Error Handling

Python-এ exception handling হলো **রuntime সমস্যা (errors) কে সুন্দরভাবে ধরা, লগ করা, ও প্রতিক্রিয়া জানানোর** উপায়। AI/ML প্রজেক্টে ফাইল-ই/ডেটা-পর্সিং/মডেল-লোড/IO ও নেটওয়ার্ক অপারেশনে এটা একদম অপরিহার্য। এখানে আমি ধাপে ধাপে সিনট্যাক্স, গুরুত্বপূর্ণ কনসেপ্ট, উদাহরণ, best-practices ও AI/ML-স্পেসিফিক টিপস দেবো।

Syntax — বেসিক প্যাটার্ন

```
try:
    # ঝুঁকিপূর্ণ কোড
    result = risky_operation()
except SomeException as e:
    # exception ধরলে এখানে আসে
    handle(e)
else:
    # try ব্লকে কোনো exception না হলে এখানে চলে
    post_success(result)
finally:
    # সবসময় চালানো হয় – cleanup (resource release) এর জন্য
    cleanup()
```

- try — সম্ভাব্য error-উত্পন্ন কোড।
- except — কোন exception ধরবে (মাল্টিপল ধরা যায়)।
- else — try সফল হলে চলে (except ছাড়া)। else রাখলে ঐ লজিককে try থেকে আলাদা রাখে।
- finally — সবসময় চলে; resources (ফাইল/কনেকশন) বন্ধ করার জন্য উপযোগী।

ধরনভিত্তিক except

```
try:
    x = int(s)
except ValueError:
    print("invalid int")
except (TypeError, KeyError) as e:
    print("type/key problem:", e)
except Exception as e:
    # catch-all (but সতর্কতার সঙ্গে)
    print("unexpected:", e)
```

নোট: সর্বোচ্চ নির্দিষ্ট → সাধারণ ক্রমে লিখো। except Exception: হল catch-most; except: (বিনা শ্রেণি) কখনো ব্যবহার কোরো না — এটা SystemExit, KeyboardInterrupt ইত্যাদি ধরতেও পারে।

Raise / re-raise / chaining

- কোনো পরিস্থিতি নিজেরা exception raise করা যায়:

```
if not valid:
    raise ValueError("invalid input")
```

- re-raise — ধরার পর আবার উপরে পাঠাতে:

```
try:
    do()
except Exception as e:
    log(e)
    raise   # preserves original traceback
```

- chaining (cause নির্দিষ্ট করা):

```
try:
    json.loads(s)
except json.JSONDecodeError as e:
    raise MyParseError("bad json") from e
```

Context managers + with → নিরাপদ resource handling

with ব্লক ব্যবহার করে অনেক ক্ষেত্রে try/finally নিজে না লিখেই cleanup পাওয়া যায়:

```
with open("file.csv") as f:
    df = pd.read_csv(f)
# file closed automatically
```

নিজস্ব contextmanager বানাতে contextlib.contextmanager বা class with __enter__/__exit__ লিখতে হয় — __exit__-এ exception handling করে cleanup করা যায়।

Useful tools / patterns

1) Logging exceptions

```
try:
    risky()
except Exception:
    pass  # silently ignore — dangerous
```

2) Catch specific and fail fast

Bad:

```
try:
    risky()
except Exception:
    pass  # silently ignore — dangerous
```

2) Catch specific and fail fast

Bad:

```
try:
    risky()
except Exception:
    pass  # silently ignore — dangerous
```

Good:

```
try:
    risky()
except FileNotFoundError as e:
    logging.error(...)
    raise
```

3) Retrying transient errors (simple retry decorator)

```
import time
from functools import wraps

def retry(times=3, delay=1, exceptions=(Exception,)):
    def deco(f):
        @wraps(f)
        def wrapper(*a, **kw):
            last_exc = None
            for i in range(times):
                try:
                    return f(*a, **kw)
                except exceptions as e:
                    last_exc = e
                    time.sleep(delay)
            raise last_exc
        return wrapper
    return deco

@retry(times=3, delay=2, exceptions=(ConnectionError,))
def download_data():
    ...
```

4) contextlib.suppress — নির্দিষ্ট exception নীরবভাবে উপেক্ষা

```
from contextlib import suppress
with suppress(FileNotFoundError):
    os.remove("maybe_missing.tmp")
```

5) Validate inputs early (EAFP vs LBYL)

