Python Exam Revision: Algorithms to Data Structures

1 source

These documents comprise a two-part lecture series focusing on revising key concepts for an exam in a programming or computer science course. The first part, Lecture 12.1, covers fundamental programming topics such as algorithms, flowcharts, variables, expressions, conditional execution, iteration, functions, objects, strings, and file handling. Lecture 12.2 continues the revision by addressing data structures (lists, dictionaries, sets), software errors (syntax, runtime, logic, exceptions), module usage, code structuring and documentation, and algorithm design strategies, including time complexity, sorting, searching, and recursion. Both lectures incorporate practice questions to aid student understanding and explicitly mention that exam details will be provided.

Summarize the key topics covered in this Python revision guide.

This Python revision guide covers a comprehensive range of topics essential for understanding programming in Python, spanning from fundamental concepts to more advanced strategies. The material is divided into two main lectures.

Lecture 12.1 focuses on the following six topics:

1.

Algorithms and Flowcharts: This section introduces programming's importance, the definition of an "algorithm," programming languages, and the difference between "thinking like a computer" and "thinking like a human". It also covers flowcharts, their utility, essential elements, how to draw them for simple algorithms, and qualities of well-drawn flowcharts.

2.

Statements and Expressions: Key concepts include variables, statements, and comments, along with numeric and string literals, rules for valid and meaningful variable names, and case sensitivity. The section also delves into expressions, numeric operators (like +, -, \*, /, \*\*, %, //), operator precedence, the behavior of '+' with strings and numbers, Python's string, integer, and float types, and type conversion.

3.

Booleans and Conditional Execution: This topic covers Booleans, truthy and falsy values for string and numeric types, comparison operators (==, !=, >, <, >=, <=), and logical operators (and, or, not). It also explains control flow, sequence and selection control structures, Python's selection statements (if, elif, else), indentation, and nesting.

4.

Iteration: This part of the guide explains iteration control structures, Python's while loops, definite and indefinite loops, and how to finish early from a loop using continue and break. It also covers for loops, ranges, and aggregation techniques (sum, average, minimum/maximum, count).

5.

Functions and Objects:

◦

Functions: The purpose of functions, how to call and write them, arguments and parameters, return values, and variable scope within functions are discussed. It highlights that inputs are provided as arguments accessed by parameter names, and outputs are returned using return statements.

◦

Objects: This section introduces objects, classes, constructors, instance variables, methods, references, and mutability. It details constructors as special methods run when a class instance is created, used to attach instance variables.

6.

Strings and Files:

◦

Strings: Key aspects include escape sequences (like \, ', ", \n, \t), different string literal kinds (single/double quotes, multiline), indexing and slicing strings, using string methods, f-strings, format specifiers for decimal places, and case sensitivity.

◦

Files: This covers text files, file objects, different file modes ('read', 'write', 'append'), the concept of a file cursor position (which dictates where reading/writing starts and moves with operations), the workflow for file operations, the behavior of readline method, and performing file system operations.

Lecture 12.2 revises the remaining five topics:

7.

Data Structures:

◦

Lists: This section covers list literals, operations (concatenation, indexing, slicing, length), mutating lists (replacing, appending, deleting items), filtering, mapping, and the consequences of list mutability.

◦

Dictionaries and Sets: It discusses the differences between sets and lists, basic set operations (add, remove, in), dictionaries, basic dictionary operations (adding, getting, deleting items, in), and iterating over dictionary items with a for loop.

8.

Software Errors: This topic categorizes errors into syntax errors, runtime errors, and logic errors, explaining what each entails and how they manifest. It also covers asserts, strategies for test design (equivalence partitioning and boundary-value analysis), exceptions, information displayed in exception output (stack traces), handling exceptions (try, except), raising exceptions (raise), and the two broad approaches to avoiding/handling runtime errors (EAFP and LBYL).

9.

Using Modules: This explains how to import and use modules, access definitions within them, and the purpose of JSON files. It also covers third-party packages and the purpose of pip.

10.

Structuring and Documenting Code: This section discusses modular programming and its benefits, creating modules in Python, and the advantages of formatting source code. It also differentiates between good and bad comments, and explains the purpose of docstrings and README files as different levels of documentation.

11.

