Python Programming Bootcamp
Week 4: File I/O & Error Handling

Hilal Khan

Center for Artificial Intelligence & Emerging Technologies

Learning Objectives

By the end of this week, you will be able to:

- Understand and handle exceptions gracefully
- Use try-except-else-finally blocks effectively
- Raise exceptions with informative messages
- Read and write files in various formats
- Work with CSV and JSON data
- Use context managers for safe resource handling
- Implement robust error handling in file operations
- Build production-ready file processing applications

Week 4 Overview

What We'll Cover This Week:

Part 1: Error Handling (40%)

- Understanding Exceptions
- Try-Except Blocks
- Else and Finally Clauses
- Raising Exceptions
- Error Handling Best Practices

Part 2: File I/O (60%)

- File Modes and Operations
- Reading and Writing Files
- Context Managers
- CSV File Handling
- JSON File Handling

Quick Recap: Previous Weeks

Week 1:

- Python basics, variables, data types, operators
- Input/output, conditionals, loops

Week 2:

- Data structures: Lists, strings, tuples, dictionaries, sets

Week 3:

- Functions, parameters, return values
- Modules and imports

This week: We'll learn how to handle errors gracefully and work with files to persist data!

Part 1: Error Handling

Understanding and Managing Exceptions

Making Your Programs Robust and User-Friendly

Understanding Exceptions i

What are exceptions?

Exceptions are errors that occur during program execution. Python uses exceptions to signal that something went wrong.

```
# Common exceptions you've probably seen
print(10 / 0)              # ZeroDivisionError
print(undefined_var)       # NameError
numbers = [1, 2, 3]
print(numbers[10])         # IndexError
person = {"name": "Alice"}
print(person["age"])       # KeyError
int("hello")               # ValueError
"text" + 5                 # TypeError
```

Understanding Exceptions ii

Why Handle Exceptions?

- Prevent program crashes
- Provide meaningful error messages to users
- Clean up resources properly (close files, connections)
- Make programs more robust and user-friendly
- Enable debugging and error logging

Try-Except Blocks i

Basic exception handling:

```
# Without exception handling - program crashes
number = int(input("Enter a number: "))  # If user
    enters "abc"
result = 10 / number
print(f"Result: {result}")
```

With exception handling - program continues:

Try-Except Blocks ii

```
# Basic try-except
try:
    number = int(input("Enter a number: "))
    result = 10 / number
    print(f"Result: {result}")
except:
    print("Something went wrong!")
    print("Please try again.")
```

Better: specify exception types:

Try-Except Blocks iii

```
try:
    number = int(input("Enter a number: "))
    result = 10 / number
    print(f"Result: {result}")
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
except ValueError:
    print("Error: Please enter a valid number!")
```

Common Python Exceptions i

Built-in exception hierarchy:

```
# BaseException (root of all exceptions)
#  +-- SystemExit
#  +-- KeyboardInterrupt
#  +-- Exception
#       +-- ArithmeticError
#       |    +-- ZeroDivisionError
#       |    +-- OverflowError
#       +-- LookupError
#       |    +-- IndexError
#       |    +-- KeyError
#       +-- OSError
#       |    +-- FileNotFoundError
#       |    +-- PermissionError
#       +-- ValueError
```

Common Python Exceptions ii

```
#         +-- TypeError
#         +-- NameError
#         +-- AttributeError
```

Common Python Exceptions iii

Common exception examples:

```
# ValueError - invalid value for operation
int("abc")                  # Cannot convert to integer
float("12.34.56")           # Invalid float format

# TypeError - operation on wrong type
"hello" + 5                 # Cannot concatenate str and
    int
len(5)                      # int has no length

# AttributeError - invalid attribute
x = 10
x.append(5)                 # int has no append method

# IndexError - index out of range
lst = [1, 2, 3]
```

Common Python Exceptions iv

```
print(lst[10])              # Index 10 doesn't exist

# KeyError - key not in dictionary
d = {"a": 1}
print(d["b"])               # Key 'b' not found

# FileNotFoundError - file doesn't exist
with open("nonexistent.txt") as f:
    content = f.read()
```

