Python *exceptions*

🐍

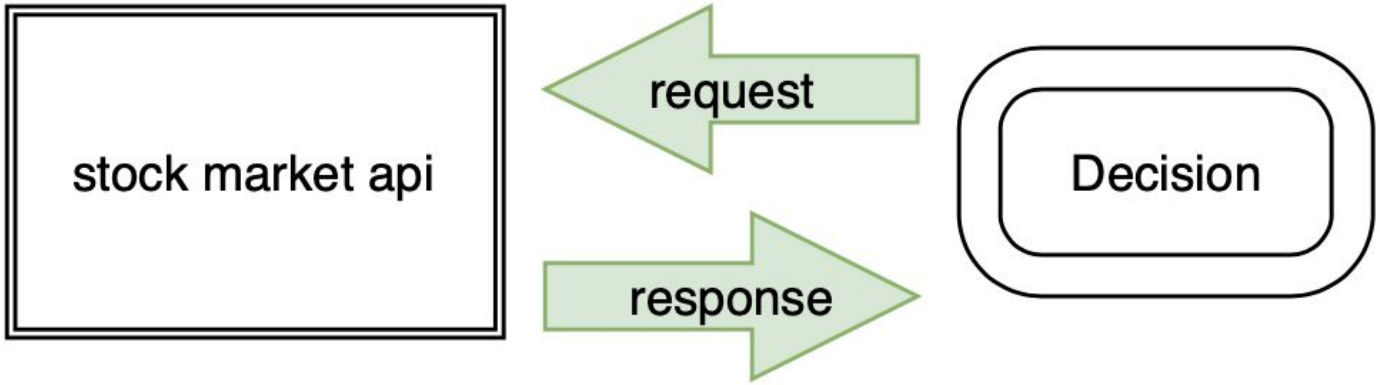
Overview

Exceptions are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions.

An exception is ***raised*** at the point where the error is detected;

it may be ***handled*** by the surrounding code block or by any code block that directly or indirectly invoked the code block where the error occurred.

