Python Text File Handling Guide

Python offers robust tools for working with text files through its built-in file handling capabilities. This guide covers the most common methods for reading, navigating, and manipulating text files, with practical examples.

Basic File Operations

Opening and Closing Files

```
# Basic syntax
file = open('filename.txt', 'mode')
# Operations with the file
file.close()

# Recommended approach using context manager
with open('filename.txt', 'mode') as file:
    # Operations with the file
    # File automatically closes when leaving this block
```

Common file modes:

- `'r'` - Read (default)
- `'w'` - Write (creates new file or truncates existing)
- `'a'` - Append
- `'b'` - Binary mode (e.g., 'rb', 'wb')
- `'t'` - Text mode (default)
- `'+'` - Update (read and write)

Reading File Content

Reading Entire File Content

```
file_path = 'example.txt'
# Open the file in read mode
with open(file_path, 'r') as file:
    content = file.read()

# Print the content
print(content)
```

Reading Line by Line

```
file_path = 'example.txt'
# Open the file in read mode
with open(file_path, 'r') as file:
    for line in file:
        print(line.strip())  # strip() removes extra whitespace, including the newline character
```

Reading All Lines into a List

```
file_path = 'example.txt'
# Open the file in read mode
with open(file_path, 'r') as file:
    lines = file.readlines()

# Print the lines
for line in lines:
    print(line.strip())
```

Reading a Specific Number of Characters

```
file_path = 'example.txt'

# Open the file in read mode
with open(file_path, 'r') as file:
    partial_content = file.read(20)  # Reads the first 20 characters

# Print the partial content
print(partial_content)
```

File Navigation with `seek()` and `tell()`

The `seek()` and `tell()` methods are powerful tools for navigating within a file:

Understanding File Position with `tell()`

The `tell()` method returns the current position of the file pointer (as number of bytes from the beginning of the file).

```
file_path = 'example.txt'

with open(file_path, 'r') as file:
    print(f"Initial position: {file.tell()}")  # Should print 0

    # Read 10 characters
    content = file.read(10)
    print(f"Content: {content}")

    # Check position after reading
    print(f"New position: {file.tell()}")  # Should print 10
```

Moving the File Pointer with `seek()`

The `seek(offset, whence)` method changes the current file position:

- `offset`: number of bytes to move
- `whence`: reference point (optional, default is 0)
  - 0: beginning of file
  - 1: current position

- 2: end of file

```
file_path = 'example.txt'

with open(file_path, 'r') as file:
    # Move to the 10th byte from the beginning
    file.seek(10)
    print(f"Position after seek(10): {file.tell()}")
    print(f"Content: {file.read(5)}")

    # Move 5 bytes forward from current position
    file.seek(5, 1)
    print(f"Position after seek(5, 1): {file.tell()}")
    print(f"Content: {file.read(5)}")

    # Move 10 bytes back from the end
    file.seek(-10, 2)
    print(f"Position after seek(-10, 2): {file.tell()}")
    print(f"Content: {file.read()}")  # Read until the end
```

Practical Example: Reading Chunks of a File

```
file_path = 'example.txt'

with open(file_path, 'r') as file:
    chunk_size = 10

    # Read the file in chunks
    while True:
        # Save current position
        position = file.tell()

        # Read a chunk
        chunk = file.read(chunk_size)

        # If no more content, break the loop
        if not chunk:
            break

        print(f"Position: {position}, Chunk: {chunk}")
```

Example: Skipping and Reading Specific Parts

```
file_path = 'example.txt'

with open(file_path, 'r') as file:
    # Skip the first 20 bytes
    file.seek(20)

    # Read the next 10 bytes
    middle_content = file.read(10)
    print(f"Middle content: {middle_content}")

    # Remember current position
    middle_position = file.tell()

    # Jump to the end, then go back 15 bytes
    file.seek(0, 2)  # Go to end
    end_position = file.tell()

    file.seek(-15, 2)  # Go 15 bytes back from end
    end_content = file.read()

    print(f"File size: {end_position} bytes")
    print(f"End content: {end_content}")

    # Return to the middle position we saved
    file.seek(middle_position)
    print(f"Back to position: {file.tell()}")
    print(f"Content from here: {file.read(10)}")
```

