Chapter 35. Exception Objects

So far, this book has been somewhat vague about what an exception actually *is*. This chapter clears up the mystery by disclosing the facts behind exception objects—both built-in and user-defined. As suggested in the preceding chapters, exceptions are identified by *class instance objects*. This is what is raised and propagated along by exception processing, and the source of the class matched against `except` clauses in `try` statements.

Although this means you must use object-oriented programming to define new exceptions in your programs—and introduces a knowledge dependency lamented in the prior chapter's note—basing exceptions on classes and OOP offers a number of benefits. Among them, class-based exceptions support:

Flexible exception categories

Exception classes allow code to choose specificity and ease future changes. Adding new exception subclasses, for example, need not require changes in `try` statements.

State information and behavior

Exception classes provide a natural place to store context for use in the `try` handler. Both attributes and methods, for example, are available on the raised instance.

Reuse by inheritance

Exceptions classes can participate in inheritance hierarchies to obtain and customize common behavior. Inherited error displays, for example, can provide a common look and feel.

Because of these advantages, class-based exceptions support program evolution and larger systems well. As you'll learn here, all built-in exceptions are identified by classes and are organized into an inheritance tree for the reasons just listed. You can do the same with user-defined exceptions of your own.

In fact, the built-in exceptions we'll study here turn out to be integral to new exceptions you define. Because Python largely requires user-defined exceptions to inherit from built-in exception classes that provide useful defaults for printing and state, the task of coding user-defined exceptions also involves understanding the roles of these built-ins.

Exception Classes

Whether built-in or user-defined, exceptions work much of their magic by *superclass relationships*: a raised exception matches an `except` clause if that clause names the exception's class or any superclass of it. Put another way, a `try` statement's `except` matches both the class it lists, as well as all of that class's subclasses lower in the class tree.

The net effect is that class exceptions naturally support the construction of exception *hierarchies*: superclasses become *category* names, and subclasses become *specific* kinds of exceptions within a category. By naming a general exception superclass, an `except` clause can catch an entire category of exceptions—any more specific subclass will match.

In addition to this category idea, class-based exceptions support exception state information and allow exceptions to inherit common behaviors, as noted. To see how all these assets come together in code, let's turn to an example.

Coding Exceptions Classes

In the file listed in Example 35-1, *categoric.py*, we define a superclass called `General` and two subclasses called `Specific1` and `Specific2`. This example illustrates the notion of exception categories— `General` is a category name, and its two subclasses are specific types of exceptions within the category. Handlers that catch `General` will also catch any subclasses of it, including `Specific1` and `Specific2`.

Example 35-1. categoric.py

```
class General(Exception): pass
class Specific1(General): pass
class Specific2(General): pass

def raiser0():
```

```
    X = General()            # Raise superclass instanc
    raise X

def raiser1():
    X = Specific1()          # Raise subclass instance
    raise X

def raiser2():
    X = Specific2()          # Raise different subclass
    raise X

for func in (raiser0, raiser1, raiser2):
    try:
        func()
    except General:          # Match General or any sub
        import sys
        print('caught:', sys.exc_info()[0])
```

When this example runs, its `try` statement catches and reports instances of all three of its classes because the `except` clause names their common superclass:

```
$ python3 categoric.py
caught: <class '__main__.General'>
caught: <class '__main__.Specific1'>
caught: <class '__main__.Specific2'>
```

This code is mostly straightforward, but here are a few points to notice:

Exception superclass

Classes used to build exception category trees have very few requirements—in fact, in this example, they are mostly empty, with bodies that do nothing but `pass`. Notice, though, how the top-level class here inherits from the built-in `Exception` class. This is required: classes that don't inherit from a built-in exception class won't work in most exception contexts. The built-in superclass is normally `Exception`, the root for nonexit exceptions, but may also be `BaseException`, the root for all exceptions, or other. Although we don't employ it here, `Exception` provides behavior you'll meet later that makes inheriting from it useful, required or not.