Examples: Stock market



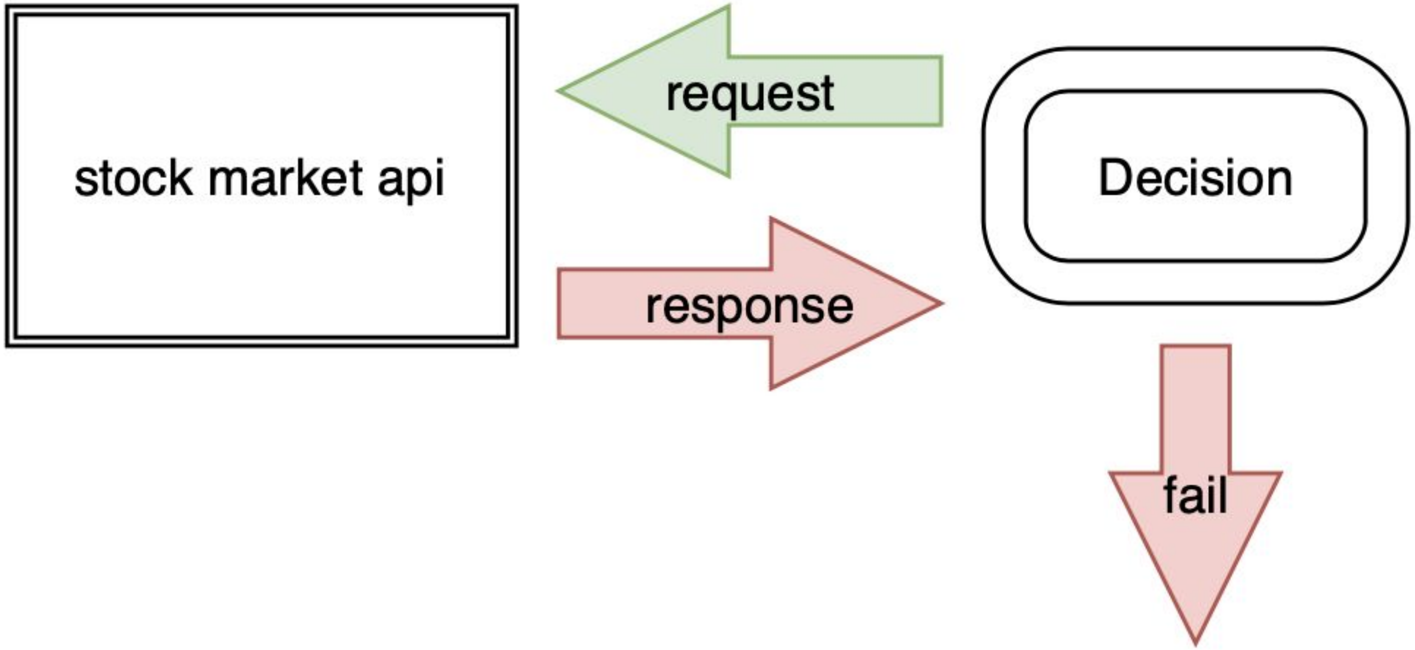
stock market api

request

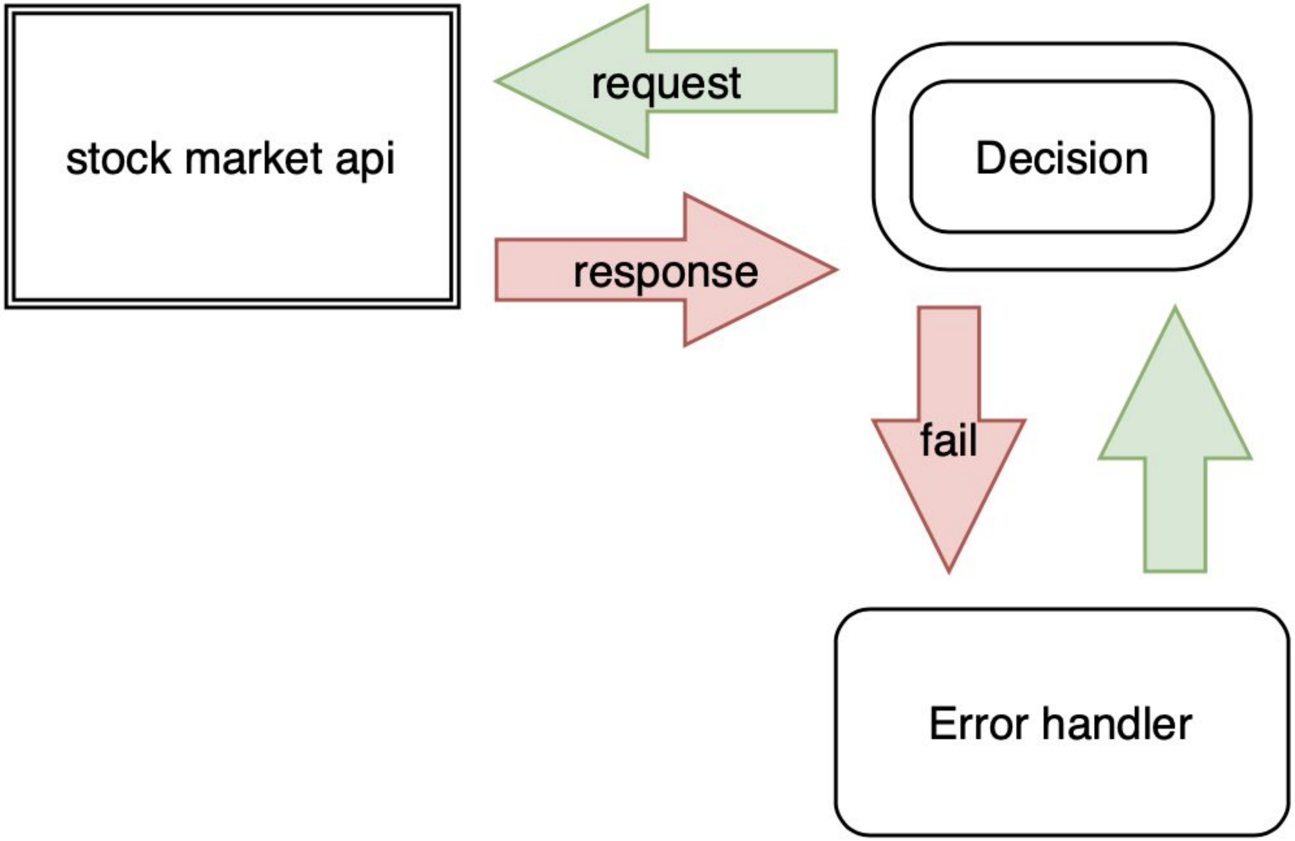
response

Decision

Examples: Stock market