- Pythonic হল **EAFP** (Easier to Ask Forgiveness than Permission): চেষ্টা করে, ব্যর্থ হলে catch করো।
- LBYL (Look Before You Leap) মাঝে মাঝে ব্যবহারযোগ্য— কিন্তু race condition বা expensive checks এ এড়ানো উচিত।

Custom Exceptions (হায়ারার্কি)

প্রজেক্টে নিজের ব্যতিক্রম টাইপ বানাও — যাতে caller সহজে ধরতে পারে:

```
class MLProjectError(Exception):
    """base for project errors"""

class DataLoadError(MLProjectError):
    pass

class ModelSaveError(MLProjectError):
    pass
```

তারপর:

```
raise DataLoadError("missing column 'age'")
```

এভাবে external code except MLProjectError: ধরলেই project-specific errors পাবে।

AI/ML-স্পেসিফিক উদাহরণসমূহ

Example 1 — CSV লোডিং robustly

```python
import pandas as pd
import logging

def load_dataset(path):
    try:
        df = pd.read_csv(path)
    except FileNotFoundError as e:
        logging.error("Dataset not found: %s", path)
        raise
    except pd.errors.EmptyDataError:
        logging.error("CSV empty: %s", path)
        return pd.DataFrame()  # fallback
    except pd.errors.ParserError as e:
        logging.exception("Parse error")
        raise DataLoadError("corrupt csv") from e
    else:
        return df
```

Example 2 — model load with fallback

```python
def load_model(path):
    try:
        return joblib.load(path)
    except FileNotFoundError:
        logging.warning("No saved model; train new one")
        return train_new_model()
    except Exception as e:
        logging.exception("Model load failed")
        raise ModelSaveError("can't load model") from e
```

Example 3 — GPU error handling (PyTorch)

```
import torch
try:
    model.to("cuda")
except RuntimeError as e:
    logging.warning("CUDA not available, using CPU")
    model.to("cpu")
```

Debugging tips & best practices

1. **Catch specific exceptions — except ValueError better than except Exception.**
2. **Don't swallow exceptions silently — always log or reraise. Silent failures are worst.**
3. **Use logging.exception() inside except to record traceback.**
4. **Use raise (no args) to re-raise preserving traceback when you only want to log.**
5. **Add context when raising: raise MyError("context") from e helps debugging.**
6. **Avoid using exceptions for normal control flow — exceptions are for *exceptional* conditions.**
7. **Keep try-block small: only wrap the statement that can fail — এতে traceback স্পষ্ট হয়।**

```
# bad: try covers too much
try:
    do_a()
    do_b()
    do_c()
except Exception: ...
# better: separate try blocks per risky call
```

8. **Test exception paths — write unit tests that simulate failures (mock IO, mock network).**
9. **For long-running ML jobs use checkpoints and robust exception handling to resume.**
10. **Use context managers for resource cleanup instead of bare try/finally when possible.**

When to use else?

else ভালো যখন try অংশ শুধুমাত্র ঝুঁকিপূর্ণ কল করা এবং পরে যদি সফল হয় তখন কিছু করতে চাও। এতে exception-হ্যান্ডলিং ও সফল-লজিক আলাদা থাকে — পড়তে সহজ হয়।

```
try:
    data = load()
except Exception as e:
    handle(e)
else:
    process(data)   # runs only if no exception
```

Small checklist — Quick rules

- **Catch specific exceptions.**

- **Log exceptions with logging.exception.**
- **Prefer with for resources.**
- **Use raise ... from ... when wrapping exceptions.**
- **Avoid bare except: and avoid silencing exceptions.**
- **Keep try blocks minimal.**
- **Write unit tests for failure cases.**
- **Use retry for transient network/file errors.**
- **Create custom exception classes for your domain.**

Short practical snippet (AI/ML pipeline example)

```
def run_pipeline(cfg_path):
    try:
        cfg = load_json(cfg_path)
    except FileNotFoundError:
        logging.error("Config missing")
        return
    try:
        df = load_dataset(cfg["data_path"])
        X, y = preprocess(df)
        model = load_model(cfg["model_path"])
    except DataLoadError as e:
        logging.exception("Data problem")
        return
    except ModelSaveError:
        logging.exception("Model problem; aborting")
        return

    try:
        preds = model.predict(X)
    except Exception as e:
        logging.exception("Prediction failed")
        raise
    else:
        evaluate(preds, y)
    finally:
        logging.info("Pipeline finished")
```