Python Exception Handling - Cheat Sheet

Basic Syntax

Complete Structure

```
try:
    # Code that might raise exception
    risky_operation()
except SpecificException as e:
    # Handle specific exception
    handle_error(e)
except (Exception1, Exception2) as e:
    # Handle multiple exceptions
    handle_multiple(e)
else:
    # Executes ONLY if no exception occurs
    success_code()
finally:
    # Always executes (cleanup code)
    cleanup()
```

Execution Flow

Block	When It Runs	Skipped If
`try`	Always runs first	-
`except`	Only if matching exception occurs	No exception raised
`else`	Only if NO exception in try	Exception occurred
`finally`	**Always runs**	Never skipped

Order: try → except (if error) → else (if no error) → finally (always)

Common Built-in Exceptions

Exception	Occurs When
`ValueError`	Invalid value for operation
`TypeError`	Wrong data type
`IndexError`	List index out of range

Exception	Occurs When
`KeyError`	Dictionary key not found
`FileNotFoundError`	File doesn't exist
`ZeroDivisionError`	Division by zero
`AttributeError`	Invalid object attribute
`ImportError`	Module import failed
`NameError`	Variable not defined
`IOError`	Input/Output operation failed

Quick Examples

Example 1: Division

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Example 2: File Operations

```
try:
    with open("file.txt", 'r') as f:
        content = f.read()
except FileNotFoundError:
    print("File not found!")
finally:
    print("File operation completed")
```

Example 3: Multiple Exceptions

```
try:
    value = int(input("Enter number: "))
    result = 100 / value
except ValueError:
    print("Invalid input!")
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Custom Exceptions

Basic Pattern

```python
class CustomError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)

# Usage
raise CustomError("Something went wrong!")
```

Advanced Pattern (with attributes)

```python
class InsufficientStockError(Exception):
    def __init__(self, product, requested, available):
        self.product = product
        self.requested = requested
        self.available = available
        msg = f"Need {requested}, only {available} available"
        super().__init__(msg)

# Usage
try:
    raise InsufficientStockError("Laptop", 10, 5)
except InsufficientStockError as e:
    print(e)
    print(f"Product: {e.product}")
```

Exception Hierarchy

```python
class DatabaseError(Exception):
    """Base exception"""
    pass

class ConnectionError(DatabaseError):
    """Specific error"""
    pass

class QueryError(DatabaseError):
    """Specific error"""
    pass
```

Raising Exceptions

Basic Raise

```
if age < 0:
    raise ValueError("Age cannot be negative")
```

Re-raising Exceptions

```
try:
    risky_operation()
except Exception:
    log_error()
    raise  # Re-raises the same exception
```

Exception Chaining

```
try:
    operation()
except ValueError as e:
    raise RuntimeError("Operation failed") from e
```

Best Practices

DO

```
# Be specific with exceptions
try:
    data = json.loads(string)
except json.JSONDecodeError as e:
    print(f"Invalid JSON: {e}")

# Use finally for cleanup
try:
    file = open("data.txt")
    process(file)
finally:
    file.close()

# Keep try blocks minimal
try:
    result = risky_operation()
except SpecificError:
    handle_error()
else:
    process_result(result)
```

DON'T

```
# Don't use bare except
try:
    operation()
except:  # BAD - catches everything
    pass

# Don't ignore exceptions silently
try:
    operation()
except Exception:
    pass  # BAD - silent failure

# Don't catch too broadly
try:
    operation1()
    operation2()
    operation3()
except Exception:  # BAD - which operation failed?
    handle_error()
```

Interview Q&A - Quick Reference

Q1: `except Exception` vs `except:`

Answer: `except Exception` catches most exceptions but not SystemExit/KeyboardInterrupt. Bare `except:` catches everything (discouraged).

Q2: Exception in `finally` block?

Answer: Exception in `finally` suppresses any exception from `try` block.

Q3: Return in `finally`?