Examples: Stock market



Examples: Database transaction

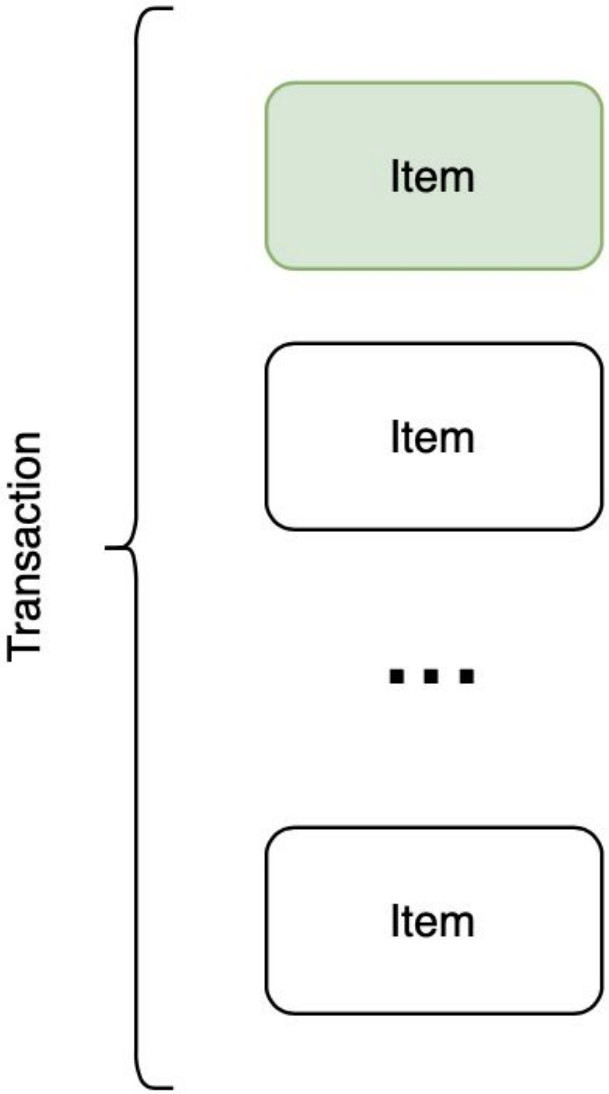
Item

Item

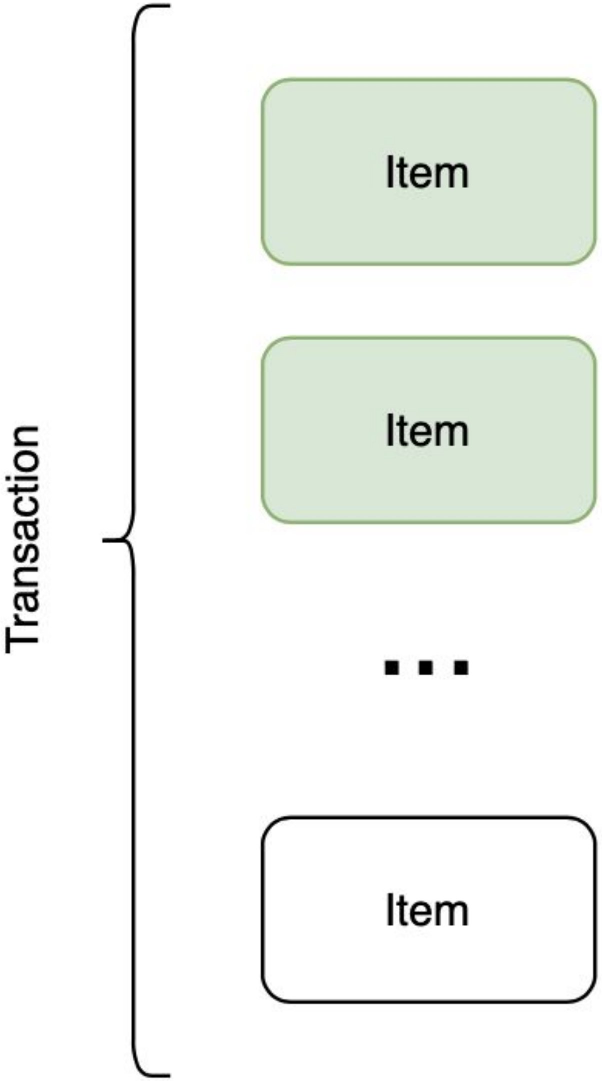
...

