# Chapter 33. Exception Basics

This part of the book deals with *exceptions*—events that signal conditions and modify the flow of control through a program. In Python, exceptions are triggered automatically on errors, and they can be both triggered and intercepted by your code. They are processed by four statements we'll study here, the first of which comes in multiple flavors that qualify as different statement forms by some measures:

*`try`*/*`except`*/*`else`*/*`finally`*

Catch and recover from exceptions raised by Python, or by you

*`raise`*

Trigger an exception manually in your code

*`assert`*

Conditionally trigger an exception in your code

*`with`*

Use context managers that automate exception handling

We've met some of these briefly before, but full coverage of this topic was saved until the end of the main part of this book because you need to know about *classes* to code exceptions of your own. Still, with a few exceptions (pun intended), you'll find that exception handling is simple in Python because it's integrated into the language itself as another high-level tool. Before we dig into the "how," though, let's get clear on the "why."

## Why Use Exceptions?

In a nutshell, exceptions let us jump out of arbitrarily large chunks of a program. Consider the hypothetical pizza-making robot we discussed earlier in the book. Suppose we took the idea seriously and actually built such a machine. To make a pizza, our culinary automaton would need to execute a plan, which we would implement as a Python program: it would take an order, prepare the dough, add toppings, bake the pie, and so on.

Now, suppose that something goes very wrong during the “bake the pie” step. Perhaps the oven is broken, or perhaps our robot miscalculates its reach and spontaneously combusts. Clearly, we want to be able to jump to code that handles such unusual states quickly. As we have no hope of finishing the pizza task in such unusual cases, we might as well abandon the entire plan.

That’s exactly what exceptions let your programs do: they can jump to an exception handler in a single step, abandoning all activity begun since the exception handler was entered. Code in the exception handler can then respond to the raised exception as appropriate (by calling the fire department, for instance!).

One way to think of an exception is as a sort of structured “go-to.” An *exception handler* ( `try` statement) leaves a marker and executes some code. Somewhere further ahead in the program, an exception is raised that makes Python jump back to that marker, abandoning any code that was started and functions that were called after the marker was left. The net effect unwinds the program’s control flow back to the marker and resumes there.

This protocol provides a coherent way to respond to unusual events. Moreover, because Python jumps to the handler statement immediately, your code is simpler—there is usually no need to check status codes after every operation and function call that could possibly fail. Instead, we catch errors only where we need to recover from them.

## Exception Roles

In less hypothetical programs, exceptions serve a variety of purposes. Here are some of their most common roles:

*Error handling*

> Python raises exceptions whenever it detects errors in programs at runtime. You can catch and respond to the errors in your code, or ignore the exceptions that are raised. If an error is ignored, Python’s default exception-handling behavior kicks in: it stops the program and prints an error message. If you don’t want this default behavior, code a `try` statement to catch and recover from the exception—Python will jump to your `try` handler when the error is detected in the statement’s code, and your program will resume execution after the `try` .

*Event notification*

Exceptions can also be used to signal valid conditions without you having to pass result flags around a program or test them explicitly. For instance, a search routine might raise an exception on failure, rather than returning an integer result code—and hoping that the code will never be a valid result.

*Special-case handling*

Sometimes a condition may occur so rarely that it's hard to justify convoluting your code to handle it in multiple places. You can often eliminate special-case code by handling unusual cases in exception handlers in higher levels of your program. An `assert` can similarly be used to check that conditions are as expected during development.

*Termination actions*

As you'll see, the `finally` option in a `try` statement allows you to guarantee that required closing-time operations will be performed, regardless of the presence or absence of exceptions in your programs. The `with` statement offers an alternative in this department for objects that support its expected method-call protocol.

*Unusual control flows*

Finally, because exceptions are a sort of high-level and structured "go-to," you can use them as the basis for implementing exotic control flows. For instance, although the language does not explicitly support backtracking, you can implement it in Python by using exceptions and logic to unwind assignments.[1] There is no "go to" statement in Python (thankfully) and no built-in backtracking (today), but exceptions can sometimes serve similar roles; a `raise`, for instance, can be used to jump out of multiple loops in ways that `break` cannot.

We saw some of these roles briefly earlier and will study typical exception use cases in action later in this part of the book. For now, let's get started with a look at Python's exception-processing tools.

## Exceptions: The Short Story

Compared to some other core language topics we've explored in this book, exceptions are a fairly lightweight tool in Python. Because they are so simple, let's jump right into some code.

