Fundamentals of Python

PYTHON FILE & EXCEPTION HANDLING

Shruti Bharat

@shrutibharat0105

Table of Contents

File handling

Types of data used for I/O

1. Text
2. Binary

I/O operations on file:

1. Open
2. Read
3. Write
4. Close

Different ways to perform this operation:

1. If the file is not present

```
# Here new file is created in same directory
f = open('sample.txt','w')

# This will give 11 output as it shows how many characters you
# have written in file.
f.write('Hello World')

# file closing
f.close()

# Performing write operation after file close
f.write('Hey')  # error
```

File handling

2. If the file is already present

```
# If file is already present then, below function will overwrite it.
f = open('sample.txt','w')
f.write('Hello World')

# file closing
f.close()
# If you don't close file then it will remain in ram until & unless
# garbage collector removes it.

# Append mode
f = open('sample.txt','a')
f.write('\n Python programming')
f.close()
```

3. Writing multiple lines:

```
L = ['hello \n','Hey \n','Python \n']
f = open('sample.txt','w')
f.writelines(L)
f.close()
```

File handling

4. Reading from files

```
f = open('sample.txt','r')
s = f.read(10)  # Reading only 10 characters
print(s)
f.close()
```

5. Reading entire file using readline:

```
f = open('sample.txt','r')

while True:
  data = f.readline()
  if data == '':
    break
  else:
    print(data,end='')
f.close()
```

File handling

6. Using context manager (with):
   - with keyword closes the file as soon as the usage is over.
   - While using the with keyword you don't have to use f.close()

```
# Reading from a text file
with open("example.txt", "r") as file:
    content = file.read()

# Writing to a text file
with open("output.txt", "w") as file:
    file.write("Hello, world!\n")

# Loading a big file in memory
big_L = ['hello world' for i in range(1000)]
with open("test.txt", "w") as file:
    file.writelines(big_L)

with open("test.txt", "r") as f:
    chunk_size = 10
    while len(f.read(chunk_size)) > 0:
        print(f.read(chunk_size), end='')
        f.read(chunk_size)
```

File handling

7. seek and tell function:

```python
# Seek and Tell
with open("example.txt", "r") as file:
    print(f.read(10))
    print(f.tell())   # 10
    file.seek(5)      # Setting pointer to 5th character
    print(f.read(10))
    print(f.tell())   # 15
```

Problems with working text mode:

- You can’t work with binary files like images.
- It is not suitable for other data types like int, float, list, and tuple.

```python
# Reading from a binary file
with open("binary_file.bin", "rb") as file:
    content = file.read()

# Writing to a binary file
with open("binary_file.bin", "wb") as file:
    file.write(b"Binary data")
```

Serialization & deserialization

Serialization is the process of converting an object into a format that can be stored or transmitted, while deserialization is the reverse process of converting the serialized data back into an object.

```python
import json

# Python object to be serialized
data = {
    "name": "Alice",
    "age": 30,
    "is_student": False,
    "courses": ["Math", "Science"],
    "grades": {"Math": 90, "Science": 95}
}

# Serialization: Convert Python object to JSON string
json_data = json.dumps(data)
print("Serialized JSON string:")
print(json_data)

# Deserialization: Convert JSON string back to Python object
deserialized_data = json.loads(json_data)
print("\nDeserialized Python object:")
print(deserialized_data)
```

Serialization & deserialization

Drawbacks of JSON format:

- JSON supports a limited set of data types (e.g., strings, numbers, lists, dictionaries), which means certain Python objects like sets and tuples cannot be directly serialized.
- JSON may lose precision for floating-point numbers.
- Deserializing data from untrusted sources can be a security risk, as it may lead to code injection vulnerabilities.

Pickling & unpickling:

- Pickling is the process of converting a Python object into a byte stream, and unpickling is the reverse process of converting the byte stream back into a Python object.
- The pickle module in Python is used for serializing and deserializing Python objects to maintain the functionality.