Item

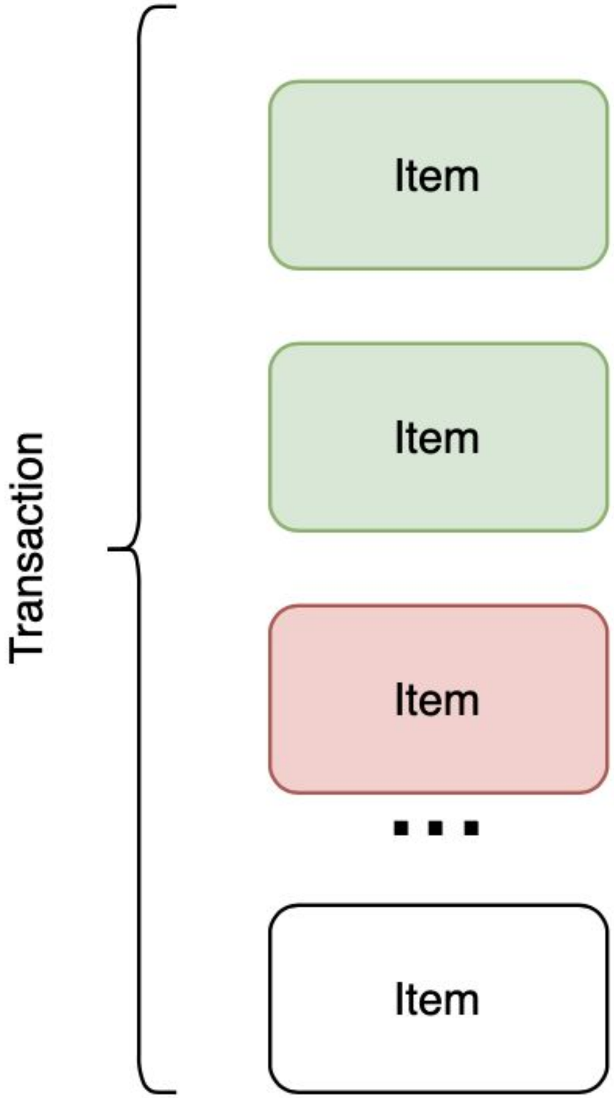
Examples: Database transaction



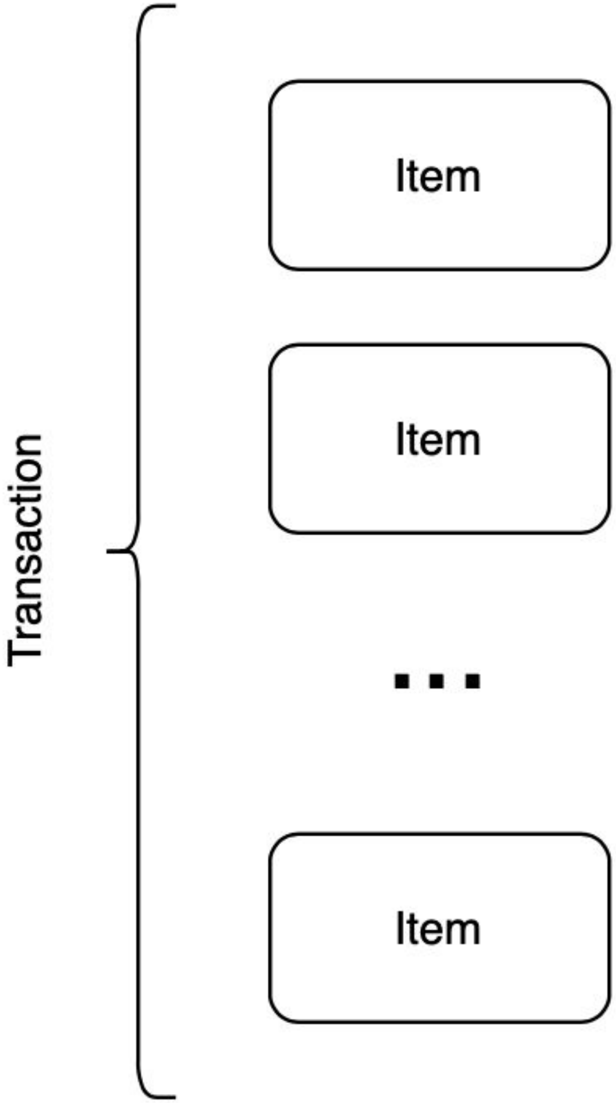
Examples: Database transaction



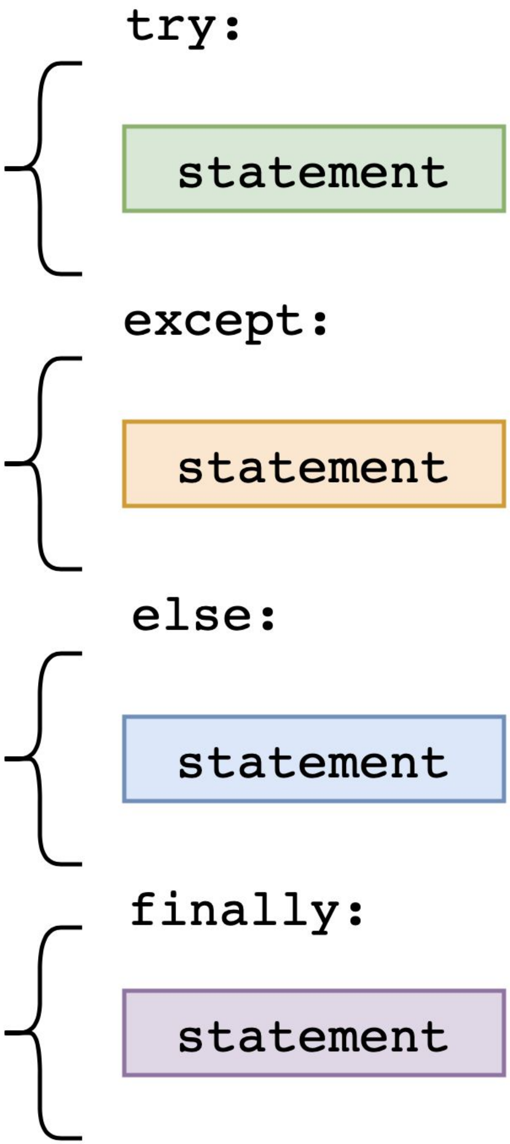
Examples: Database transaction



Examples: Database transaction



Structure



try:

statement

except:

statement

else:

statement

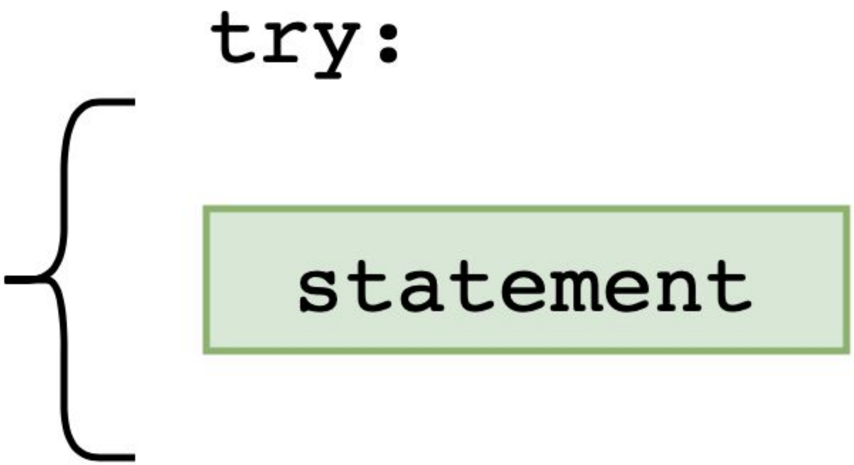
finally:

statement

Structure

First, the *try clause* (the statement(s) between the try and except keywords) is executed

If no exception occurs, the *except clause* is skipped and execution of the try statement is finished



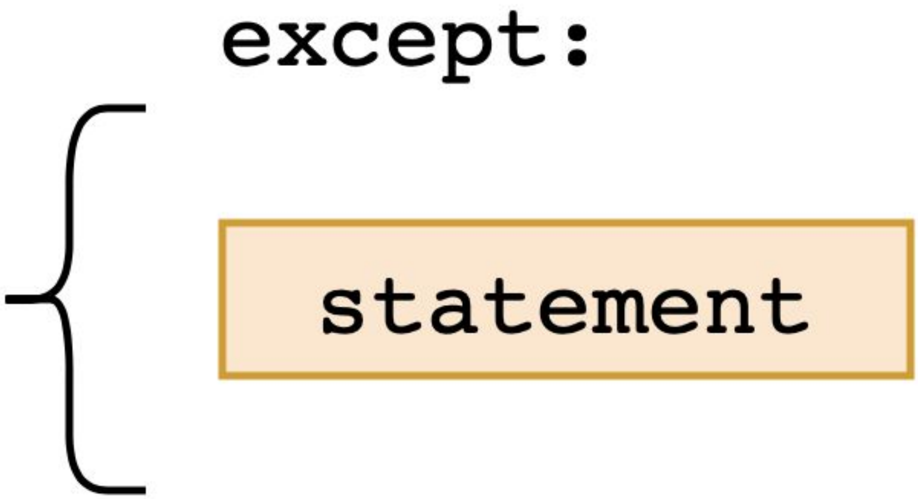
try:

statement

Structure

If an exception occurs during execution of the try clause, the rest of the clause is skipped

Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement



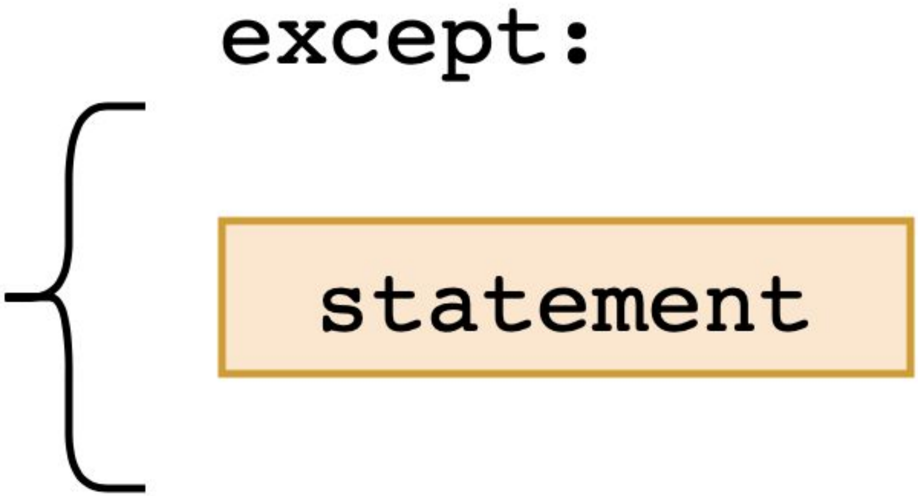
except:

statement

Structure

If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements

if no handler is found, it is an ***unhandled*** *exception* and execution stops with a message as shown above



Handling exceptions

```
>>> def divide(a: int, b: int) -> float:
...     try:
...         return a / b
...     except ZeroDivisionError as e:
...         print('ZeroDivisionError occurred')

>>> divide(1, 2)
0.5

>>> divide(1, 0)
ZeroDivisionError occurred during division process
```

Handling exceptions

A try statement may have more than one except clause, to specify handlers for different exceptions.