Raising instances

In this code, we call classes to make *instances* for the `raise` statements (notice the parentheses). In the class exception model, we always raise and catch a class instance object. If we list a class name without parentheses in a `raise`, Python makes an instance for us by calling the class with no constructor arguments. Exception instances can be created before the `raise`, as done here, or within the `raise` statement itself.

Catching categories

This code includes functions that raise instances of all three of our classes as exceptions, as well as a top-level `try` that calls the functions and catches `General` exceptions. The same `try` also catches the two specific exceptions because they are subclasses of `General`—that is, members of its category.

Exception details

The exception handler here uses the `sys.exc_info` call, which is one way to fetch the exception being handled in a generic fashion. As you'll see in more detail in the next chapter, the first item in this call's result tuple is the class of the exception raised, and the second is the actual instance raised. In a general `except` clause like the one here that catches all classes in a category, `sys.exc_info` can be used to determine exactly what has occurred.

The last point merits elaboration. When an exception is caught, we can be sure that the instance raised is an instance of the class listed in the `except` or one of its subclasses. Because of that, the specific kind of exception raised can also be had via the `type` result or `__class__` attribute of the instance, regardless of how the instance is obtained.

To demo, the variant in Example 35-2 works the same as the prior example but uses the `as` extension in its `except` clause to directly assign a variable to the instance raised, from which `type` yields the exception's kind.

Example 35-2. categoric2.py

```
class General(Exception): pass
class Specific1(General): pass
class Specific2(General): pass

def raiser0(): raise General()
def raiser1(): raise Specific1()
def raiser2(): raise Specific2()
```

```
for func in (raiser0, raiser1, raiser2):
    try:
        func()
    except General as X:                # X is the rai
        print('caught:', type(X))       # Same as sys.
```

Because the `except`'s `as` can be used to access the exception directly this way, `sys.exc_info` is more useful for *empty* `except` clauses that do not otherwise have a way to access the instance or its class. More importantly, well-designed programs usually should *not have to care* about which specific exception was raised at all—calling methods of the exception instance should automatically dispatch to behavior tailored for the exception raised.

There's more on this and `sys.exc_info` and its ilk in the next chapter. Also, see Chapter 29 and Part VI at large if you've forgotten what `__class__` means in an instance, and the prior chapter for a review of the `as` used here.

Why Exception Hierarchies?

Because there are only three possible exceptions in the prior section's examples, it doesn't really do justice to the utility of class exceptions. In principle, we could achieve the same effects by coding a list of exception names in a parenthesized tuple within the `except` clause:

```
try:
    func()
except (General, Specific1, Specific2):     # Catch any
    …
```

This approach may work for smaller, self-contained code. For large or high exception hierarchies, however, it will probably be easier to catch categories using class-based categories than to list every member of a category in a single `except` clause. Perhaps more importantly, you can extend exception hierarchies as software needs evolve by adding new subclasses without breaking existing handler code.

Suppose, for example, you code a numeric programming library in Python to be used by a large number of people. While you are writing your library, you identify two things that can go wrong with numbers in your code—division by zero and numeric overflow. You document these as the two standalone exceptions that your library may raise:

```
# mathlib.py
class Divzero(Exception): pass
class Oflow(Exception): pass

def func():
    …
    raise Divzero()
…and so on…
```

Now, when people use your library, they typically wrap calls to your functions or classes in `try` statements that catch your two exceptions; after all, if they do not catch your exceptions, exceptions from your library will kill their code:

```
# client.py
import mathlib

try:
    mathlib.func()
except (mathlib.Divzero, mathlib.Oflow):
    …handle and recover…
```

This works fine, and lots of people start using your library. Six months down the road, though, you revise it (as programmers are prone to do). Along the way, you identify a new thing that can go wrong—underflow, perhaps—and add that as a new exception:

```
# mathlib.py
class Divzero(Exception): pass
class Oflow(Exception): pass
class Uflow(Exception): pass
```

Unfortunately, when you re-release your code, you create a maintenance problem for your users. If they've listed your exceptions explicitly, they now

have to go back and change every place they call your library to include the newly added exception name:

```
# client.py
try:
    mathlib.func()
except (mathlib.Divzero, mathlib.Oflow, mathlib.Uflow):
    …handle and recover…
```

This may not be the end of the world. If your library is used only in-house, you can make the changes yourself. You might also ship a Python script that tries to fix such code automatically (it would probably be only a few dozen lines, and it would guess right at least some of the time). If many people have to change all their `try` statements each time you alter your exception set, though, this is not exactly the politest of upgrade policies.