Algorithm Design Strategies:

◦

Time Complexity: This part covers strategies for approaching problems, the importance of algorithmic complexity, and the concepts of optimal and approximate solutions.

◦

Sorting, Searching, Recursion: It includes using sorting and searching to solve problems, and a detailed explanation of recursion, including base cases (which return without a recursive call) and recursive cases (which involve a call to the same function, moving closer to a base case).

What’s the difference between a function and an object in Python?

In Python, both functions and objects are fundamental programming constructs, but they serve different purposes and have distinct characteristics:

Functions:

•

Purpose: Functions are used to encapsulate reusable blocks of code that perform specific tasks. They help in organizing code and avoiding repetition.

•

Inputs and Outputs: Functions can take inputs, which are provided as arguments when the function is called. These arguments are accessed within the function by their corresponding parameter names. Functions can also produce outputs, which are returned using return statements.

•

Definition and Calling: You define a function using the def keyword, specifying its parameters. To execute the code within a function, you "call" it by its name.

•

Scope: Variables defined within a function typically have a local scope, meaning they are accessible only inside that function.

•

Example: A function like add\_up\_numbers might take a list of numbers as an argument and return their sum.

Objects:

•

Purpose: Objects are instances of classes, which serve as blueprints for creating them. They are used to bundle data (instance variables) and behavior (methods) together, representing entities or concepts within a program.

•

Classes: An object is created from a class, which defines its structure and what it can do.

•

Constructors: Most classes include a special method called a constructor (e.g., \_\_init\_\_). This constructor specifies statements that run when an instance of the class is created. It is commonly used to attach instance variables to the object instance. When an object is instantiated, the constructor is automatically invoked.

•

Instance Variables and Methods: Objects have instance variables that hold their data (e.g., self.artist or self.title for an Album object). These variables are accessed using a full stop character (.). Objects can also have methods, which are functions associated with the object that define its behavior.

•

References and Mutability: Objects are manipulated via references, and they can be mutable, meaning their internal state (instance variables) can be changed after they are created.

In essence, a function is primarily a block of code that performs an action, while an object is an entity that bundles data and the functions (methods) that operate on that data. You call a function, but you instantiate an object from a class and then interact with its data and methods.

Explain the example given in the add\_up\_numbers function.

The add\_up\_numbers function example demonstrates how inputs are provided to functions as arguments, how these arguments are accessed by parameter names within the function, and how functions return outputs.

Here's a breakdown of the example:

The Function Definition:

def add\_up\_numbers(numbers):

total = 0

for number in numbers:

total = total + number

return total

•

def add\_up\_numbers(numbers): defines a function named add\_up\_numbers. It has one parameter named numbers.

•

Inside the function, a variable total is initialized to 0.

•

A for loop iterates through each number in the numbers parameter.

•

In each iteration, the current number is added to total.

•

return total specifies that the function's output will be the final value of total.

Calling the Function and Passing Arguments:

my\_list = [4-7]

result = add\_up\_numbers(my\_list)

print(result)

•

my\_list = creates a list named my\_list containing four integer values.

•

result = add\_up\_numbers(my\_list) is the function call. Here, my\_list is provided as an argument to the add\_up\_numbers function.

•

When the function is called, the numbers parameter inside the add\_up\_numbers function will reference the same object that my\_list references (i.e., ).

Function Execution and Return Value:

•

The function then executes:

◦

total starts at 0.

◦

The loop adds 3, then 6, then 2, then 1 to total.

◦

After the loop, total will be 3 + 6 + 2 + 1 = 12.

•

The function returns 12 (the value of total).

•

Once the function finishes executing, the function call add\_up\_numbers(my\_list) evaluates to this return value.

Using the Return Value:

•

result = add\_up\_numbers(my\_list) means that the value 12 returned by the function is assigned to the variable result.

•

In this specific case, result will reference the same object that total referenced just before the function returned.

•

print(result) then outputs the value stored in result, which is 12.

Create flashcards or summaries for each major topic in this revision.”

Here are summaries for each major topic covered in the Python revision guide, suitable for use as flashcards:

--------------------------------------------------------------------------------

Topic 1: Algorithms and Flowcharts

•

Algorithms:

◦

An "algorithm" is a step-by-step procedure for solving a problem.

◦

Understanding the importance of programming and programming languages is key.

◦

Differentiate between "thinking like a computer" and "thinking like a human".

