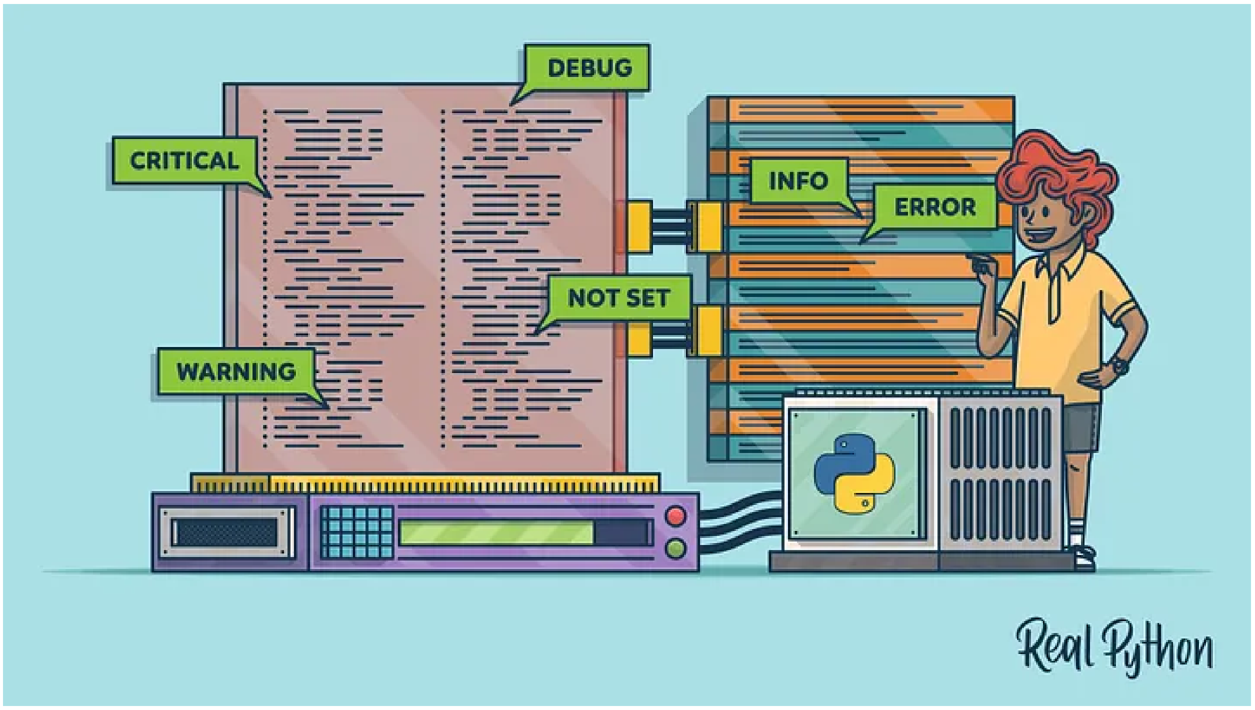
Logging :



Logging vs. Print Statements

Logging and **print statements** both output messages to the console or a file, but they serve different purposes and have key differences.

1. Purpose

- **Print**: Used for simple, temporary debugging or displaying output to users.
- **Logging**: Used for tracking and recording events in your application (especially in production), with more control and flexibility.

2. Control Over Output

- **Print**: Always outputs directly to the console.

- **Logging**: Can be configured to send output to multiple destinations like files, external systems, or the console, with different levels of severity (DEBUG, INFO, WARNING, etc.).

3. Severity Levels

- **Print**: No levels, everything is treated equally.
- **Logging**: Supports different levels, like DEBUG, INFO, WARNING, ERROR, and CRITICAL, allowing you to filter messages based on importance.

4. Configuration

- **Print**: No configuration; it outputs as-is.
- **Logging**: Can be configured with handlers and formatters to control where and how messages are logged.

Using Print:

```
pythonCopy code
print("This is a simple message")
```

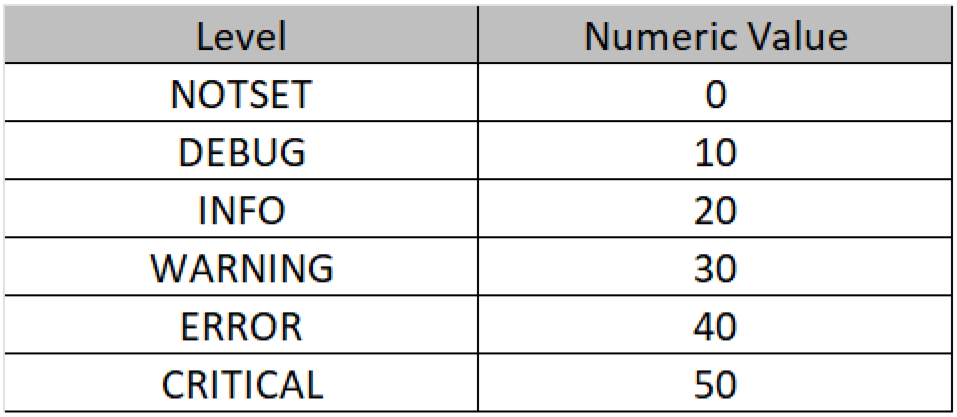
Using Logging:

```
pythonCopy code
import logging

# Configuring the logging system
logging.basicConfig(level=logging.INFO, format='%(levelname)s
- %(message)s')

logging.info("This is an info message")
logging.warning("This is a warning message")
```