## Default Exception Handler

Suppose we write the following function in the interactive REPL of our choice:

```
>>> def fetcher(obj, index):
        return obj[index]
```

There's not much to this function—it simply indexes an object on a passed-in index. In normal operation, it returns the result of a legal index:

```
>>> food = 'pizza'
>>> fetcher(food, 4)                             # Like x[4]
'a'
```

However, if we ask this function to index off the end of the string, an exception will be triggered when the function tries to run `obj[index]`. Python detects out-of-bounds indexing for sequences and reports it by *raising* (triggering) the built-in `IndexError` exception:

```
>>> fetcher(food, 5)                             # Default h
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
```

Because our code does not explicitly catch this exception, it filters back up to the top level of the program and invokes the *default exception handler*, which simply prints the standard error message shown here.

By this point in the book, you've probably seen your share of standard error messages. They include the exception that was raised, along with a *stack trace* —a list of all the lines and functions that were active when the exception occurred, which has been largely omitted in this book for space and brevity.

The error message text here was printed by Python 3.12 in a console. It can vary slightly per release, and even per interactive REPL, so you shouldn't rely upon its exact form—in either this book or your code. When you're coding interactively in a console interface, the filename may be just “<stdin>,” meaning the standard input stream.

When working in the IDLE GUI’s interactive shell today, though, the filename is “<pyshell…>,” and source lines are displayed, too. Either way, file line numbers are not very meaningful when there is no file (you’ll see more interesting error messages later in this part of the book):

```
>>> fetcher(food, 5)                              # Default h
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    fetcher(food, 5)
  File "<pyshell#0>", line 2, in fetcher
    return obj[index]
IndexError: string index out of range
```

In a more realistic program launched outside the interactive REPL, after printing an error message the default handler at the top also *terminates* the program immediately. That course of action makes sense for simple scripts; errors often should be fatal, and the best you can do when they occur is inspect the standard error message.

## Catching Exceptions

Sometimes, though, program termination on exceptions isn’t what you want. Server programs, for instance, typically need to remain active even after internal errors. If you don’t want the default exception behavior, wrap the call in a `try` statement to catch exceptions yourself (copy/pasters: omit the “...” here per the note ahead):

```
>>> try:
...     fetcher(food, 5)
... except IndexError:                            # Catch and
...     print('got exception')
...
got exception
>>>
```

Now, Python automatically jumps to your *handler*—the block under the `except` clause that names the exception raised—when an exception is triggered while the `try` block is running. The net effect is to wrap a nested block of code in an error handler that intercepts the block's exceptions.

When working interactively like this, after the `except` clause runs, we wind up back at the Python prompt. In a more realistic program, `try` statements not only catch exceptions but also *recover* from them:

```
>>> def catcher():
        try:
            fetcher(food, 5)
        except IndexError:
            print('got exception')          # Catch anc
        print('continuing')

>>> catcher()
got exception
continuing
>>>
```

This time, after the exception is caught and handled, the program resumes execution after the entire `try` statement that caught it—which is why we get the "continuing" message here. We don't see the standard error message, and the program continues on its way normally.

Notice, though, that there's no way in Python to *go back* to the code that triggered the exception (short of rerunning the code that reached that point all over again, of course). Once you've caught the exception, control continues after the entire `try` that caught the exception, not after the statement that kicked off the exception. In fact, Python clears the memory of any functions that were exited as a result of the exception, like `fetcher` in our example; their variables are discarded, and they're not resumable. The `try` both catches exceptions and is where the program resumes.

Python does not, however, *undo* any work done by the `try` block up to the point where the exception occurred—any changes made to referenced mutable objects and accessible global names live on. This isn't a problem if it's known:

```
>>> L, S = [], 'text'
>>> def modder():
        L.append('added')                 # Change a
        global S; S = 'changed'           # Change a
        fetcher(food, 5)                  # Trigger c

>>> try:
...     modder()
... except IndexError:
...     print('got exception')
...
got exception
>>> L, S                                  # Changes r
(['added'], 'changed')
```

As you'll see later in this part of the book, the `try` can also use `except*` clauses to process *multiple* exceptions, but this is a convoluted extension with narrow scope that doesn't play well with others, and you can safely defer studying until you've mastered the fundamentals.

---

NOTE

*Presentation note*: The interactive REPL's "..." continuation prompt reappears in this part for some top-level `try` statements, because their code won't work if copied and pasted unless nested in a function or class (the `except` and other lines must align with the `try`, and not have extra preceding spaces that are needed to illustrate their indentation structure here). To run, simply type or paste statements with "..." prompts one line at a time, and without their leading "..." prompts.