Catching Multiple Exceptions i

Different ways to handle multiple exception types:

```
# Multiple except blocks
try:
    number = int(input("Enter a number: "))
    result = 10 / number
    print(f"Result: {result}")
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
except ValueError:
    print("Error: Please enter a valid number!")
except Exception as e:
    print(f"Unexpected error: {e}")
```

Grouping exceptions:

Catching Multiple Exceptions ii

```
# Handle multiple exceptions the same way
try:
    number = int(input("Enter a number: "))
    result = 10 / number
    print(f"Result: {result}")
except (ZeroDivisionError, ValueError) as e:
    print(f"Invalid input: {type(e).__name__}")
    print("Please enter a valid non-zero number.")
```

Catching Multiple Exceptions iii

Accessing exception information:

```
# Get exception details
try:
    numbers = [1, 2, 3]
    print(numbers[10])
except IndexError as e:
    print(f"Error type: {type(e).__name__}")
    print(f"Error message: {e}")
    print(f"Error args: {e.args}")

# Output:
# Error type: IndexError
# Error message: list index out of range
# Error args: ('list index out of range',)
```

Else and Finally Clauses i

Complete exception handling structure:

```
def divide_numbers(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Error: Division by zero!")
        return None
    except TypeError:
        print("Error: Both arguments must be numbers!"
    )
        return None
    else:
        # Executes ONLY if no exception occurred
        print("Division completed successfully!")
        return result
```

Else and Finally Clauses ii

```
    finally:
        # ALWAYS executes, regardless of exceptions
        print("Cleanup: This always runs")

# Test cases
print(divide_numbers(10, 2))      # Success
print(divide_numbers(10, 0))      # Zero division
print(divide_numbers(10, "2"))    # Type error
```

Else and Finally Clauses iii

Else vs Finally

- **else**: Code that should run only when no exceptions occur
- **finally**: Cleanup code that must run regardless of exceptions

Practical example with file handling:

```
def read_file_safely(filename):
    file_handle = None
    try:
        file_handle = open(filename, 'r')
        content = file_handle.read()
    except FileNotFoundError:
        print(f"Error: {filename} not found!")
        return None
    except PermissionError:
```

Else and Finally Clauses iv

```
        print(f"Error: Permission denied for {filename
    }")
        return None
    else:
        print(f"Successfully read {len(content)} bytes
    ")
        return content
    finally:
        # Always close the file if it was opened
        if file_handle is not None:
            file_handle.close()
            print("File closed")
```

Raising Exceptions i

Manually raising exceptions:

```
# Basic raise statement
def calculate_age(birth_year):
    current_year = 2024
    if birth_year > current_year:
        raise ValueError("Birth year cannot be in the
    future!")
    if birth_year < 1900:
        raise ValueError("Birth year seems unrealistic
    !")
    return current_year - birth_year

# Using the function
try:
    age = calculate_age(2030)
```

Raising Exceptions ii

```
    print(f"Age: {age}")
except ValueError as e:
    print(f"Error: {e}")
```

Raising Exceptions iii

Raising with custom messages:

```
def withdraw(amount, balance):
    if amount <= 0:
        raise ValueError("Amount must be positive!")
    if amount > balance:
        raise ValueError(
            f"Insufficient funds! "
            f"Available: ${balance:.2f}, "
            f"Requested: ${amount:.2f}"
        )
    return balance - amount

# Using the function
try:
    new_balance = withdraw(1000, 500)
    print(f"New balance: ${new_balance:.2f}")
```

Raising Exceptions iv

```
except ValueError as e:
    print(f"Withdrawal failed: {e}")
```

Raising Exceptions v

Re-raising exceptions:

```
def process_data(data):
    try:
        result = int(data)
        return result * 2
    except ValueError:
        print("Warning: Invalid data encountered")
        # Log the error and re-raise
        raise  # Re-raises the same exception

try:
    result = process_data("abc")
except ValueError:
    print("Could not process data")
```

Error Handling Best Practices i

Do's:

```
# DO: Be specific about exceptions
try:
    value = int(user_input)
except ValueError:  # Specific exception
    print("Invalid number format")

# DO: Use else for success code
try:
    result = risky_operation()
except SomeError:
    handle_error()
else:
    # Only runs if no exception
    process_result(result)
```