Logging Levels :



Level	Numeric Value
NOTSET	0
DEBUG	10
INFO	20
WARNING	30
ERROR	40
CRITICAL	50

Python Logging Levels Overview

1. **DEBUG (Numeric Value: 10)**:
    - **Purpose**: Capture detailed information, typically useful for diagnosing issues.
    - **When to Use**: When you want to see detailed information during development.
    - **Example**: Tracking variable values or program flow.
    - **Message Format**: `DEBUG:MyApp:This is a debug message`
2. **INFO (Numeric Value: 20)**:
    - **Purpose**: Confirm that things are working as expected.

- **When to Use**: To log general events, such as successful execution of functions or user actions.
- **Example**: "User logged in successfully."
- **Message Format**: `INFO:MyApp:This is an info message`

3. **WARNING (Numeric Value: 30)**:
    - **Purpose**: Indicate something unexpected, or warn of potential issues in the near future.
    - **When to Use**: For potential problems that might require attention but don't necessarily interrupt the flow of the program.
    - **Example**: "Disk space running low."
    - **Message Format**: `WARNING:MyApp:This is a warning message`
4. **ERROR (Numeric Value: 40)**:
    - **Purpose**: Record more serious issues that have prevented part of the program from functioning.
    - **When to Use**: When an error occurs that affects part of the application but the program can continue.
    - **Example**: "Failed to open database connection."
    - **Message Format**: `ERROR:MyApp:This is an error message`
5. **CRITICAL (Numeric Value: 50)**:
    - **Purpose**: Indicate severe errors that may prevent the program from continuing to run.
    - **When to Use**: When the program encounters a critical failure and might shut down.
    - **Example**: "Out of memory, shutting down."
    - **Message Format**: `CRITICAL:MyApp:This is a critical message`

Setting Logging Level

When you set a logging level in your program, only messages at or above that level will be recorded. For example, if you set the level to `INFO`, all `INFO`, `WARNING`, `ERROR`, and `CRITICAL` messages will be logged, but `DEBUG` messages will be ignored.

Example: Logging Levels in Practice

```
import logging

# Configure the logging system
logging.basicConfig(level=logging.INFO)  # Set the logging le
vel to INFO

# Create a logger
logger = logging.getLogger('MyApp')

# Log some messages at various levels
logger.debug('This is a debug message')    # Will not be logg
ed because level is set to INFO
logger.info('This is an info message')     # Will be logged
logger.warning('This is a warning message') # Will be logged
logger.error('This is an error message')    # Will be logged
logger.critical('This is a critical message') # Will be logge
d
```

Logging Congiuration:

Key Parameters of `basicConfig()`

Here are the commonly used parameters in the `basicConfig()` function:

1. `level`:

- Specifies the severity level of the root logger. Only messages that are at this level or higher will be logged. This is useful for filtering logs based on their importance.
- Example Values: `logging.DEBUG`, `logging.INFO`, `logging.WARNING`, `logging.ERROR`, `logging.CRITICAL`

2. **`filename`**:
   - Specifies the name of the file where you want to store the logs. If this parameter is set, the logs will be written to the specified file instead of the console.
   - Example Value: `'app.log'`
3. **`filemode`**:
   - Specifies the mode in which the file is opened. The default mode is `'a'` for append. You can set it to `'w'` to overwrite the log file each time the program runs.
   - Example Value: `'w'` (for overwrite) or `'a'` (for append)
4. **`format`**:
   - Defines the format of the log messages. This includes what information is included in the log (e.g., timestamp, log level, logger name, message). The default format is `'%(levelname)s:%(name)s:%(message)s'`.
   - Example Value: `'%(asctime)s - %(name)s - %(levelname)s - %(message)s'`

Example 1:

Let's say you want to log messages to a file called `test.log`, capturing all messages from `INFO` level and above, and customizing the format of the messages.

```
import logging
```

```
logging.basicConfig(
    filename='test.log',
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(messag
e)s', mat
    datefmt='%Y-%m-%d %H:%M:%S',

logger = logging.getLogger('MyApp')


logger.info('This will be written in test.log')
logger.warning('This is a warning')
logger.error('This is an error')
```

File Handler :

In Python's `logging` module, **handlers** are used to send the log records to the appropriate output destinations. You can use multiple handlers for a single logger to send log messages to various places such as files, the console, email, HTTP servers, etc. Handlers are essential in configuring where and how log messages are stored or displayed.