Advanced Reading Techniques

Reading and Processing Lines with `enumerate()`

```
file_path = 'example.txt'

# Open the file in read mode
with open(file_path, 'r') as file:
    for line_number, line_content in enumerate(file, 1):
        print(f"Line {line_number}: {line_content.strip()}")
```

Reading a Specific Number of Lines

```
file_path = 'example.txt'

# Open the file in read mode
with open(file_path, 'r') as file:
    num_lines_to_read = 3
    for i in range(num_lines_to_read):
        line = file.readline()
        if not line:  # End of file
            break
        print(f"Line {i+1}: {line.strip()}")
```

Reading a Specific Line

```
def read_specific_line(file_path, target_line_number):
    try:
        with open(file_path, 'r') as file:
            for current_line_number, content in enumerate(file, 1):
                if current_line_number == target_line_number:
                    return content.strip()

            # If we get here, the line number was out of range
            return f"Error: Line {target_line_number} not found"

    except FileNotFoundError:
        return f"Error: File not found at {file_path}"
    except Exception as e:
        return f"An error occurred: {e}"

# Example usage
print(read_specific_line('quotes.txt', 7))
```

Efficient Approach for Reading a Specific Line

Using `seek()` and `readline()` for better performance with large files:

```python
def read_line_efficient(file_path, target_line_number):
    try:
        with open(file_path, 'r') as file:
            # Initialize variables
            current_position = 0
            line_number = 0

            # Skip lines until target
            while line_number < target_line_number - 1:
                # Save current position
                current_position = file.tell()

                # Read a line
                line = file.readline()

                # Check if EOF
                if not line:
                    return f"Error: File has fewer than {target_line_number} lines"

                line_number += 1

            # Now we're at the target line
            target_line = file.readline()
            if target_line:
                return target_line.strip()
            else:
                return f"Error: Line {target_line_number} not found"

    except FileNotFoundError:
        return f"Error: File not found at {file_path}"
    except Exception as e:
        return f"An error occurred: {e}"

# Example usage
print(read_line_efficient('quotes.txt', 7))
```

Reading with Character Sets and Encodings

```
# Specify encoding when opening a file
with open('example.txt', 'r', encoding='utf-8') as file:
    content = file.read()

# For files with different encodings
with open('international.txt', 'r', encoding='latin-1') as file:
    content = file.read()
```

Error Handling in File Operations

```
try:
    with open('nonexistent.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print("The file does not exist")
except PermissionError:
    print("You don't have permission to read this file")
except UnicodeDecodeError:
    print("The file encoding is not compatible")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

Practical Use Cases

Finding Text in a File

```
def find_text_in_file(file_path, search_text):
    try:
        with open(file_path, 'r') as file:
            positions = []
            line_number = 0

            for line in file:
                line_number += 1
                if search_text in line:
                    positions.append((line_number, line.strip()))

            return positions
    except Exception as e:
        return f"Error: {e}"

# Example usage
results = find_text_in_file('example.txt', 'Python')
for line_num, line_content in results:
    print(f"Found at line {line_num}: {line_content}")
```

Binary File Navigation

For binary files, `seek()` and `tell()` work with byte positions:

```python
with open('image.jpg', 'rb') as binary_file:
    # Read the file header (first 10 bytes)
    header = binary_file.read(10)
    print(f"Header bytes: {header}")

    # Jump to a specific offset
    binary_file.seek(1024)  # Jump to 1KB position

    # Read 512 bytes from current position
    data_chunk = binary_file.read(512)

    # Check file size by seeking to the end
    binary_file.seek(0, 2)  # 2 means seek from end
    file_size = binary_file.tell()

    print(f"File size: {file_size} bytes")
```

Best Practices

1. **Always use a context manager (`with` statement)** to ensure files are properly closed.
2. **Handle exceptions** appropriately for robust file operations.
3. **Be mindful of file size** when using `read()` on large files.
4. **Use `seek()` and `tell()`** for efficient navigation in large files.
5. **Specify encoding** when working with non-ASCII text files.
6. **Use binary mode (`'rb'`, `'wb'`)** when working with non-text files.