At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement.

An except clause may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Handling exceptions

```
>>> def divide(a, b) -> float:
...     try:
...         return int(a) / int(b)
...     except (ZeroDivisionError, ValueError) as e:
...         print(f'Exception occurred: {e!r}')

>>> divide('1', 'str')
Exception occurred: ValueError("invalid literal for int()
with base 10: 'str'")
```

Handling exceptions

```
>>> try:
...     int("bad idea")
... except ValueError as e:
...     print(isinstance(e, Exception))
...     print(e.__context__)
...     print(e.__cause__)
...     print(e.__traceback__)
True
invalid syntax (<stdin>, line 1)
None
<traceback object at 0x7fc89ae42248>
```

Handling exceptions

Variable **e** lives only inside ***except***

```
>>> try:
...     int("bad idea")
... except ValueError as e:
...     print(isinstance(e, Exception))
...     print(e.__context__)
...     print(e.__cause__)
...     print(e.__traceback__)
True
invalid syntax (<stdin>, line 1)
None
<traceback object at 0x7fc89ae42248>
```

Handling exceptions

A class in an except clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around – an except clause listing a derived class is not compatible with a base class)

Handling exceptions

```
>>> class B(Exception):
...     ...
...
... class C(B):
...     ...
...
... class D(C):
...     ...
...
... for cls in [B, C, D]:
...     try:
...         raise cls()
...     except D:
...         print("D")
...     except C:
...         print("C")
...     except B:
...         print("B")
...
B
C
D
```

Raising exceptions

The raise statement allows the programmer to force a specified exception to occur:

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

Hello there
```

Raising exceptions

The raise statement allows the programmer to force a specified exception to occur:

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

Hello there
```

Raising exceptions

The sole argument to raise indicates the exception to be raised.
This must be either an exception instance or an exception class (a class that derives from Exception).

If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
>>> raise ValueError  # shorthand for 'raise ValueError()'
```

Raising exceptions

```
>>> try:
...     int("Bad")
... except ValueError as e:
...     tb = e.__traceback__
...     raise Exception().with_traceback(tb)
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
  File "<stdin>", line 2, in <module>
Exception
```

```
>>> try:
...     int("Bad")
... except ValueError as e:
...     raise Exception
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
Exception
```

Raising exceptions

Python uses the “termination” model of error handling: an exception handler can find out what happened and continue execution at an outer level, but it cannot repair the cause of the error and retry the failing operation (except by re-entering the offending piece of code from the top).

Raising exceptions

If you need to determine whether an exception was raised but don’t intend to handle it, a simpler form of the raise statement allows you to re-raise the exception:

```
>>> try:
...     int('bad idea')
... except Exception:
...     print('Handle with additional logic')
...     raise

Handle with additional logic
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'bad
idea'

invalid literal for int() with base 10: 'bad idea'
```

Raising exceptions

The raise statement allows an optional from which enables chaining exceptions. For example:

```
>>> try:
...     v = {}['key']
... except KeyError as e:
...     raise ValueError('failed') from e

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyError: 'key'

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError: failed
```

Exception hierarchy

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      ...
      +-- OSError
           ...
      |    +-- TimeoutError
      +-- SyntaxError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      +-- Warning
           +-- DeprecationWarning
           ...
```

Exception hierarchy

In Python, all exceptions must be instances of a class that derives from BaseException

The built-in exception classes can be subclassed to define new exceptions; programmers are encouraged to derive new exceptions from the Exception class or one of its subclasses, and not from BaseException

```
>>> while True:
...     try:
...         print('Wow')
...     except BaseException:
...         pass
```

Exception hierarchy

```
>>> BaseException.__subclasses__()
[<class 'Exception'>,
<class 'GeneratorExit'>,
<class 'SystemExit'>,
<class 'KeyboardInterrupt'>]
```

Only system-dependant exceptions inherited from ***BaseException*** *directly*

Exception arguments

When an exception occurs, it may have an associated value, also known as the exception’s *argument*. The presence and type of the argument depend on the exception type

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as e:
...     print(type(e))    # the exception instance
...     print(e.args)     # arguments stored in
.args
...     print(e)
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
```

Warnings

Warning messages are typically issued in situations where it is useful to *alert* the user of some condition in a program, where that condition (normally) ***doesn’t warrant raising an exception and terminating the program***

For example, one might want to issue a warning when a program uses an obsolete module

Warnings