Here are some of the most commonly used handlers in Python:

1. `StreamHandler`

- **Description**: This handler is used to send log messages to streams like `sys.stdout` or `sys.stderr`. By default, it sends logs to the console.
- **Use Case**: Useful for logging to the terminal or console output, such as debugging while developing.

- **Example**:

```
import logging

# Create a logger
logger = logging.getLogger('MyApp')
logger.setLevel(logging.DEBUG)  # Set the log level

# Create a StreamHandler to send logs to the console
stream_handler = logging.StreamHandler()
stream_handler.setLevel(logging.DEBUG)

# Set a format for the handler
formatter = logging.Formatter('%(asctime)s - %(name)s - %
(levelname)s - %(message)s')
stream_handler.setFormatter(formatter)

# Add the handler to the logger
logger.addHandler(stream_handler)

# Log some messages
logger.debug('This is a debug message')
logger.info('This is an info message')
```

  Output:

```
csharpCopy code
2024-08-21 10:30:00 - MyApp - DEBUG - This is a debug mess
age
2024-08-21 10:30:01 - MyApp - INFO - This is an info messa
ge
```

2. `FileHandler`

- **Description**: This handler is used to send log messages to a file. It can be configured to overwrite the file or append to it.
- **Use Case**: Useful for persisting logs to files for future analysis, especially for production systems.
- **Example**:

```
pythonCopy code
import logging

# Create a logger
logger = logging.getLogger('MyApp')
logger.setLevel(logging.WARNING)

# Create a FileHandler to write logs to a file
file_handler = logging.FileHandler('app.log')
file_handler.setLevel(logging.WARNING)

# Set a format for the handler
formatter = logging.Formatter('%(asctime)s - %(name)s - %
(levelname)s - %(message)s')
file_handler.setFormatter(formatter)

# Add the handler to the logger
logger.addHandler(file_handler)

# Log some messages
logger.warning('This is a warning message')
logger.error('This is an error message')
```

Content of `app.log`:

```
vbnetCopy code
2024-08-21 10:35:00 - MyApp - WARNING - This is a warning
message
2024-08-21 10:35:01 - MyApp - ERROR - This is an error mes
sage
```

Logging with Exception:

Example 1: Basic Logging with Exception Handling

You can use `logging.exception()` to log an error along with the stack trace. This is a shorthand for logging the exception message and stack trace when an exception occurs.

```
pythonCopy code
import logging

# Configure logging
logging.basicConfig(filename='app.log', level=logging.ERROR,
                    format='%(asctime)s - %(levelname)s - %(m
essage)s')

# Create a logger
logger = logging.getLogger(__name__)

try:
    # Simulate an error (e.g., division by zero)
    x = 1 / 0
except ZeroDivisionError:
    # Log the exception with stack trace
    logger.exception("An error occurred: Division by zero")
```

Content of `app.log`:

```
vbnetCopy code
2024-08-21 10:45:00 - ERROR - An error occurred: Division by
zero
Traceback (most recent call last):
  File "example.py", line 10, in <module>
    x = 1 / 0
ZeroDivisionError: division by zero
```

Example 2: Using Different Logging Levels in Exception Handling

You can also use different logging levels, such as `ERROR` or `CRITICAL`, to log exceptions based on their severity.

```
pythonCopy code
import logging

# Configure logging
logging.basicConfig(filename='app.log', level=logging.DEBUG,
                    format='%(asctime)s - %(levelname)s - %(m
essage)s')

# Create a logger
logger = logging.getLogger(__name__)

try:
    # Simulate an error (e.g., file not found)
    with open('nonexistent_file.txt', 'r') as f:
        data = f.read()
```

```
except FileNotFoundError as e:
    # Log an error
    logger.error(f"FileNotFoundError occurred: {e}")
except Exception as e:
    # Log a critical error if an unexpected exception occurs
    logger.critical(f"Unexpected error: {e}")
```

Example 4: Logging in Function-Based Exception Handling

You can log exceptions inside functions to capture errors in different parts of your codebase.

```
import logging

# Configure logging
logging.basicConfig(filename='app.log', level=logging.ERROR,
                    format='%(asctime)s - %(levelname)s - %(m
essage)s')

# Create a logger
logger = logging.getLogger(__name__)

def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        logger.exception("ZeroDivisionError in divide functio
n")

divide(10, 0)
```