Serialization & deserialization

Pickle Vs JSON:

- Pickle lets you store data in binary format to maintain functionality.
- JSON let's user store data in a human-readable format by default, it is a dictionary.

```python
import pickle

# Define a custom Python class
class Person:
    def __init__(self, name, age, is_student):
        self.name = name
        self.age = age
        self.is_student = is_student

    def __repr__(self):
        return f"Person(name={self.name}, age={self.age}, is_student={self.is_student})"

# Create an instance of the Person class
person = Person("Alice", 30, False)

# Pickling: Serialize the custom Python object to a byte stream and save it to a file
with open('person.pkl', 'wb') as file:
    pickle.dump(person, file)
print("Person object has been pickled and saved to 'person.pkl'.")

# Unpickling: Load the byte stream from the file and deserialize it back to a Python object
with open('person.pkl', 'rb') as file:
    loaded_person = pickle.load(file)
print("\nPerson object has been unpickled from 'person.pkl':")
print(loaded_person)
```

Exception Handling

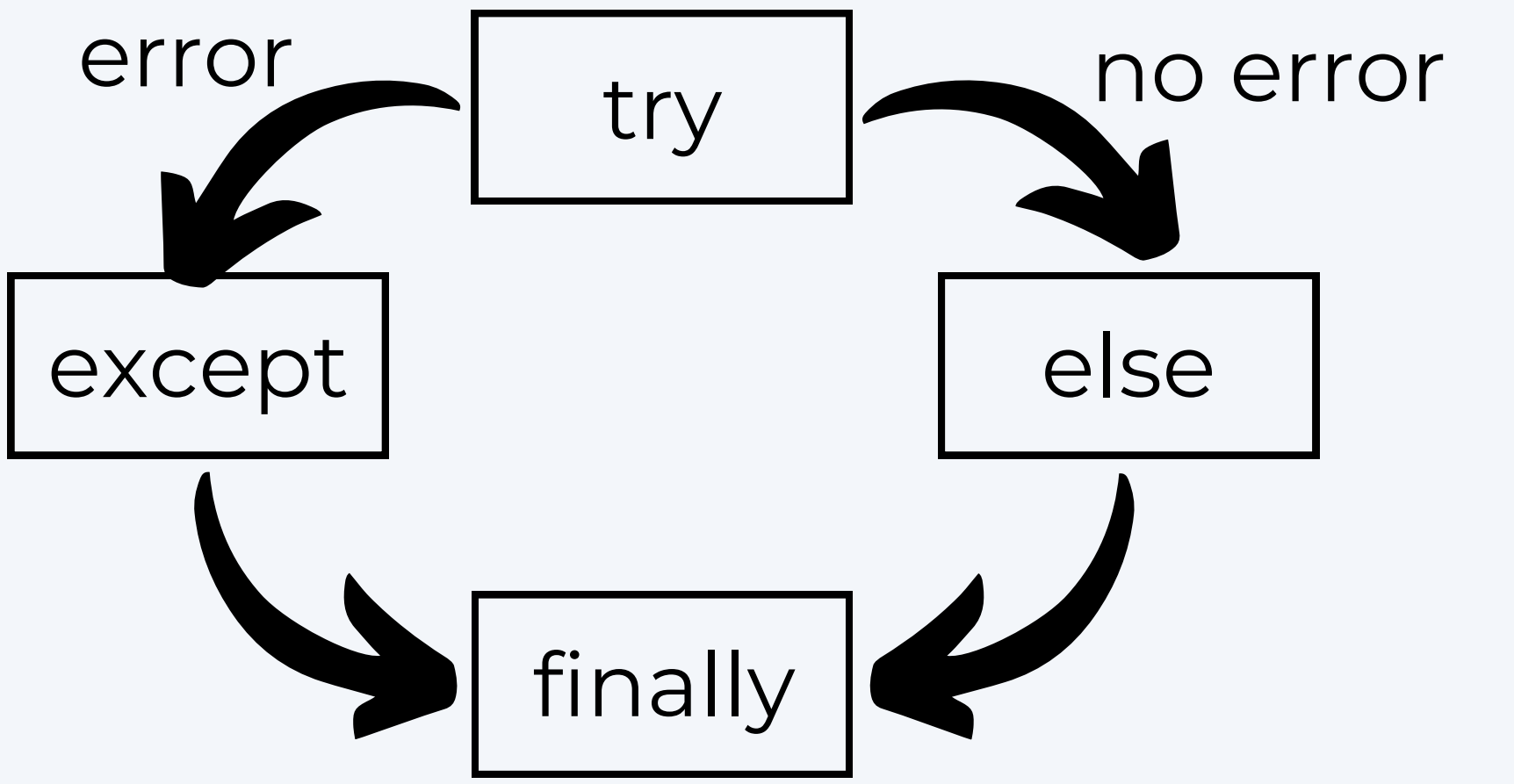
Exception handling in Python allows you to handle errors gracefully and maintain the normal flow of the program.