```
>>> import warnings

>>> def divide(a, b) -> float:
...     if any((isinstance(a, str), isinstance(b, str))):
...         warnings.warn("string type depricated",
DeprecationWarning)
...     try:
...         return int(a) / int(b)
...     except (ZeroDivisionError, ValueError) as e:
...         print(f'Exception occurred: {e!r}')

>>> a = divide('1', '2')
/Users/nightingale/miniconda3/bin/ptpython:3:
DeprecationWarning: string type depricated
```

Warnings

```
>>> import warnings

>>> def divide(a, b) -> float:
...     if any((isinstance(a, str), isinstance(b, str))):
...         warnings.warn(
                "string type depricated",
DeprecationWarning)
...     try:
...         return int(a) / int(b)
...     except (ZeroDivisionError, ValueError) as e:
...         print(f'Exception occurred: {e!r}')

>>> a = divide('1', '2')
/Users/nightingale/miniconda3/bin/ptpython:3:
DeprecationWarning: string type depricated
```

Assertion Error

AssertionError raised when an assert statement fails

Assertion Error

```
>>> assert True == False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

```
>>> assert True == False, "True is not equal False"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: True is not equal False

True is not equal False
```

Assertion Error

Assert statements are a convenient way to insert debugging assertions into a program

Assertion Error

```
>>> assert True == False
```

```
if __debug__:
    if not True == False: raise AssertionError
```

```
>>> assert True == False, "True is not equal False"
```

```
if __debug__:
    if not True == False:
        raise AssertionError("True is not equal False")
```

Assertion Error

These equivalences assume that `__debug__` and `AssertionError` refer to the built-in variables with those names.

In the current implementation, the built-in variable `__debug__` is `True` under normal circumstances, `False` when optimization is requested (command line option `-O`).

Assertion Error

Assignments to __debug__ are illegal. The value for the built-in variable is determined when the interpreter starts

```
>>> __debug__ = False
Syntax Error: assignment to keyword (<input>, line 1)
```

User-defined exceptions

Programs may name their own exceptions by creating a new exception class

```
>>> class CustomError(Exception):
...     """ My own exception """

>>> def func():
...     raise CustomError

>>> try:
...     func()
... except CustomError as e:
...     print(f'{e!r}')
CustomError()
```

User-defined exceptions

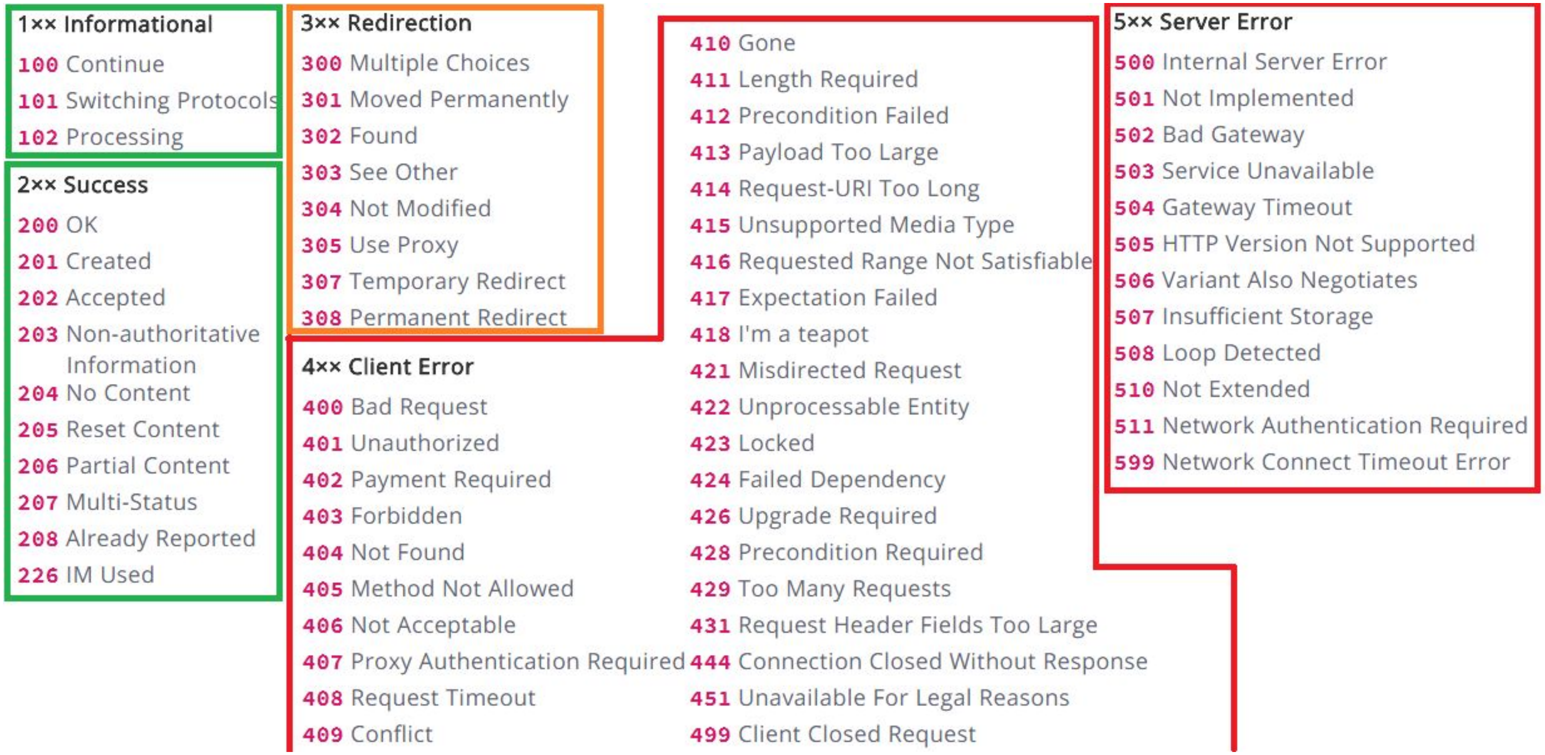
```
>>> class CustomArgumentError(CustomError):
...     """ Custom exception with arguments """
...     def __init__(self, text, payload):
...         self.text = text
...         self.payload = payload
...
...     def __str__(self):
...         return str((self.text, self.payload))
...
...     def __repr__(self):
...         return f"CustomArgumentError({self.text, self.payload})"

>>> def func():
...     raise CustomArgumentError("text", {'payload': True})

>>> try:
...     func()
... except CustomError as e:
...     print(repr(e))
CustomArgumentError(('text', {'payload': True}))
```