Error Handling Best Practices ii

```
# DO: Use finally for cleanup
try:
    resource = acquire_resource()
    use_resource(resource)
finally:
    release_resource(resource)  # Always runs
```

Error Handling Best Practices iii

Don’ts:

```
# DON'T: Use bare except
try:
    do_something()
except:  # Bad - catches everything!
    pass

# DON'T: Ignore exceptions silently
try:
    important_operation()
except Exception:
    pass  # Bad - error is hidden!

# DON'T: Use exceptions for flow control
try:
    value = dictionary[key]
```

Error Handling Best Practices iv

```
except KeyError:
    value = default
# Better: value = dictionary.get(key, default)

# DON'T: Catch too broad exceptions early
try:
    do_something()
except Exception:  # Too broad
    handle_generic()
except ValueError:  # Never reached!
    handle_specific()
```

Part 2: File I/O

Working with Files

Reading, Writing, and Processing Data

Introduction to File Handling i

Why work with files?

- Store data permanently (persistence)
- Read configuration files
- Process large datasets
- Save program state
- Export results and reports
- Share data between programs

File types:

- **Text files**: .txt, .csv, .json, .xml, .log
- **Binary files**: .jpg, .pdf, .zip, .exe, .pkl

File Modes i

Understanding file opening modes:

```
# Read modes
'r'   # Read (default) - file must exist
'rb'  # Read binary - for images, PDFs, etc.

# Write modes
'w'   # Write - creates new file or OVERWRITES
    existing
'wb'  # Write binary
'x'   # Exclusive creation - fails if file exists

# Append modes
'a'   # Append - creates new or appends to existing
'ab'  # Append binary
```

File Modes ii

```
# Read and write modes
'r+'  # Read and write - file must exist
'w+'  # Write and read - overwrites file
'a+'  # Append and read - creates if doesn't exist
```

File Modes iii

Mode examples:

```
# Read existing file (error if not found)
file = open('data.txt', 'r')

# Create new file or overwrite
file = open('output.txt', 'w')

# Add to end of file
file = open('log.txt', 'a')

# Read binary file (image, PDF, etc.)
file = open('image.jpg', 'rb')

# Create new file (error if exists)
file = open('new_file.txt', 'x')
```

File Modes iv

Warning
Mode 'w' will DELETE all existing content! Use 'a' to add to existing files.

Reading Files i

Different ways to read file content:

```
# Method 1: read() - read entire file
file = open('example.txt', 'r')
content = file.read()
file.close()
print(content)

# Method 2: read(n) - read n characters
file = open('example.txt', 'r')
first_100_chars = file.read(100)
file.close()
print(first_100_chars)

# Method 3: readline() - read one line
file = open('example.txt', 'r')
```

Reading Files ii

```
first_line = file.readline()
second_line = file.readline()
file.close()
print(first_line)
```

Reading Files iii

```
# Method 4: readlines() - read all lines into list
file = open('example.txt', 'r')
lines = file.readlines()
file.close()
for line in lines:
    print(line.strip())  # strip() removes \n

# Method 5: iterate over file (best for large files)
file = open('example.txt', 'r')
for line in file:
    print(line.strip())
file.close()
```

Important
Always close files after use! Better: use context managers (coming next).

Writing to Files i

Writing data to files:

```
# Write string to file (overwrites)
file = open('output.txt', 'w')
file.write("Hello, World!\n")
file.write("This is line 2.\n")
file.close()

# Append to file
file = open('output.txt', 'a')
file.write("This line was added.\n")
file.close()

# Write multiple lines
lines = [
    "First line\n",
```

Writing to Files ii

```
    "Second line\n",
    "Third line\n"
]
file = open('multiple.txt', 'w')
file.writelines(lines)
file.close()
```

Writing to Files iii

```
# Writing numbers and other data types
numbers = [1, 2, 3, 4, 5]
file = open('numbers.txt', 'w')
for number in numbers:
    file.write(f"{number}\n")  # Convert to string
file.close()

# Writing formatted data
data = [
    ("Alice", 25, "Engineer"),
    ("Bob", 30, "Doctor"),
    ("Charlie", 35, "Teacher")
]
file = open('people.txt', 'w')
for name, age, job in data:
```

Writing to Files iv

```
    file.write(f"{name},{age},{job}\n")
file.close()
```

Remember
write() does NOT add newlines automatically. Use \n explicitly!