•

Flowcharts:

◦

Flowcharts are diagrams that visually represent algorithms, showing their steps and decision points.

◦

They are useful for visualizing logic and planning programs.

◦

Know the essential elements (start/end, process, input/output, decision, flow lines) and how to draw a flowchart for a simple algorithm.

◦

Understand the qualities of well-drawn flowcharts (e.g., clarity, correctness).

--------------------------------------------------------------------------------

Topic 2: Statements and Expressions

•

Variables, Statements, and Comments:

◦

Literals: Numeric (e.g., 3, 3.0) and string (e.g., '3') literals represent fixed values.

◦

Variables: Used to store data; understand rules for valid and meaningful variable names, and case sensitivity in Python.

◦

Statements: Instructions that Python can execute (e.g., assignment statements, print statements).

◦

Comments: Used to explain code and improve readability; ignored by the interpreter.

•

Expressions and Types:

◦

Expressions: Combinations of values, variables, and operators that Python evaluates to produce a result.

◦

Numeric Operators: Learn +, -, \*, / (division), \*\* (exponentiation), % (modulo), // (floor division).

◦

Operator Precedence: Understand the order in which operators are evaluated in an expression.

◦

+ Behavior: How the + operator works differently with strings (concatenation) and numbers (addition).

◦

Data Types: Familiarize yourself with Python's core data types: string (str), integer (int), and float (float).

◦

Type Conversion: Know how to convert values between different data types (e.g., int(), str(), float()).

--------------------------------------------------------------------------------

Topic 3: Booleans and Conditional Execution

•

Comparisons and Boolean Logic:

◦

Booleans: A data type representing True or False values.

◦

Truthy and Falsy: Understand which values (for string and numeric types) evaluate to True (truthy) or False (falsy) in a boolean context (e.g., empty strings/zero are falsy).

◦

Comparison Operators: Know == (equal to), != (not equal to), > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to).

◦

Logical Operators: Understand and, or, not for combining or negating boolean expressions.

•

Conditional Execution:

◦

Control Flow: The order in which instructions in a program are executed.

◦

Control Structures: Differentiate between sequence (straight line execution) and selection (conditional execution) control structures.

◦

Selection Statements: Master Python's if, elif, and else statements for executing code blocks based on conditions.

◦

Indentation and Nesting: Proper indentation is crucial in Python for defining code blocks and understanding nested conditional statements.

--------------------------------------------------------------------------------

Topic 4: Iteration

•

Iteration (Loops):

◦

Iteration Control Structures: Techniques for repeatedly executing a block of code.

◦

while Loops: Understand Python's while loop for indefinite iteration (loop continues as long as a condition is true).

◦

Kinds of Loops: Differentiate between definite loops (number of iterations is known, often for loops) and indefinite loops (number of iterations is unknown until a condition is met, often while loops).

◦

Finishing Early: Know how to use continue (skips current iteration, continues with next) and break (exits the loop entirely).

•

for Loops and Aggregation:

◦

for Loops: Used for definite iteration, typically over a sequence (e.g., list, string, range).

◦

Ranges: Understand range() function to generate sequences of numbers for for loops.

◦

Aggregation: Techniques to process data in a loop to produce a single summary result, such as calculating sum, average, finding minimum/maximum, or counting items.

--------------------------------------------------------------------------------

Topic 5: Functions and Objects

•

Functions:

◦

Purpose: Encapsulate reusable code blocks, promote modularity, and avoid repetition.

◦

Calling Functions: Executing a defined function by its name.

◦

Writing Functions: Defining functions using def, including parameters and a function body.

◦

Arguments and Parameters: Inputs are provided as arguments during a function call, and these are accessed inside the function by their corresponding parameter names.

◦

Return Values: Functions output results using return statements.

◦

Variable Scope: Understand that variables defined within a function typically have local scope, meaning they are only accessible inside that function.

◦

Example (add\_up\_numbers): An argument (e.g., my\_list) is passed to the function, becoming the value of the parameter (numbers). The function performs calculations and returns a result (total), which can then be assigned to a variable outside the function (e.g., result).

•

Objects:

◦

Objects and Classes: Objects are instances of classes, which serve as blueprints.

◦

Constructors: A special method (often \_\_init\_\_) within a class that runs automatically when an object (instance) of that class is created. They are used to attach instance variables to the object.