Your users might try to avoid this pitfall by coding empty `except` clauses to catch *all* possible exceptions:

```
# client.py
try:
    mathlib.func()
except:                                 # Catch everything h
    …handle and recover…
```

But as noted in the prior chapter, this workaround might catch more than they bargained for—things like running out of memory, keyboard interrupts (Ctrl+C), system exits, and even typos in their own `try` block's code will all trigger exceptions, and such things should pass, not be caught and erroneously classified as library errors. Catching the `Exception` superclass improves on this but still intercepts—and thus may mask—program errors.

And really, in this scenario, users want to catch and recover from *only* the specific exceptions the library is defined and documented to raise. If any other exception occurs during a library call, it's likely a genuine bug in the library (and it's probably time to contact the vendor). As a rule of thumb, it's usually better to be specific than general in exception handlers—an idea we'll revisit as a "gotcha" in the next chapter.

So what to do, then? In principle again, the library module could provide a tuple object that contains all the exceptions it can possibly raise. The client could then import the tuple and name it in an `except` clause to catch all the library's exceptions (recall that using a tuple catches any of its exceptions). This would work and support mods, but you'd need to keep the tuple up-to-date with library exceptions, and that's both error-prone and tedious.

Class exception hierarchies solve this dilemma better. Rather than defining your library's exceptions as a set of autonomous classes, arrange them into a class tree with a common superclass to encompass the entire category:

```
# mathlib.py
class NumErr(Exception): pass
class Divzero(NumErr): pass
class Oflow(NumErr): pass

def func():
    …
    raise DivZero()
…and so on…
```

This way, users of your library simply need to list the common superclass (i.e., *category*) to catch all of your library's exceptions—both now and in the future:

```
# client.py
import mathlib

try:
    mathlib.func()
except mathlib.NumErr:
    …handle and recover…
```

When you go back and hack (update) your code again now, you can add new exceptions as new *subclasses* of the common superclass:

```
# mathlib.py
…
class Uflow(NumErr): pass
```

The end result is that user code that catches your library's exceptions will keep working, *unchanged*. In fact, you are free to add, delete, and change exceptions arbitrarily in the future—as long as clients name the superclass, and that superclass remains intact, they are insulated from changes in your exceptions set. In other words, class exceptions provide a better answer to maintenance issues than other solutions can.

Class-based exception hierarchies also support state retention and inheritance in ways that make them ideal in larger programs. To understand these roles, though, we first need to see how user-defined exception classes relate to the built-in exceptions from which they inherit.

Built-in Exception Classes

The prior section's example wasn't really pulled out of thin air. All built-in exceptions that Python itself may raise are predefined class objects. Moreover, they are organized into a shallow hierarchy with general superclass categories and specific subclass types, much like the prior section's final exceptions class tree.

All the familiar exceptions you've seen (e.g., `SyntaxError`) are really just predefined classes, available as built-in names in the module named `builtins`. In addition, Python organizes the built-in exceptions into a hierarchy to support a variety of catching modes. For example:

`BaseException` *: topmost root, with printing and constructor defaults*

: The top-level root superclass of exceptions. This class is not supposed to be directly inherited by user-defined classes (use `Exception` instead). It provides default printing and state retention behavior inherited by subclasses. If the `str` built-in is called on an instance of this class (e.g., by `print`), the class returns the display strings of the constructor arguments passed when the instance was created (or an empty string if there were no arguments). In addition, unless subclasses replace this class's constructor, all of the arguments passed to this class at instance construction time are stored in its `args` attribute as a tuple.

