# Programming, Data Structures, and Algorithms in Python: Week 5

This document covers the key topics from Week 5, focusing on robust programming techniques. These include handling runtime errors through exceptions, interacting with the user via standard input and output, managing data with files, and processing text with Python's powerful string methods.

## 1. Exception Handling

Real-world programs must be resilient. They can encounter errors during execution for many reasons: mathematical impossibilities like division by zero, invalid user input causing type conversion failures, attempts to access a list index that is out of bounds, or external issues like trying to open a file that doesn't exist. Python provides a structured and powerful way to manage these runtime errors, known as **exception handling**.

* **Exception:** An error that occurs during the execution of a program. It represents an "exceptional" event that disrupts the normal flow of instructions. If not handled, it will propagate to the top level and cause the program to crash, often with a traceback message.
* **Raising an Exception:** When Python's interpreter detects a runtime error, it creates an exception object and "raises" it, signaling that something has gone wrong. Each exception has a specific type (e.g., ZeroDivisionError, ValueError, IndexError) that inherits from a base Exception class, indicating the precise nature of the problem.
* **Handling an Exception:** The core idea of robust programming is to anticipate and "catch" these exceptions. This allows a program to gracefully recover from an error, provide feedback to the user, or perform cleanup actions, rather than abruptly terminating.

### The try...except Block

The primary mechanism for handling exceptions is the try...except block. This structure cleanly separates the main logic of your code from the error-handling logic.

* **try:** The code that might potentially raise an exception is placed inside the try block. Python will attempt to execute this code as normal.
* **except:** If an exception of a matching type occurs at any point within the try block, Python immediately stops executing the rest of the code in the try block and jumps to the corresponding except block.
  + except ValueError: will only catch exceptions of the type ValueError.
  + except (NameError, KeyError): can handle multiple specified exception types in a single block.
  + A bare except: will catch *any* exception. This should be used sparingly, as it can hide unexpected errors and make debugging difficult. It's generally better to be specific about the errors you intend to handle.
* **else:** This optional block is a powerful feature that is executed only if the try block completes successfully **without raising any exceptions**. This is useful for code that should only run when the initial operation succeeds, separating it from the code that could raise the error in the first place.
* **finally:** This optional block is always executed when leaving the try block, regardless of whether an exception was raised, caught, or not. It is essential for cleanup operations that must happen no matter what, such as closing a file or releasing a network connection.
* **Flow of Control:** If an exception is raised within a function and is not handled by a local try...except block, it **propagates up the call stack** to the function that called it. If it's not handled there, it continues propagating upwards. If the exception reaches the top level of the program without being handled, the program will terminate. This allows for flexible error handling, where a high-level function can manage errors from multiple low-level functions it calls.

### "Positive" Use of Exceptions

Exception handling can also be used as a valid and often more readable coding pattern, sometimes referred to as "It's Easier to Ask for Forgiveness than Permission" (EAFP).

* **Traditional Approach (LBYL - Look Before You Leap):** First, check if a condition is met or a key exists, then proceed. This involves an explicit check before every action.  
  # LBYL: Check if the key exists before trying to use it.  
  if key in my\_dict:  
   my\_dict[key].append(value)  
  else:  
   my\_dict[key] = [value]
* **Exception-Based Approach (EAFP):** Assume the operation will succeed and handle the potential error if it doesn't. This can be more direct and is often considered more "Pythonic," especially if the exceptional case is rare.  
  # EAFP: Just try to append, and handle the error if the key doesn't exist.  
  try:  
   my\_dict[key].append(value)  
  except KeyError:  
   my\_dict[key] = [value]

## 2. Standard Input and Output

Programs often need to interact with the user directly by reading keyboard input and displaying information on the screen.

### Reading from Standard Input

* **input():** This built-in function pauses the program's execution and waits for the user to type something on the keyboard and press the Enter key.
* **Prompt:** You can provide a string as an argument to input() to display a helpful message (a prompt) to the user, clarifying what kind of input is expected, e.g., name = input("Enter your name: ").
* **Return Value:** A crucial point is that input() **always returns a string**. If you expect a number, you must explicitly convert the returned string using a type conversion function like int() or float(). This conversion itself can fail if the user enters non-numeric text, making it a common place to use a try...except ValueError block for robust input validation.

### Writing to Standard Output

* **print():** The print() function is the standard way to display values on the screen.
  + It can take multiple arguments, which it will display separated by spaces by default: print(x, y, z).
  + **sep argument:** Controls the separator string used between arguments. print(x, y, sep='--') will print the values separated by --.
  + **end argument:** Controls what is printed at the very end. By default, this is a newline character (end='\n'). You can change this to a space (end=' ') or an empty string (end='') to prevent the output from advancing to the next line, allowing subsequent print calls to continue on the same line.

### Formatted Output with f-strings and .format()

For more precise control over the layout of your output, you should use string formatting.