```
def test():
    try:
        return "try"
    finally:
        return "finally"  # This wins!

print(test())  # Output: "finally"
```

Q4: `raise` vs `raise e`

- `raise` - Preserves original traceback

- `raise e` - Creates new traceback

Q5: When does `else` execute?

Answer: ONLY when no exception occurs in `try` block.

Exception Hierarchy (Simplified)

```
BaseException
  SystemExit
  KeyboardInterrupt
  GeneratorExit
  Exception
      StopIteration
      ArithmeticError
          ZeroDivisionError
          OverflowError
      AttributeError
      LookupError
          IndexError
          KeyError
      NameError
      OSError
          FileNotFoundError
          PermissionError
      TypeError
      ValueError
```

Advanced Patterns

Context Managers

```python
class DatabaseConnection:
    def __enter__(self):
        self.connect()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.disconnect()
        return False  # Don't suppress exceptions

with DatabaseConnection() as db:
    db.query()
```

Multiple Exception Handlers

```
try:
    operation()
except FileNotFoundError:
    print("File not found")
except PermissionError:
    print("Permission denied")
except OSError as e:
    print(f"OS error: {e}")
except Exception as e:
    print(f"Unexpected error: {e}")
```

Exception Information

```
try:
    risky_operation()
except Exception as e:
    print(f"Type: {type(e).__name__}")
    print(f"Message: {str(e)}")
    print(f"Args: {e.args}")
```

Real-World Use Cases

API Client

```
class APIError(Exception): pass
class AuthError(APIError): pass
class RateLimitError(APIError): pass

try:
    api.call()
except AuthError:
    refresh_token()
except RateLimitError:
    wait_and_retry()
```

File Processing

```
class FileSizeError(Exception):
    def __init__(self, size, max_size):
        self.size = size
        self.max_size = max_size
        super().__init__(f"File too large: {size}MB > {max_size}MB")

def process_file(file):
```

```
    if file.size > MAX_SIZE:
        raise FileSizeError(file.size, MAX_SIZE)
```

Database Operations

```
try:
    db.connect()
    db.execute(query)
    db.commit()
except ConnectionError:
    log("Connection failed")
except QueryError:
    db.rollback()
    log("Query failed, rolled back")
finally:
    db.close()
```

Pro Tips

1. **Catch specific exceptions first**, then general ones
2. **Use `else`** to separate success code from error-prone code
3. **Always use `finally`** for cleanup (or use context managers)
4. **Create custom exceptions** for domain-specific errors
5. **Log exceptions** with context for debugging
6. **Don't catch exceptions** you can't handle properly
7. **Re-raise exceptions** after logging if needed
8. **Use exception chaining** to preserve error context

Common Patterns Cheat

Pattern 1: Try-Except-Else-Finally

```
try:
    resource = acquire()
except AcquisitionError:
    handle_acquisition_error()
else:
    process(resource)
finally:
    cleanup()
```

Pattern 2: Exception Chaining

```
try:
    low_level_operation()
except LowLevelError as e:
    raise HighLevelError("Failed") from e
```

Pattern 3: Suppressing Exceptions

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove("file.txt")
```

Pattern 4: Custom Exception with Data

```
class ValidationError(Exception):
    def __init__(self, field, value, reason):
        self.field = field
        self.value = value
        self.reason = reason
        super().__init__(f"{field}={value}: {reason}")
```

Key Takeaways

Concept	Remember
Specificity	Catch specific exceptions, not generic `Exception`
Cleanup	Use `finally` or context managers
else Block	Runs ONLY when no exception occurs
Bare except	Never use `except:` without exception type
Re-raise	Use `raise` to preserve traceback
Custom Exceptions	Inherit from `Exception`, add attributes
Error Messages	Make them clear, actionable, and informative

Quick Syntax Reference

```
# Basic
try: code()
except Error: handle()

# Multiple
try: code()
```

```
except (E1, E2): handle()

# With variable
try: code()
except Error as e: print(e)

# Full structure
try: code()
except E1: handle1()
except E2: handle2()
else: success()
finally: cleanup()

# Raise
raise ValueError("message")

# Re-raise
except Error: raise

# Chain
except E1 as e: raise E2() from e

# Custom
class MyError(Exception): pass
```

Note: This cheat sheet covers essential exception handling concepts. For production code, always add proper logging, user-friendly messages, and appropriate error recovery mechanisms.

Last Updated: November 2024 | Python 3.x