`Exception` *: root of user-defined exceptions*

: The top-level root superclass of application-related exceptions. This is an immediate subclass of `BaseException` and is a superclass to

every other built-in exception, except the system exit event classes (`SystemExit` , `KeyboardInterrupt` , and `GeneratorExit`) and an exception-group class we'll ignore here. Nearly all user-defined classes should inherit from this class, not `BaseException` . When this convention is followed, naming `Exception` in a `try` statement's handler ensures that your program will catch everything but system exit events, which should normally be allowed to pass. In effect, `Exception` becomes a catchall in `try` statements but is more accurate than an empty `except` .

`ArithmeticError` : root of numeric errors

A subclass of `Exception` , and the superclass of all numeric errors. Its subclasses identify specific numeric errors: `OverflowError` , `ZeroDivisionError` , and `FloatingPointError` .

`LookupError` : root of indexing errors

A subclass of `Exception` , and the superclass category for indexing errors for both sequences and mappings: `IndexError` and `KeyError` .

`OSError` : root of IO and other system-function errors, with details

A subclass of `Exception` , with attributes for error details (e.g., `errno` , `strerror` , and `filename`), and subclasses for specific errors: `FileNotFoundError` , `PermissionError` , `TimeoutError` , and more.

And so on—because the built-in exception set is prone to frequent changes, this book doesn't document it exhaustively. You can read further about its contents and structure in the Python library manual.

Built-in Exception Categories

The built-in class tree allows you to choose how specific or general your handlers will be. For example, because the built-in exception `ArithmeticError` is a superclass for more specific exceptions such as `OverflowError` and `ZeroDivisionError` :

- By listing `ArithmeticError` in a `try` , you will catch *any* kind of numeric error raised.
- By listing `ZeroDivisionError` , you will intercept *just* that specific type of error and no others.

Similarly, because `Exception` is the superclass of all application-level exceptions, you can generally use it as a *catchall*—as outlined in the prior chapter, the effect is much like an empty `except`, but it allows system exit exceptions to pass and propagate as they usually should:

```
try:
    …
except Exception:                                  # Exits
    …handle all application exceptions…
else:
    …handle no-exception case…
```

This technique is reliable because Python requires all classes to derive from built-in exceptions. Still, this scheme suffers most of the same potential pitfalls as the empty `except`, as described in the prior chapter—it might intercept exceptions intended for elsewhere, and it might mask genuine programming errors. Since this is such a common issue, we'll revisit it one more time as a "gotcha" in the next chapter.

Whether or not you will leverage the categories in the built-in class tree, it serves as a good example. By using similar techniques for class exceptions in your own code, you can provide exception sets that are flexible and easily modified.

Default Printing and State

Built-in exceptions also provide default print displays and state retention, which is often as much logic as user-defined classes require. Unless you redefine the constructors your classes inherit from built-ins, any constructor arguments you pass to these classes are automatically saved in the instance's `args` tuple attribute and are automatically displayed when the instance is printed. An empty tuple and display string are used if no constructor arguments are passed, and a single argument displays as itself (not as a tuple) and serves as message details.

This explains why arguments passed to *built-in* exception classes show up in error messages—any constructor arguments are attached to the instance and displayed when the instance is printed:

```
>>> raise IndexError                    # Same as Inde
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError

>>> raise IndexError('bad')             # Constructor
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: bad

>>> i = IndexError('bad', 'stuff')      # Available ir
>>> i.args
('bad', 'stuff')
>>> print(i)                            # Displays arg
('bad', 'stuff')
>>> i                                   # Uses repr fc
IndexError('bad', 'stuff')
```

The same holds true for *user-defined* exceptions because they inherit the constructor and display methods present in their built-in superclasses:

```
>>> class E(Exception): pass

>>> raise E
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
E

>>> raise E('bad')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
E: bad

>>> i = E('bad', 'stuff')
>>> i.args
('bad', 'stuff')
>>> print(i)
('bad', 'stuff')
>>> i
E('bad', 'stuff')
```

When intercepted in a `try` statement, the exception instance object gives access to both the original constructor arguments and the display method:

```
>>> try:
...     raise E('bad')                    # Displays +
... except E as X:
...     print(f'{X} - {X.args} - {X!r}')
...
bad - ('bad',) - E('bad')

>>> try:
...     raise E('bad', 'stuff')           # Multiple a
... except E as X:
...     print(f'{X} - {X.args} - {X!r}')
...
('bad', 'stuff') - ('bad', 'stuff') - E('bad', 'stuff')
```

Note that exception instance objects are not strings themselves, but use the `__str__` and `__repr__` operator-overloading methods we studied in Chapter 30 to provide display strings for `print` and other contexts. To concatenate with real strings, perform manual conversions: `str(X)`, `'%s' % X`, `f'{X}'`, and the like.