Context Managers - The With Statement i

Why use context managers?

```
# Manual file handling (not recommended)
file = open('example.txt', 'r')
try:
    content = file.read()
    # Process content
    print(content)
finally:
    file.close()  # Must remember to close!

# Using context manager (RECOMMENDED)
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
# File automatically closed here!
```

Context Managers - The With Statement ii

Benefits of Context Managers

- Automatic resource cleanup
- File closed even if exceptions occur
- Cleaner, more readable code
- No need for explicit close()
- Prevents resource leaks

Working with multiple files:

Context Managers - The With Statement iii

```
# Read from one file, write to another
with open('input.txt', 'r') as infile, \
     open('output.txt', 'w') as outfile:
    content = infile.read()
    outfile.write(content.upper())
# Both files automatically closed
```

File Error Handling i

Handling file-related errors:

```
# Basic file error handling
try:
    with open('data.txt', 'r') as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("Error: File not found!")
except PermissionError:
    print("Error: Permission denied!")
except IOError as e:
    print(f"Error reading file: {e}")
```

File Error Handling ii

Robust file operations:

```
def safe_read_file(filename):
    """Safely read a file with error handling"""
    try:
        with open(filename, 'r') as file:
            content = file.read()
            return content
    except FileNotFoundError:
        print(f"Error: '{filename}' not found!")
        return None
    except PermissionError:
        print(f"Error: No permission to read '{
    filename}'")
        return None
    except UnicodeDecodeError:
```

File Error Handling iii

```
        print(f"Error: Cannot decode '{filename}' as
    text")
        return None
    except Exception as e:
        print(f"Unexpected error: {e}")
        return None

# Usage
content = safe_read_file("data.txt")
if content is not None:
    print(f"Read {len(content)} characters")
```

Working with CSV Files i

CSV (Comma-Separated Values):

```
import csv

# Sample CSV file: students.csv
# name,age,grade
# Alice,20,A
# Bob,22,B
# Charlie,21,A

# Reading CSV files
try:
    with open('students.csv', 'r') as file:
        csv_reader = csv.reader(file)
        header = next(csv_reader)  # Read header
        print(f"Headers: {header}")
```

Working with CSV Files ii

```
        for row in csv_reader:
            name, age, grade = row
            print(f"{name} is {age} years old, Grade:
    {grade}")
except FileNotFoundError:
    print("CSV file not found!")
```

Working with CSV Files iii

Reading CSV as dictionaries:

```
# DictReader - access columns by name
try:
    with open('students.csv', 'r') as file:
        csv_reader = csv.DictReader(file)

        for row in csv_reader:
            print(f"Name: {row['name']}")
            print(f"Age: {row['age']}")
            print(f"Grade: {row['grade']}")
            print("---")
except FileNotFoundError:
    print("CSV file not found!")
```

Working with CSV Files iv

Writing CSV files:

```
# Writing CSV with writer
data = [
    ['Name', 'Age', 'City'],
    ['Alice', '25', 'New York'],
    ['Bob', '30', 'London'],
    ['Charlie', '35', 'Tokyo']
]

try:
    with open('people.csv', 'w', newline='') as file:
        csv_writer = csv.writer(file)
        csv_writer.writerows(data)
    print("CSV file created successfully!")
except IOError as e:
    print(f"Error writing CSV: {e}")
```

Working with CSV Files v

Working with CSV Files vi

Writing CSV with DictWriter:

```
# Writing CSV with DictWriter
people = [
    {'name': 'Alice', 'age': 25, 'city': 'New York'},
    {'name': 'Bob', 'age': 30, 'city': 'London'},
    {'name': 'Charlie', 'age': 35, 'city': 'Tokyo'}
]

try:
    with open('people_dict.csv', 'w', newline='') as
    file:
        fieldnames = ['name', 'age', 'city']
        csv_writer = csv.DictWriter(file, fieldnames=
    fieldnames)

        csv_writer.writeheader()  # Write header row
```

Working with CSV Files vii

```
            csv_writer.writerows(people)
        print("CSV file created successfully!")
except IOError as e:
        print(f"Error writing CSV: {e}")
```