It uses try, except, else, and finally blocks to catch and handle exceptions.

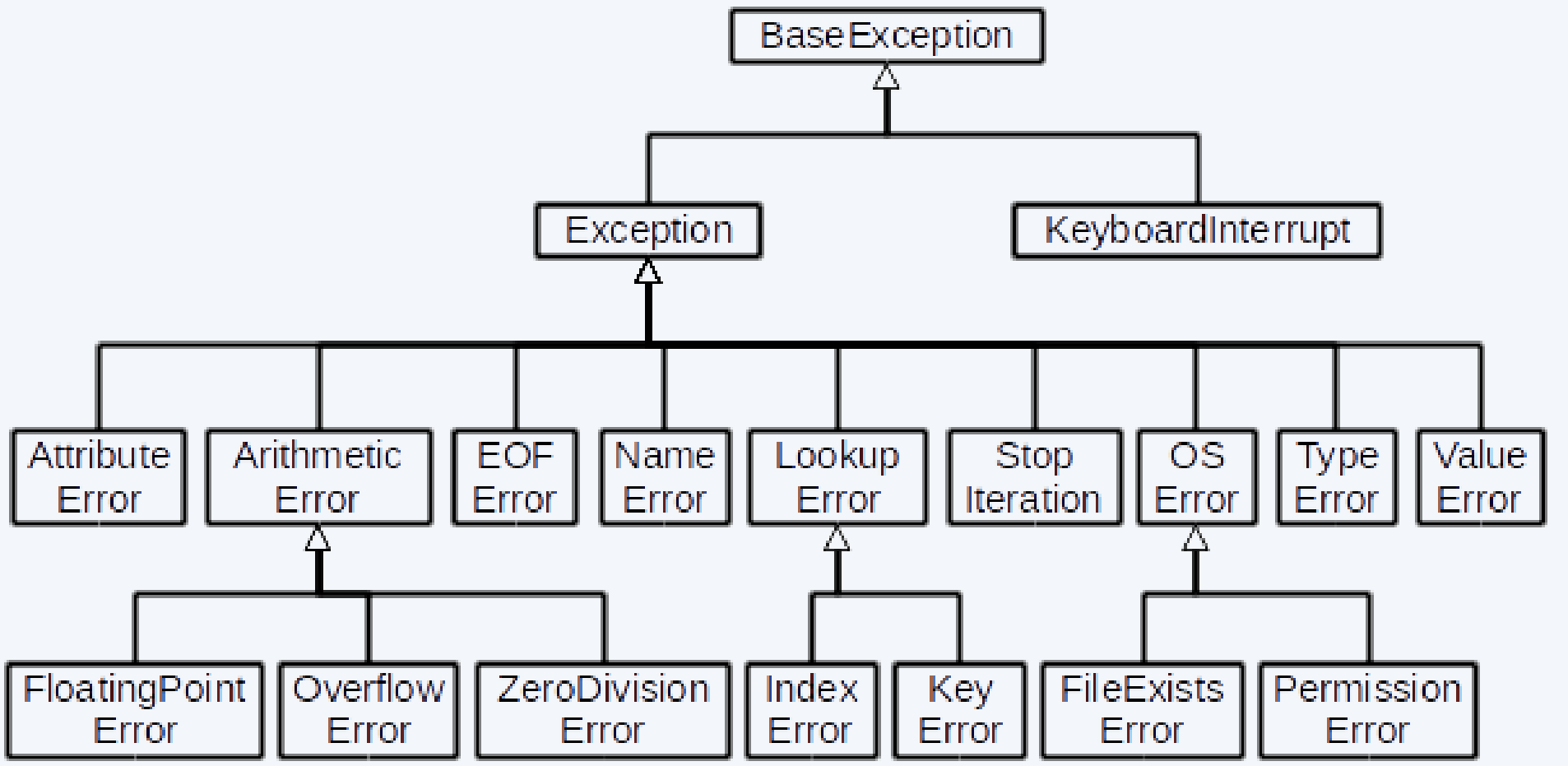
- **Exception:** An error was detected during execution.
- **try block:** Code that may raise an exception is written inside this block.
- **except block**: Code that runs if an exception occurs in the try block.
- **else block:** Code that runs if no exception occurs in the try block.
- **finally block:** Code that runs regardless of whether an exception occurs or not, often used for cleanup actions.