Although this automatic state and display support is useful by itself, for more specific display and state retention needs, you can always redefine inherited methods such as `__str__` and `__init__` in `Exception` subclasses—as the next section shows.

Custom Print Displays

As we saw in the preceding section, by default, instances of class-based exceptions display whatever you passed to the class constructor when they are caught and printed:

```
>>> class MyBad(Exception): pass

>>> try:
...     raise MyBad('Sorry--my mistake!')
... except MyBad as X:
...     print(X)
...
Sorry--my mistake!
```

This inherited default display model is also used if the exception is displayed as part of an error message when the exception is not caught:

```
>>> raise MyBad('Sorry--my mistake!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MyBad: Sorry--my mistake!
```

For many roles, this is sufficient. To provide a more custom display, though, you can define one of two string-representation overloading methods in your class (`__repr__` or `__str__`) to return the string you want to display for your exception. The string the method returns will be displayed if the exception either is caught and printed or reaches the default handler:

```
>>> class MyBad(Exception):
        def __str__(self):
            return 'Stuff happens...'

>>> try:
...     raise MyBad()
... except MyBad as X:
...     print(X)
...
Stuff happens...

>>> raise MyBad()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MyBad: Stuff happens...
```

Whatever your method returns is included in error messages for uncaught exceptions and used when exceptions are printed explicitly. The method returns a hardcoded string here to illustrate, but it can also perform arbitrary text processing, possibly using state information attached to the instance object. The next section looks at state information options.

First, though, one fine point: you generally must redefine `__str__` for exception display purposes because the built-in exception superclasses already have a `__str__` method, and `__str__` is preferred to `__repr__` in some contexts—including error-message displays. If you define a `__repr__` , printing will happily call the built-in superclass's `__str__` instead:

```
>>> class Oops(Exception):
        def __repr__(self): return 'Custom display not

>>> raise Oops("Nobody's perfect")
…
Oops: Nobody's perfect
```

But a custom `__str__` is used if defined:

```
>>> class Oops(Exception):
        def __str__(self): return 'Custom display used'

>>> raise Oops("Nobody's perfect")
…
Oops: Custom display used
```

See Chapter 30 for more details on these special operator-overloading methods.

Custom State and Behavior

Besides supporting flexible hierarchies, exception classes also provide storage for extra state information as instance *attributes*. As discussed earlier, built-in exception superclasses provide a default constructor that automatically saves constructor arguments in an instance tuple attribute named `args`. Although the default constructor is adequate for many cases, for more custom needs we can provide a constructor of our own. In addition, classes may define methods for use in handlers that provide precoded exception processing logic.

Providing Exception Details

When an exception is raised, it may cross arbitrary file boundaries—the `raise` statement that triggers an exception and the `try` statement that catches it may be in completely different module files. It is not generally feasible to store extra details in global variables because the `try` statement might not know which file the globals reside in. Passing extra state information along in the exception itself allows the `try` statement to access it more reliably.

With classes, this is nearly automatic. As we've seen, when an exception is raised, Python passes the class instance object along with the exception. Code in `try` statements can access the raised instance by listing an extra variable after the `as` keyword in an `except` handler. This provides a natural hook for supplying data and behavior to the handler. Generic instance-access tools like `sys.exc_info` used earlier enable the same interfaces.