Working with JSON Files i

JSON (JavaScript Object Notation):

```
import json

# Sample data
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York",
    "hobbies": ["reading", "swimming", "coding"],
    "is_student": True,
    "grades": {
        "math": 95,
        "science": 87,
        "english": 92
    }
```

Working with JSON Files ii

```
}

# Writing JSON to file
try:
    with open('person.json', 'w') as file:
        json.dump(person, file, indent=4)
    print("JSON file created successfully!")
except IOError as e:
    print(f"Error writing JSON: {e}")
```

Working with JSON Files iii

Reading JSON from file:

```
# Reading JSON from file
try:
    with open('person.json', 'r') as file:
        loaded_person = json.load(file)

    print(f"Name: {loaded_person['name']}")
    print(f"Age: {loaded_person['age']}")
    print(f"City: {loaded_person['city']}")
    print(f"Hobbies: {', '.join(loaded_person['hobbies
    ']))}")
    print(f"Math grade: {loaded_person['grades']['math
    ']}")

except FileNotFoundError:
    print("JSON file not found!")
```

Working with JSON Files iv

```
except json.JSONDecodeError:
    print("Error: Invalid JSON format!")
```

Working with JSON Files v

Working with JSON strings:

```
# Convert Python object to JSON string
json_string = json.dumps(person, indent=2)
print("JSON string:")
print(json_string)

# Convert JSON string to Python object
loaded_data = json.loads(json_string)
print(f"\nType: {type(loaded_data)}")
print(f"Name: {loaded_data['name']}")
```

Working with JSON Files vi

JSON vs Python Types

- JSON object $\leftrightarrow$ Python dict
- JSON array $\leftrightarrow$ Python list
- JSON string $\leftrightarrow$ Python str
- JSON number $\leftrightarrow$ Python int/float
- JSON true/false $\leftrightarrow$ Python True/False
- JSON null $\leftrightarrow$ Python None

Practical Example 1: Student Grade Manager i

Complete application with error handling:

```
import json

# Global dictionary to store student data
students_db = {}

def load_data(filename='grades.json'):
    """Load student data from JSON file"""
    try:
        with open(filename, 'r') as file:
            return json.load(file)
    except FileNotFoundError:
        print(f"File {filename} not found. Creating
    new.")
        return {}
```

Practical Example 1: Student Grade Manager ii

```
    except json.JSONDecodeError:
        print("Error: Invalid JSON. Starting fresh.")
        return {}

def save_data(students, filename='grades.json'):
    """Save student data to JSON file"""
    try:
        with open(filename, 'w') as file:
            json.dump(students, file, indent=4)
        return True
    except IOError as e:
        print(f"Error saving data: {e}")
        return False
```

Practical Example 1: Student Grade Manager iii

```
def add_student(students, name, grades):
    """Add a student with their grades"""
    if not name:
        raise ValueError("Student name cannot be empty
    ")
    if not isinstance(grades, list):
        raise TypeError("Grades must be a list")
    if not all(0 <= g <= 100 for g in grades):
        raise ValueError("All grades must be between 0
     and 100")

    students[name] = {
        'grades': grades,
        'average': sum(grades) / len(grades) if grades
     else 0
```

Practical Example 1: Student Grade Manager iv

```
    }
    print(f"Added {name} successfully!")
    return students

def get_student(students, name):
    """Get a student's information"""
    if name not in students:
        raise KeyError(f"Student {name} not found")
    return students[name]

def display_all(students):
    """Display all students"""
    if not students:
        print("No students in database.")
        return
```

Practical Example 1: Student Grade Manager v

```
    print("\n=== All Students ===")
    for name, data in students.items():
        avg = data['average']
        print(f"{name}: Average = {avg:.2f}")
```

Practical Example 1: Student Grade Manager vi

```
# Usage example
def main():
    # Load existing data
    students = load_data('grades.json')

    try:
        # Add students
        add_student(students, "Alice", [95, 87, 92,
    88])
        add_student(students, "Bob", [78, 85, 80, 82])
        add_student(students, "Charlie", [92, 94, 89,
    95])

        # Save data
        save_data(students, 'grades.json')
```