* **f-strings (Formatted String Literals):** Introduced in Python 3.6, f-strings are the modern, preferred method. They are concise and readable, allowing you to embed expressions directly inside string literals.  
  name = "Alice"  
  age = 30  
  print(f"User: {name}, Age: {age}")  
  # Output: User: Alice, Age: 30
* **.format() Method:** The older but still powerful way to format strings. It allows you to insert values into a string using placeholders.
  + **Positional Placeholders:**  
    "First: {0}, Second: {1}".format(47, 11)  
    # Output: 'First: 47, Second: 11'
  + **Named Placeholders:**  
    "First: {f}, Second: {s}".format(f=47, s=11)
* **Formatting Specifiers:** Both methods allow you to add a colon (:) inside the placeholder to specify formatting for width, alignment, and type.
  + {value:5d}: Format value as a decimal integer (d) in a field of width 5, right-aligned.
  + {value:^10s}: Format value as a string (s) centered (^) in a field of width 10.
  + {value:8.2f}: Format value as a floating-point number (f) in a field of width 8, with 2 digits of precision after the decimal point.

## 3. Handling Files

For larger volumes of data or for persistent storage, programs must read from and write to files on the disk.

* **File Handle:** A program interacts with a file through a **file handle**, which is an object returned by the open() function that represents the live connection to the file.
* **Buffers:** To improve efficiency, data is not read or written character-by-character from the disk, which is a slow process. Instead, data is transferred in larger, more efficient blocks between the disk and a memory area called a **buffer**. The close() operation is vital because it **flushes** this buffer, ensuring all data is written.

### The Modern File I/O Workflow with with

The recommended way to work with files is using the with statement, which guarantees that the file is automatically closed, even if errors occur.

1. **with open(...) as f::** This syntax establishes a connection to a file and assigns the file handle to the variable f. The subsequent indented block is the context in which the file is open and available.  
   # 'r' for read, 'w' for write, 'a' for append  
   with open("data.txt", "r") as f\_in:  
    content = f\_in.read()  
   # The file is automatically closed here  
   * **Modes:**
     + 'r': Read-only. The file must exist, otherwise a FileNotFoundError is raised.
     + 'w': Write. Creates a new file. **Warning:** If the file already exists, its contents are erased.
     + 'a': Append. Adds new content to the end of an existing file without erasing its current content.
2. **Read/Write Operations:** Use methods on the file handle f to interact with the file.
   * **Reading:**
     + f.read(): Reads the entire file content as a single string. Be cautious with very large files.
     + f.readline(): Reads one line from the file, including the trailing newline character (\n).
     + f.readlines(): Reads all lines into a list of strings, with each string ending in \n.
     + **Iteration:** The most memory-efficient way to process a file line-by-line is to iterate directly over the file handle: for line in f:.
   * **Writing:**
     + f.write(s): Writes the string s to the file.
     + f.writelines(list\_of\_strings): Writes each string from the list to the file. Note that you must add newline characters manually if you want line breaks.
3. **Automatic close():** When the with block is exited, Python automatically calls f.close(), ensuring the file is properly saved and its resources are released.

## 4. String Processing

Python provides a rich set of methods for manipulating strings, which is essential when processing text from files or user input. Remember that strings are **immutable**, so these methods always return a new, modified string; they do not change the original.

* **Whitespace Removal:**
  + s.strip(): Removes leading and trailing whitespace (spaces, tabs, newlines).
  + s.lstrip(): Removes leading (left) whitespace only.
  + s.rstrip(): Removes trailing (right) whitespace only. This is very useful for removing the \n character from lines read from a file.
* **Searching and Replacing:**
  + s.find(pattern): Returns the index of the first occurrence of pattern, or -1 if it is not found.
  + s.startswith(prefix) / s.endswith(suffix): Efficiently check if a string begins or ends with a specific substring.
  + s.replace(old, new): Returns a new string with all occurrences of old replaced by new.
* **Splitting and Joining:**
  + s.split(separator): Splits a string into a list of substrings based on a separator. If no separator is provided, it splits on any whitespace.
  + separator.join(list\_of\_strings): A powerful method that joins a list of strings into a single string, with the separator string inserted between each element.
* **Case Conversion:**
  + s.lower(): Converts the entire string to lowercase.
  + s.upper(): Converts the entire string to uppercase.
  + s.capitalize(): Capitalizes the first letter of the string and lowercases the rest.

## 5. Other Useful Concepts

* **pass statement:** A placeholder statement that does nothing. It is used when the Python syntax requires an indented block but no action is needed, such as in an empty except block or as a placeholder for a function or class that is yet to be implemented.
* **del statement:** Removes an item from a mutable collection like a list or dictionary, or undefines a name, freeing it from its object.
  + del my\_list[i]: Deletes the item at index i, causing subsequent items to shift left.
  + del my\_dict[key]: Deletes the entry (key-value pair) associated with the specified key.
* **None value:** A special, unique singleton object that represents the absence of a value or a null value. It is often used to initialize a variable before it is assigned a meaningful value or as a default return value for functions that do not explicitly return anything. The idiomatic way to check for it is with the is operator: if my\_var is None:.