For example, a program that parses data files might signal a formatting error by raising an exception instance that is filled out with extra details about the error (the input file here isn't real because this demo dies before reading it!):

```
>>> class FormatError(Exception):
        def __init__(self, line, file):             # Cu
            self.line = line
            self.file = file

>>> def parser(file):                               # Po
        raise FormatError(62, file=file)            # Wh

>>> try:
...     parser('code.py')
... except FormatError as X:
...     print(f'Error at: {X.file} #{X.line}')     # Cu
...
Error at: code.py #62
```

In the `except` clause here, the variable `X` is assigned a reference to the instance that was generated when the exception was raised. This gives access to the attributes attached to the instance by the custom constructor. Although we could rely on the default state retention of built-in superclasses, it's less relevant to our application (and doesn't support the keyword arguments used in the prior example):

```
>>> class FormatError(Exception): pass              # In

>>> def parser(file):
        raise FormatError(file, 62)                 # No

>>> try:
...     parser('code.py')
... except FormatError as X:
```

```
...     print(f'Error at: {X.args[0]} #{X.args[1]}')
...
Error at: code.py #62
```

Providing Exception Methods

Besides enabling application-specific state and display, classes also support extra behavior for exception objects. That is, the exception class can also define unique *methods* to be called in handlers. The file *parsely.py* in Example 35-3, for example, adds a method that uses exception custom state information to log errors to a file automatically.

Example 35-3. parsely.py

```
from time import asctime

class FormatError(Exception):
    logfile = 'parser-errors.txt'
    def __init__(self, line, file):
        self.line = line
        self.file = file
    def logerror(self):
        with open(self.logfile, 'a') as log:
            print(f'Error at: {self.file} #{self.line}

def parser(file):
    # Parse a file here...
    raise FormatError(line=62, file=file)

if __name__ == '__main__':
    try:
        parser('code.py')
    except FormatError as exc:
        exc.logerror()
```

When run, this script appends its error message to a file in response to method calls in the exception handler (use `type` instead of the Unix `cat` on Windows, and see Python manuals for `time.asctime`):

```
$ python3 parsely.py
$ python3 parsely.py
```

```
$ cat parser-errors.txt
Error at: code.py #62 [Sat Jul 13 12:22:19 2024]
Error at: code.py #62 [Sat Jul 13 12:22:25 2024]
```

In such a class, methods (like `logerror`) may also be inherited from superclasses, and instance attributes (like `line` and `file`) provide a place to save state information that provides extra context for use in later method calls. Moreover, exception classes are free to customize and extend inherited behavior:

```
class CustomFormatError(FormatError):
    def logerror(self):
        …something unique here…
…
raise CustomFormatError(…)
…
try:
    …
except FormatError as exc:
    exc.logError()              # Runs the raised clas
```

In other words, because they are defined with classes, all the benefits of OOP that we studied in Part VI are available for use with exceptions in Python.

Two final notes here: first, the raised instance object assigned to `exc` in this code is also available generically as the second item in the result tuple of the `sys.exc_info()` call used earlier—a tool that returns information about the exception being handled. This call can be used if you do not list an exception name in an `except` clause but still need access to the exception that occurred, or to any of its attached state information or methods. And second, although our class's `logerror` method appends a custom message to a logfile, it could also generate Python's standard error message with stack trace using tools in the `traceback` standard-library module, which uses traceback objects.

To learn more about `sys.exc_info` and tracebacks, though, we need to move ahead to the next chapter.

Exception Groups: Yet Another Star!

But wait—just when you thought it was safe to put exceptions in the win column, the exceptions story has recently sprouted yet another wild plot twist, which is sufficiently limited and arcane to pass as an optional follow-up topic for most Python learners (and was deferred until now for this reason). Lest it crop up in one of those silly job interviews that favor the inane over the practical, though, here's a quick peek.

