables are unique to each object, class variables are shared among all instances of a class. Defined inside the class but outside of any methods, class variables are used to store data common to all objects of that class.

Type Identification

- **Type Identification:** Python offers ways to determine the type of an object at runtime. The *type()* function returns the type of an object, while the *isinstance()* function checks if an object is an instance of a particular class or a subclass of that class. These functions are valuable for making decisions based on the type of an object, particularly when dealing with multiple types of data.ng