◦

Instance Variables: Data attributes that belong to a specific object instance (e.g., self.artist, self.title in an Album object). Accessed using dot notation (.).

◦

Methods: Functions defined within a class that operate on the object's data.

◦

References: Objects are manipulated via references.

◦

Mutability: Understand that objects can be mutable, meaning their internal state (instance variables) can be changed after creation.

--------------------------------------------------------------------------------

Topic 6: Strings and Files

•

Strings:

◦

Escape Sequences: Special characters used within strings (e.g., \\ for backslash, \' for single quote, \" for double quote, \n for newline, \t for tab).

◦

String Literals: Different ways to define strings (single quotes '', double quotes "", or multiline strings using triple quotes ''' ''' or """ """).

◦

Indexing and Slicing: Accessing individual characters (indexing) or substrings (slicing) using square brackets [].

◦

String Methods: Built-in functions that perform operations on strings (e.g., .upper(), .lower(), .strip()).

◦

f-strings: A modern and convenient way to embed expressions inside string literals for formatting.

◦

Format Specifiers: Used within f-strings or .format() for precise output formatting, such as .6f for floating-point numbers with 6 decimal places.

◦

Case Sensitivity: Understand that string comparisons are case-sensitive.

•

Files:

◦

Text Files: Basic understanding of how text files are stored and accessed.

◦

File Objects: When a file is opened, Python returns a file object that represents the connection to the file.

◦

File Modes: Different modes for opening files: 'r' (read), 'w' (write, overwrites existing file), 'a' (append, adds to end of existing file). Also 'r+' (read and write, cursor at start), 'w+' (write and read, overwrites), 'a+' (append and read, cursor at end).

◦

Cursor Position: The conceptual "cursor" in a file dictates where reading or writing will begin. Its initial position depends on the file mode.

◦

Workflow: The typical steps for working with files: open, perform operations (read/write), close.

◦

readline() Method: Reads a single line from the file, including the newline character, and moves the cursor.

◦

Cursor Movement: Reading, writing, and seeking (e.g., seek(0) to move to the beginning) all change the cursor's position.

◦

File System Operations: General awareness of operations beyond reading/writing content, such as deleting or renaming files (though not detailed here).

--------------------------------------------------------------------------------

Topic 7: Data Structures

•

Lists:

◦

List Literals: Creating lists using square brackets [].

◦

List Operations: Concatenation (+), indexing ([index]), slicing ([start:end]), and determining length (len()).

◦

Mutating Lists: Lists are mutable, meaning their contents can be changed after creation.

▪

Replacing: Changing an item at a specific index.

▪

Appending: Adding items to the end using .append().

▪

Deleting: Removing items using del with the item's position (index).

◦

Filtering and Mapping: Concepts of creating new lists by selecting (filtering) or transforming (mapping) elements from an existing list.

◦

Consequences of List Mutability: Understanding how changes to a list referenced by multiple variables can affect all references.

•

Dictionaries and Sets:

◦

Sets vs. Lists: Understand key differences, primarily that sets are unordered collections of unique elements, while lists are ordered and can contain duplicates.

◦

Basic Set Operations: Adding elements (.add()), removing elements (.remove()), and checking for membership (in).

◦

Dictionaries: Unordered collections of key-value pairs. Keys must be unique and immutable.

◦

Basic Dictionary Operations:

▪

Adding/Updating: my\_dict[key] = value adds a new key-value pair or updates an existing one.

▪

Getting: Accessing a value using its key (my\_dict[key]).

▪

Deleting: Removing a key-value pair using del my\_dict[key].

▪

Membership: Checking if a key exists using in.

◦

Iterating: How to loop over items (keys, values, or key-value pairs) in a dictionary using a for loop.

--------------------------------------------------------------------------------

Topic 8: Software Errors

•

Kinds of Errors:

◦

Syntax Error: A "grammatical error" in the code (e.g., typos, incorrect indentation, mismatched brackets) that prevents the program from running. Results in an error message.

◦

Runtime Error: An error that occurs while the program is running, causing it to crash if unhandled (e.g., trying to read a non-existent file, dividing by zero). Results in an error message and often a "stack trace." Can typically be handled with try/except blocks.