---

## Raising Exceptions

So far, we've been letting Python raise exceptions for us by making mistakes (on purpose this time!), but our scripts can raise exceptions too—that is, exceptions can be raised by Python or by your program, and can be caught or not. To trigger an exception manually, simply run a `raise` statement. User-triggered exceptions are caught the same way as those Python raises. The following may not be the most useful Python code ever penned, but it makes the point—raising the built-in `IndexError` exception:

```
>>> try:
...     raise IndexError                          # Trigger e
... except IndexError:
...     print('got exception')
...
got exception
```

As usual, if they're not caught, user-triggered exceptions are propagated up to the top-level default exception handler and terminate the program with a standard error message:

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

As you'll see in the next chapter, the `assert` statement can be used to trigger exceptions, too—it's a conditional `raise` predicated on a test, used mostly for debugging purposes and sanity checks during development:

```
>>> assert 1 < 0, 'Not in this universe!'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Not in this universe!
```

Also in the next chapter, you'll learn that `raise` can use a `from` clause to "chain" exceptions; generally speaking, this is not common, but can be used to give more context where it's useful.

## User-Defined Exceptions

The `raise` statement demos in the prior section raised `IndexError`, a *built-in* exception defined in Python's built-in scope. As you'll learn later in this part of the book, you can also define new exceptions of your own that are specific to your programs. User-defined exceptions are coded with *classes*, which inherit from a built-in exception class—usually, the class named `Exception`:

```
>>> class Combust(Exception): pass          # User-defi

>>> def makePizza():
        raise Combust()                     # Raise an

>>> try:
...     makePizza()
... except Combust:                         # Catch cla
...     print('got exception')
...
got exception
>>>
```

As you'll see in upcoming chapters, exception classes allow scripts to build exception categories, which can inherit behavior and have attached state information and methods, and an `as` clause on an `except` can gain access to the exception object itself. Exception classes can also customize their message text displayed if they're not caught:

```
>>> class Combust(Exception):
        def __str__(self):
            return 'Call the fire department!...'

>>> raise Combust()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Combust: Call the fire department!...
>>>
```

## Termination Actions

Finally, `try` statements can say "finally"—that is, they may include `finally` blocks. These look like `except` handlers for exceptions, but the `try` / `finally` combination specifies termination actions that always execute "on the way out," regardless of whether an exception occurs in the `try` block or not. Continuing our REPL session:

```
>>> try:
...     fetcher(food, 4)
... finally:                                # Terminati
...     print('after fetch')
...
'a'
after fetch
>>>
```

Here, if the `try` block finishes *without* an exception, the `finally` block will run, and the program will resume after the entire `try`. In this case, this statement seems a bit silly—we might as well have simply typed the `print` right after a call to the function, and skipped the `try` altogether:

```
fetcher(food, 4)
print('after fetch')
```

There is a problem with coding this way, though: if the function call raises an exception, the `print` will never be reached. The `try` / `finally` combination avoids this pitfall—when an exception *does* occur in a `try` block, `finally` blocks are executed while the program is being unwound:

```
>>> def after():
        try:
            fetcher(food, 5)
        finally:
            print('after fetch')                 # Always ru
        print('after try?')                      # Run only

>>> after()
after fetch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in after
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
>>>
```

Here, we don't get the "after try?" message because control does not resume after the `try` statement when an exception occurs. Instead, Python jumps back to run the `finally` action and then *propagates* the exception up to a prior handler (in this case, to the default handler at the top). If we change the call inside this function so as not to trigger an exception, the `finally` code still runs, but the program continues after the `try`:

```
>>> def after():
        try:
            fetcher(food, 4)
        finally:
            print('after fetch')              # Both run
        print('after try?')

>>> after()
after fetch
after try?
>>>
```

In practice, `except` clauses in a `try` are useful for catching and recovering from exceptions, and `finally` clauses come in handy to guarantee that termination actions will fire regardless of any exceptions that may occur in the `try` block's code. For instance, you might use `try` / `except` combinations to catch errors raised by code that you import from a third-party library, and `try` / `finally` combos to ensure that calls to close files or terminate server connections are always run. We'll code some such practical examples later in this part of the book.

Although they serve conceptually distinct purposes, you can also mix `except` and `finally` clauses in the *same* `try` statement—the `finally` is run on the way out regardless of whether an exception was raised, and regardless of whether the exception was caught by an `except` clause. Such combos have rules you'll meet in the next chapter.

As you'll also learn in the next chapter, Python provides an alternative to the `try` / `finally` mix when using some types of built-in and user-defined objects. The `with` statement runs a *context manager* object's methods to guarantee that termination actions occur, irrespective of any exceptions in its nested block:

```
>>> with open('pizzarobot.txt', 'w') as file:        #
        file.write('Catch fire!\n')
```

Although this option requires fewer lines of code, it's applicable only when processing certain object types, so `try` / `finally` is a more general termination structure, and is often simpler than coding a class in cases where `with` is not already supported. On the other hand, `with` may also run startup actions too, and supports user-defined context management code with access to Python's full OOP toolset. To see how, let's move on to the next chapter.