Let's get right to the code. As we've seen, `try` statements normally run at most *one* matching handler clause, plus an optional `finally` on exit:

```
>>> try:
...     raise IndexError()
... except IndexError:
...     print('Got IE')
... except (SyntaxError, TypeError):
...     print('Got SE')
...
Got IE
```

Python 3.11, though, adds the ability to trigger *multiple* matching handlers in a single `try` statement by wrapping them in an *exception group* and catching them with `except*` clauses:

```
>>> try:
...     raise ExceptionGroup('Many', [IndexError(), Syr
... except* IndexError:
...     print('Got IE')
... except* (SyntaxError, TypeError):
...     print('Got SE')
...
Got IE
Got SE
```

In a nutshell, each `except*` clause can process and consume one exception, or one batch of them, in the group. Here, the first clause runs for `IndexError` and the second for `SyntaxError` (a tuple means "any" as before). The *group* is simply an exception object made by calling a built-in

exception class provided for this role, passing a message-string label used in displays, along with a sequence of exceptions to be matched by `except*` clauses in a `try` .

Syntactically, an empty `except*` is not allowed, and a basic `except` cannot be mixed with `except*` —but an `else` and `finally` can. Moreover, `except*` cannot host a `break` , `continue` , or `return` —but `except` can. Like the awkward `except` / `else` / `finally` mixing rules before it, these special cases probably qualify `except*` as a distinct statement form; adding it to the mix makes `try` an overloaded jumble of semi-related functionality.

Semantically, each `except*` clause executes at most once, and consumes all matching exceptions in the group. In addition, each exception in the group is handled by at most one `except*` clause—the topmost clause that matches it; optional `as` variables are assigned exception groups—with attributes like `exceptions` that expose their contents; and any *unmatched* exceptions in the group are reraised after the `try` statement processes matches—with a top-level message that denotes those unmatched:

```
>>> excs = ExceptionGroup('Many', [IndexError(), Syntax
>>> try:
...     raise excs
... except* IndexError as E:
...     print(f'Got IE: {E} => {E.exceptions}')
... except* SyntaxError as E:
...     print(f'Got SE: {E} => {E.exceptions}')
...
Got IE: Many (1 sub-exception) => (IndexError(),)
Got SE: Many (1 sub-exception) => (SyntaxError(),)
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  | ExceptionGroup: Many (1 sub-exception)
  +-+---------------- 1 ----------------
    | TypeError
    +------------------------------------
```

When the group has multiple exceptions of the *same* type, a matching `except*` consumes them all and can process them in normal iteration code. As also shown next, *spaces* around the `*` are allowed and ignored—despite

all the `except*` labels in docs, and more reflective of the wildcard nature of these clauses:

```
>>> try:
...     raise ExceptionGroup('Dups', [IndexError(), Typ
... except *IndexError:
...     print('Got IE')
... except *TypeError as E:
...     print(f'Got TE: {E} => {E.exceptions}')
...
Got IE
Got TE: Dups (2 sub-exceptions) => (TypeError(1), TypeE
```

As usual, a group's exception matches an `except*` that names its class or one of its *superclasses*. Because of this, the *ordering* of clauses is both subtle and important—the first match wins and removes exceptions from the group. In the following, for example, the first clause gobbles `IndexError` via its `LookupError` superclass, along with the two `TypeError` s in the group (but reversing the clauses' order would handle `IndexError` separately):

```
>>> try:
...     raise ExceptionGroup('Dups', [IndexError(), Typ
... except* (TypeError, LookupError) as E:
...     print('Got1:', E)
... except* IndexError as E:
...     print('Got2:', E)
...
Got1: Dups (3 sub-exceptions)
```

The `except*` can also match basic *individual* exceptions, which are automatically wrapped in a group to appease group-based code in the handler:

```
>>> try:
...     raise IndexError
... except* IndexError as E:
...    print(f'Got IE: {E} => {E.exceptions}')
...
Got IE:  (1 sub-exception) => (IndexError(),)
```

And a basic `except` can catch a group as a *collective* and process it manually, but an `except*` cannot catch a group because it would be ambiguous (a schism of the sort that's usually a hallmark of an ad hoc extension):

```
>>> try:
...     raise ExceptionGroup('Lots', [IndexError(), Syr
... except ExceptionGroup as E:
...     print(f'Got group: {E} => {E.exceptions}')
...     for exc in E.exceptions:
...         print('With exc:', type(exc))
...
Got group: Lots (2 sub-exceptions) => (IndexError(), Sy
With exc: <class 'IndexError'>
With exc: <class 'SyntaxError'>
```

