Python Exceptions — Detailed Revision Notes

1. Exception model (what happens when an error occurs)

- An **exception** is an **object** Python creates when something goes wrong at runtime.
- When raised, the interpreter:
  1. Creates an exception object (type + message + traceback).
  2. **Unwinds the call stack** searching for an `except` block that matches the exception type.
  3. If a matching `except` is found, that handler runs; otherwise Python prints a traceback and the program exits.
- Key exception object attributes (useful for debugging):
  - `exc.args` — arguments passed to the exception.
  - `exc.__traceback__` — a traceback object (stack frames at the time of the exception).
  - `exc.__context__` and `exc.__cause__` — relate to exception chaining.
- **Propagation**: if a function doesn't catch an exception, it bubbles up to the caller, and so on.

2. Exception hierarchy (simplified)

Python exceptions are classes. Most user exceptions inherit from `Exception`. Don't catch `BaseException` unless you truly mean to catch system-exiting exceptions.

```
BaseException
├─ SystemExit
├─ KeyboardInterrupt
├─ GeneratorExit
└─ Exception
   ├─ StopIteration
   ├─ ArithmeticError
   │  ├─ FloatingPointError
   │  ├─ OverflowError
   │  └─ ZeroDivisionError
   ├─ LookupError
   │  ├─ IndexError
   │  └─ KeyError
   ├─ ValueError
   ├─ TypeError
   ├─ ImportError
   │  └─ ModuleNotFoundError
   ├─ OSError
   │  ├─ FileNotFoundError
```

```
│   └─ PermissionError
└─ RuntimeError
```

3. Basic handling: try, except, else, finally

```python
try:
    risky_code()
except ValueError as e:
    handle_value_error(e)
except (TypeError, KeyError) as e:
    handle_other(e)
else:
    success_path()
finally:
    cleanup()
```

4. Handling multiple exceptions

```python
# - Catch different exceptions with multiple `except` blocks:
try:
    ...
except ValueError:
    ...
except IndexError:
    ...
# - Or catch several in one line with a tuple:
except (ValueError, TypeError) as e:
    print("value/type error:", e)
```

- **Order matters**: put more specific exceptions before more general ones.

5. Re-raising and exception chaining

```python
# - Re-raise the current exception:
try:
    ...
except Exception:
    log()
    raise
# - Chaining with `from` preserves original cause:
try:
    x = int("bad")
except ValueError as e:
    raise RuntimeError("parsing failed") from e
```

6. Raising exceptions (raise) and assertions (assert)

```
# - Use `raise` to signal an error condition you detect:
if amount > balance:
    raise ValueError("Insufficient funds")
# - Use `assert` for debugging invariants *only* (not for input validation).
assert n > 0, "n must be positive"
```

7. Custom exceptions (when & how)

- Create a custom exception by subclassing `Exception`:

```
class TooManyAttemptsError(Exception):

    pass
```

8. Patterns & idioms

- **EAFP**** vs **LBYL** approaches.
- Keep `try` blocks small.
- Use `with` for cleanup instead of finally.
- Use `contextlib.suppress` to ignore certain exceptions.

9. Debugging & logging exceptions

- Use `traceback.print_exc()` to print stack trace.
- Use `logging.exception()` inside `except` blocks for production.

10. Real examples

```
### Example: propagate vs handle
def read_int_from_file(path):
    with open(path) as f:
        return int(f.read().strip())

try:
    value = read_int_from_file("n.txt")
except FileNotFoundError:
    value = 0
except ValueError:
    raise RuntimeError("Invalid file contents")
```

11. Common pitfalls & gotchas

- Avoid bare `except:` unless truly intended.
- Don’t catch `Exception` too broadly.

- Don’t use `assert` for user input validation.
- Avoid swallowing exceptions silently with `pass`.

12. Quick reference table

- Catch specific: `except ValueError:`
- Catch many: `except (ValueError, TypeError) as e:`
- Re-raise: `raise`
- Chain: `raise MyError() from e`
- Cleanup: `finally` or `with`
- Log: `logger.exception("msg")`
- Custom: `class MyError(Exception): pass`

13. Cheat sheet

```
class MyError(Exception):
    pass

try:
    x = int(input("Number: "))
except ValueError as e:
    print("Not a number:", e)
else:
    print("Good:", x)
finally:
    print("Always runs")
```

14. Suggested exercises

1. Write a function to open a file, return int(value), or None if missing.
2. Create a `ValidationError` and use it in a validator.
3. Implement a retry decorator for certain exceptions.