Practical Example 1: Student Grade Manager vii

```
        # Display all students
        display_all(students)

        # Get specific student
        alice_data = get_student(students, "Alice")
        print(f"\nAlice's grades: {alice_data['grades
    ']}")
        print(f"Alice's average: {alice_data['average
    ']:.2f}")

    except ValueError as e:
        print(f"Validation error: {e}")
    except KeyError as e:
        print(f"Lookup error: {e}")
    except Exception as e:
```

Practical Example 1: Student Grade Manager viii

```
        print(f"Unexpected error: {e}")

if __name__ == "__main__":
    main()
```

Practice Exercises

Exercise 1: Safe File Copy

Create a robust file copy utility:

Task
Write a function `safe_copy(source, destination)` that:

- Copies a file from source to destination
- Handles FileNotFoundError, PermissionError, IOError
- Validates that destination doesn't exist (or ask to overwrite)
- Uses context managers
- Returns True on success, False on failure
- Prints informative error messages

Example usage:

```
if safe_copy('data.txt', 'backup.txt'):
    print("File copied successfully!")
else:
    print("Copy operation failed")
```

Exercise 2: Configuration Manager

Build a JSON configuration system using functions:

Task
Create functions to manage configuration:

- `load_config(filename)`: Loads configuration from JSON file
- Creates default config if file doesn't exist
- `get_config(config, key, default)`: Gets value with fallback
- `set_config(config, key, value)`: Sets configuration value
- `save_config(config, filename)`: Saves to JSON file
- Handles JSON decode errors gracefully
- Validates configuration values

Example:

```
config = load_config("app.json")
timeout = get_config(config, "timeout", default=30)
set_config(config, "debug", True)
```

Exercise 3: CSV to JSON Converter

Convert between CSV and JSON formats:

Task
Write functions to:

- Convert CSV file to JSON (list of dictionaries)
- Convert JSON file to CSV
- Handle missing or extra columns
- Validate data types
- Include comprehensive error handling
- Generate conversion reports

CSV format:

```
name,age,city
Alice,25,New York
Bob,30,London
```

Summary & Best Practices

Error Handling Best Practices

Key principles:

Do:

- Be specific about which exceptions to catch
- Use finally for cleanup operations
- Raise exceptions with informative messages
- Log exceptions with context information
- Provide meaningful error messages to users
- Handle exceptions at the appropriate level

Don't:

- Use bare except clauses
- Ignore exceptions silently (empty except pass)
- Catch too broad exceptions unnecessarily
- Use exceptions for normal flow control
- Reveal sensitive information in error messages

File Handling Best Practices

Security and efficiency:

Always:

- Use context managers (with statement)
- Close files explicitly when not using with
- Validate file paths and names
- Handle file exceptions appropriately
- Use appropriate file modes
- Process large files in chunks

Be Careful:

- File permissions and access rights
- User-provided filenames (sanitize!)
- Different line endings (
  n vs
  r

Week 4 Summary

What We Learned:

Part 1: Error Handling

- Understanding exceptions and their types
- Try-except-else-finally blocks
- Raising exceptions with informative messages
- Error handling best practices

Part 2: File I/O

- File modes and operations
- Reading and writing files safely
- Context managers for resource management
- Working with CSV and JSON files
- Combining error handling with file operations

Key Takeaway

Skills Checklist

You should now be able to:

- Handle exceptions gracefully in your code
- Use try-except-else-finally appropriately
- Raise exceptions with informative messages
- Read and write text files safely
- Work with CSV files using csv module
- Work with JSON files using json module
- Use context managers (with statement)
- Implement comprehensive error handling
- Build robust file-based applications
- Validate and process data from files

Next Week Preview

Next Week Preview

Week 5: Object-Oriented Programming Fundamentals

- Introduction to Object-Oriented Programming
- Classes and Objects
- Understanding the `self` keyword
- Instance Attributes and Methods
- Class Attributes and Methods
- Static Methods
- Association: Modeling Relationships
- UML Class Diagrams
- Overview of OOP Properties

Homework

- Complete all practice exercises
- Build a file-based application (e.g., contact manager)
- Practice exception handling in your code
- Experiment with CSV and JSON files

Thank You!

Thank you for your attention!

Keep Practicing!

Access Course Materials:

Download Course Materials

Center for Artificial Intelligence & Emerging Technologies