Finally, for comparison, here's a *catchall* in both models—though there's no reason to use `except*` to catch a single exception (unless overly complicated code is your thing):

```
>>> try:
...     raise ExceptionGroup('Dups', [IndexError()])
... except* Exception as E:
...     print(type(E.exceptions[0]))
...
<class 'IndexError'>

>>> try:
...     raise IndexError()
... except Exception as E:
...     print(type(E))
...
<class 'IndexError'>
```

For another exception-groups example, see the next chapter's Example 36-2, which demos how runtime nesting consumes items in groups (short story: groups propagate until empty, then die like individual exceptions).

Design concerns aside, the "why" of `except*` is even more elusive than the "how." While it's conceivable that some programs may wish to collect a set of

exceptions and send them to a `try` statement as a *batch*, it's harder to understand why these wildly rare programs could not be expected to package with normal exception objects and process with normal iteration code—instead of convoluting the Python language for everyone.

Because exception groups are an obscure tool with very narrow roles, we'll defer to Python's manuals for more info on this esoteric `try` extension that, like many a Python mod, seems to blow up complexity radically to address a purported need that somehow managed to go unnoticed for all of Python's first three decades+. How did we live?

Chapter Summary

In this chapter, we explored both built-in exceptions and ways to code exceptions of our own. As we learned, exceptions are implemented as class instance objects. Exception classes support the concept of exception hierarchies that ease maintenance, allow data and behavior to be attached to exceptions as instance attributes and methods, and allow exceptions to inherit tools from superclasses as usual in OOP.

We saw that in a `try` statement, catching a superclass catches that class as well as all subclasses below it in the class tree—superclasses become exception category names, and subclasses become more specific exception types within those categories. We also saw that the built-in exception superclasses we must inherit from provide usable defaults for printing and state retention, which we can override if desired.

Finally, armed with our new knowledge of exception objects, we also peeked at exception groups and the `except*` clause, used to run multiple handlers in a `try` . We questioned whether this extension's convolution of `try` statements is justified by its perceived roles; it's a lot to ask of most Python users, but this question is ultimately yours to answer.

The next chapter wraps up this part of the book by exploring some common use cases for exceptions and surveying tools commonly used by Python programmers. Before we get there, though, here's this chapter's quiz.

Test Your Knowledge: Quiz

1. What are the two main constraints on user-defined exceptions in Python?
2. How are raised exceptions matched to `except` handler clauses?
3. Name two ways that you can attach context information to exception objects.
4. Name two ways that you can specify the error-message text for exception objects.
5. What do `except*` clauses do in a `try` statement?

Test Your Knowledge: Answers

1. Exceptions must be defined by *classes* (that is, a class instance object is raised and caught). In addition, exception classes must be derived from the *built-in* class `BaseException` or one of its subclasses; most programs inherit from its `Exception` subclass to support catchall handlers for normal kinds of exceptions.
2. Exceptions match by superclass relationships: naming a *superclass* in an exception handler will catch instances of that class, as well as instances of any of its *subclasses* lower in the class tree. Because of this, you can think of superclasses as general exception categories and subclasses as more specific types of exceptions within those categories.
3. You can attach context information to exceptions with either custom or default constructors. A *custom* constructor can fill out attributes in a raised instance object that are specific to the program. For simpler needs, built-in exception superclasses provide a *default* constructor that stores its arguments on the instance automatically as tuple attribute `args`. Handlers can list a variable to be assigned to the raised instance, then go through this name to access attached state information and call any methods defined in the class.
4. The error-message text in exceptions can be specified with a custom `__str__` operator-overloading method. For simpler needs, built-in exception superclasses automatically display anything you pass to the class constructor. Operations like `print` and `str` automatically fetch the display string of an exception object when it is printed either explicitly or as part of an error message.
5. In a `try`, `except*` is used to run possibly *multiple* handlers for exceptions raised as part of a *group*. The `except*` also comes with

heavy semantics, has special-case syntax and rules, does not combine with basic `except`, and is rarely useful in most Python programs. Nevertheless, there it is.