◦

Logic Error: A flaw in the program's algorithm that causes it to produce incorrect or unexpected results, even if it runs without crashing. Does not always result in an error message.

•

Exceptions and Testing:

◦

Asserts: Statements used to assert that a condition must be true at a certain point in the program; if false, they raise an AssertionError. Useful for exposing logic errors during development.

◦

Test Design Strategies:

▪

Equivalence Partitioning: Dividing inputs into groups that are expected to behave similarly.

▪

Boundary-Value Analysis: Testing inputs at the edges of valid ranges (e.g., minimum, maximum, just inside/outside boundaries).

◦

Exceptions: Objects that represent runtime errors.

◦

Exception Output: Information displayed when an unhandled exception occurs, including the type of error and a stack trace.

◦

Stack Traces: A list of function calls that were active at the moment an exception occurred, showing the path of execution leading to the error. Helps locate the cause of the exception.

◦

Handling Exceptions: Using try and except blocks to gracefully manage runtime errors without crashing the program.

◦

Raising Exceptions: Using the raise keyword to explicitly create and trigger an exception. When an exception is raised, control flow moves up the call stack until caught or the program crashes.

◦

Approaches to Runtime Errors:

▪

EAFP (Easier to Ask Forgiveness than Permission): Attempt an operation and handle potential errors if they occur (using try/except).

▪

LBYL (Look Before You Leap): Check for potential errors before attempting an operation.

--------------------------------------------------------------------------------

Topic 9: Using Modules

•

Importing and Using Modules:

◦

Modules: Python files containing definitions (functions, classes, variables) that can be imported and reused in other programs.

◦

Importing: Use the import statement (e.g., import math) to make a module's definitions available.

◦

Accessing Definitions: Access definitions within an imported module using dot notation (e.g., math.pi, math.sqrt()).

◦

JSON Files: Understand the purpose of JSON (JavaScript Object Notation) files as a common format for data exchange, often used with Python's json module.

•

Third-Party Modules:

◦

Third-Party Packages: Modules or collections of modules developed by the community, providing extended functionality beyond Python's standard library. They are useful for tasks that are not core to Python but are common in development (e.g., web frameworks, data analysis libraries).

◦

pip: The package installer for Python, used to install and manage third-party packages.

--------------------------------------------------------------------------------

Topic 10: Structuring and Documenting Code

•

Structuring Code:

◦

Modular Programming: Breaking down a program into smaller, independent, and reusable components (modules). Benefits include improved organization, readability, reusability, and maintainability.

◦

Creating Modules: Saving Python code in separate .py files allows them to be imported as modules.

◦

Formatting Source Code: Adhering to consistent code formatting guidelines (e.g., using a linter, following PEP 8) improves readability and collaboration.

•

Documenting Code:

◦

Comments: Inline explanations within the code. Good comments explain why code does something, not just what it does. Excessive or redundant comments can be unhelpful.

◦

Docstrings: Multiline strings used to document functions, classes, and modules. They provide a high-level overview of what the code block does, its arguments, and its return values. They are accessible via help().

◦

README Files: Text files (often README.md) that provide a high-level overview of an entire software project, including installation instructions, usage examples, and purpose.

--------------------------------------------------------------------------------

Topic 11: Algorithm Design Strategies

•

Algorithm Design Strategies (Time Complexity):

◦

Approaching Problems: Understand different strategies for breaking down and solving computational problems.

◦

Algorithmic Complexity: Why considering the efficiency (time and space complexity) of an algorithm is important, especially for large datasets. It measures how resource usage scales with input size.

◦

Optimal and Approximate Solutions: Differentiate between optimal solutions (the best possible solution) and approximate solutions (solutions that are "good enough" but not necessarily optimal, often used when optimal is too complex or slow). Understand when an approximate solution is appropriate.

•

Sorting, Searching, Recursion:

◦

Sorting and Searching: Know how sorting (arranging data in order) and searching (finding specific items in data) are used as fundamental techniques to solve problems.

◦

Recursion: A programming technique where a function calls itself to solve smaller instances of the same problem.

▪

Base Cases: The condition(s) that stop the recursion. A base case returns a value without making a recursive call.

▪

Recursive Cases: The part of the function that makes a recursive call to itself, typically with a modified input that moves closer to a base case.

--------------------------------------------------------------------------------