## Chapter Summary

And that is the majority of the exception story; exceptions really are a simple tool.

To summarize, Python exceptions are a control-flow device. They may be raised by Python, or by your own programs. In both cases, they may be ignored (to trigger the default error handler), or caught by `try` statements (to be processed by your code). The `try` statement comes in logically distinct forms that can be combined—one that handles exceptions, and one that runs finalization code regardless of whether exceptions occur or not. Python's `raise` and `assert` statements trigger exceptions on demand—both built-ins and new exceptions we define with classes—and the `with` statement is an alternative way to ensure that termination actions are carried out for objects that support it.

In the rest of this part of the book, we'll fill in some of the details about the statements involved, examine the other sorts of clauses that can appear under a `try` (spoiler: it also allows an `else` for the no-exception case), and discuss class-based exception objects. The next chapter begins our tour by taking a closer look at the statements we introduced here. Before you turn the page, though, here are a few quiz questions to review.

## Test Your Knowledge: Quiz

1. Name three things that exception processing is good for.
2. What happens to an exception if you don't do anything special to handle it?
3. How can your script recover from an exception?
4. Name two ways to trigger exceptions in your script.
5. Name two ways to specify actions to be run at termination time, whether an exception occurs or not.

## Test Your Knowledge: Answers

1. Exception processing is useful for error handling, termination actions, and event notification. It can also simplify the handling of special cases and can be used to implement alternative control flows as a kind of structured "go-to" operation. In general, exception processing also cuts down on the amount of error-checking code your program may require—because all errors filter up to handlers, you may not need to test the outcome of every operation (see this chapter's sidebar "Why You Will Care: Error Checks" for an illustration).
2. Any uncaught exception eventually filters up to the default exception handler Python provides at the top of your program. This handler prints the familiar error message and shuts down your program.
3. If you don't want the default message and shutdown, you can code `try` statements with `except` clauses to catch and recover from exceptions that are raised within its nested code block. Once an exception is caught, the exception is terminated and your program continues after the `try`.
4. The `raise` and `assert` statements can be used to trigger an exception, exactly as if it had been raised by Python itself. In principle, you can also raise an exception by making a programming mistake, but that's not usually an explicit goal!
5. The `try` statement with a `finally` clause can be used to ensure actions are run after a block of code exits, regardless of whether the block raises an exception or not. The `with` statement can also be used to ensure termination actions are run, but only when processing object types that support it.

#### WHY YOU WILL CARE: ERROR CHECKS

One way to see how exceptions are useful is to compare coding styles in Python and languages without exceptions. For instance, if you want to write robust programs in the C language, you generally have to test return values or status codes after every operation that could possibly go astray, and propagate the results of the tests as your programs run:

```
doStuff()
{                                       # C program
    if (doFirstThing() == ERROR)        # Detect errors eve
        return ERROR;                   # even if not handl
    if (doNextThing() == ERROR)
        return ERROR;
    ...
    return doLastThing();
}

main()
{
    if (doStuff() == ERROR)
        badEnding();
    else
        goodEnding();
}
```

In fact, realistic C programs often have as much code devoted to error detection as to doing actual work. But in Python, you don't have to be so methodical (and neurotic!). You can instead wrap arbitrarily vast pieces of a program in exception handlers and simply write the parts that do the actual work, assuming all is normally well:

```
def doStuff():          # Python code
    doFirstThing()      # We don't care about exceptior
    doNextThing()       # so we don't need to detect tł
    ...
    doLastThing()

if __name__ == '__main__':
    try:
        doStuff()       # This is where we care about r
    except:             # so it's the only place we mus
        badEnding()
    else:                   # See the next chapter for else
        goodEnding()
```

Because control jumps immediately to a handler when an exception occurs, there's no need to instrument all your code to guard for errors, and there's no extra performance overhead to run all the tests. Moreover, because Python detects errors automatically, your code often doesn't need to check for errors in the first place. The upshot is that exceptions let you largely ignore the unusual cases and avoid error-checking code that can distract from your program's purpose.

---

[1] For any computer scientists in the audience, true backtracking is not part of the Python language. Backtracking undoes computations before it jumps back, but Python exceptions do not: local variables in open function calls run by the `try` are simply discarded, but changes made to globals and objects are retained (see the demo ahead). Even the generator functions and expressions we met in Chapter 20 don't do full backtracking—they simply respond to `next(G)` requests by restoring saved state and resuming. For more on backtracking, try books on AI or the Prolog or Icon programming languages.