Exception Handling

```python
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError as e:
        print(f"ZeroDivisionError: {e}")
    except TypeError as e:
        print(f"TypeError: {e}")
    except CustomError as e:
        print(f"CustomError: {e}")
    else:
        print(f"The result is {result}")
    finally:
        print("Execution of divide function complete.")

# Function usage examples
# Example 1: Handling ZeroDivisionError
divide(10, 0)

# Example 2: Handling TypeError
divide(10, 'a')

# Example 3: Handling no exception
divide(10, 2)
```



Exception hierarchy



Custom Exception

Python also supports defining custom exceptions for more specific error handling.

```python
class NegativeNumberError(Exception):
    def __init__(self, value):
        self.value = value
        self.message = f"Negative number error: {value} is not allowed."
        super().__init__(self.message)

# Function that uses the custom exception
def check_positive_number(num):
    if num < 0:
        raise NegativeNumberError(num)
    return f"{num} is a positive number."


# Case 1: Negative number input
try:
    result = check_positive_number(-5)
    print(result)
except NegativeNumberError as e:
    print(f"Caught an exception: {e}")
```

Custom Exception

Python also supports defining custom exceptions for more specific error handling.

Used for application based errors.

```python
class NegativeNumberError(Exception):
    def __init__(self, value):
        self.value = value
        self.message = f"Negative number error: {value} is not allowed."
        super().__init__(self.message)

# Function that uses the custom exception
def check_positive_number(num):
    if num < 0:
        raise NegativeNumberError(num)
    return f"{num} is a positive number."


# Case 1: Negative number input
try:
    result = check_positive_number(-5)
    print(result)
except NegativeNumberError as e:
    print(f"Caught an exception: {e}")
```

Iterators

Iteration:

- Anytime you use a loop, explicit or implicitly to go over a group of items is called iteration.
- eg. n = [1,2,3,4]

```
for a in n:
    print(a)
```

Iterator:

- It is an object that allows a programmer to traverse a sequence of data without storing the entire data in memory.
- Iterator stores only 1 item at a time in memory. Performs operation & removes from memory.
- Every iterator is also an iterable.
- Every iterator has both iter function as well as next function.

Iterators

Iterable:

- It is an object on which one can iterate over.
- eg. L = [1,2,3,4]
  type(L)
  type(iter(L))
- Not all iterable are iterators.
- Every iterable has an iter function.

```
class MyNumbers:
    def __init__(self, n):
        self.n = n
        self.i = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.i < self.n:
            result = self.i
            self.i += 1
            return result
        else:
            raise StopIteration

# Create an instance of the class
my_numbers = MyNumbers(5)

# Iterate over the instance
for num in my_numbers:
    print(num)
```

Generators

- It is a simple way of creating iterators.
- It doesn’t have a return statement instead it has a yield keyword that returns the generator object.

```python
def fibonacci(n):
    a, b = 0, 1
    while a < n:
        yield a
        a, b = b, a + b

# Generate Fibonacci numbers up to 100
fib_gen = fibonacci(100)

# Iterate over the generator and print the values
for num in fib_gen:
    print(num, end=" ")
```

- Yield is used in Python generators.
- A generator function is defined just like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return.

Generators

Basis of Comparison	Yield	Return
Functionality	Converts a function into a generator, suspends execution, and resumes from where it left off.	Ends the execution and returns a value to the caller.
Usage	Used in generators to produce a sequence of values over time.	Used to return a value from a function.
Memory Usage	Controls the overhead of memory allocation by storing local variable states.	Does not control memory usage.
Execution	Code written after yield executes in the next function call.	Code written after return does not execute.
Number of Times	Can run multiple times.	Can only run once.
Start	Execution resumes from the last state where the function got paused.	Execution starts from the beginning.

Generators

When to Use Yield Instead of Return

- Use yield when you want to iterate over a sequence, but don’t want to store the entire sequence in memory.
- Use yield when you want to produce a sequence of values over time, rather than computing them at once and sending them back like a list.

```
# Example of yield
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3

for value in simpleGeneratorFun():
    print(value)

# Output: 1 2 3

# Example of return
def fun(n):
    for i in range(n):
        return i

print(fun(5))

# Output: 0
```

Coding problems

- Python program to build flashcard using class in Python

- Shuffle a deck of card with OOPS in Python

- How to create an empty class in Python?

- Student management system in Python

🚀 Ready to Level Up?

CONNECT FOR MORE INSIGHTS AND FUTURE UPDATES!

Shruti Bharat

@shrutibharat0105