User-defined exceptions

When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions

User-defined exceptions

Imagine we are creating module which works with http api



1×× Informational

100	Continue
101	Switching Protocols
102	Processing

2×× Success

200	OK
201	Created
202	Accepted
203	Non-authoritative Information
204	No Content
205	Reset Content
206	Partial Content
207	Multi-Status
208	Already Reported
226	IM Used

3×× Redirection

300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
307	Temporary Redirect
308	Permanent Redirect

4×× Client Error

400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Payload Too Large
414	Request-URI Too Long
415	Unsupported Media Type
416	Requested Range Not Satisfiable
417	Expectation Failed
418	I'm a teapot
421	Misdirected Request
422	Unprocessable Entity
423	Locked
424	Failed Dependency
426	Upgrade Required
428	Precondition Required
429	Too Many Requests
431	Request Header Fields Too Large
444	Connection Closed Without Response
451	Unavailable For Legal Reasons
499	Client Closed Request

5×× Server Error

500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported
506	Variant Also Negotiates
507	Insufficient Storage
508	Loop Detected
510	Not Extended
511	Network Authentication Required
599	Network Connect Timeout Error

User-defined exceptions

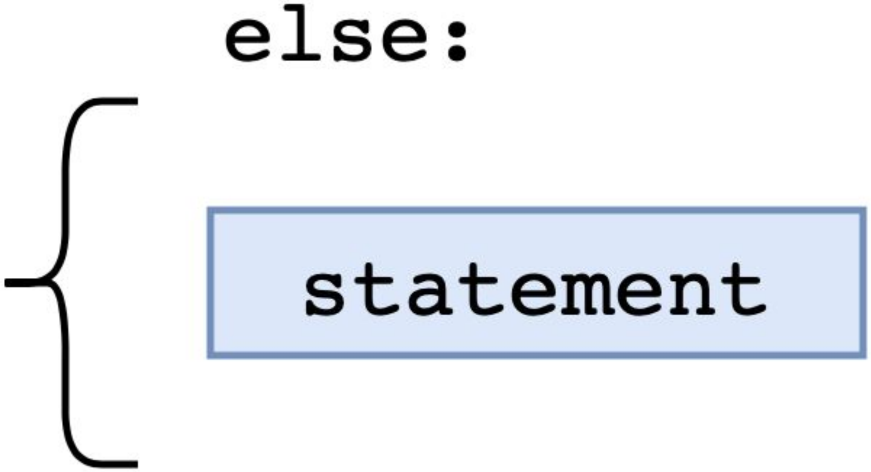
```
>>> class BaseApiError(Exception):
...     """ Base class for api errors """

>>> class ClientError(BaseApiError):
...     """ Base exception for 4xx statuses """

>>> class NotFoundError(ClientError):
...     """ Exception for 404 status """
```

User-defined exceptions

```
>>> class ValidationError(ClientError):
...     """ Exception for 400 status"""
...     def __init__(self, validation_errors=()):
...         self.validation_errors = validation_errors
...
...     def __str__(self):
...         return f"Validation errors
{self.validation_errors}"
...
...     def __repr__(self):
...         return f"ValidationError({self.validation_errors})"
```

else

The try … except statement has an optional *else clause*, which, when present, must follow all except clauses.



else:

statement

else

It is useful for code that must be executed if the try clause does not raise an exception. For example:

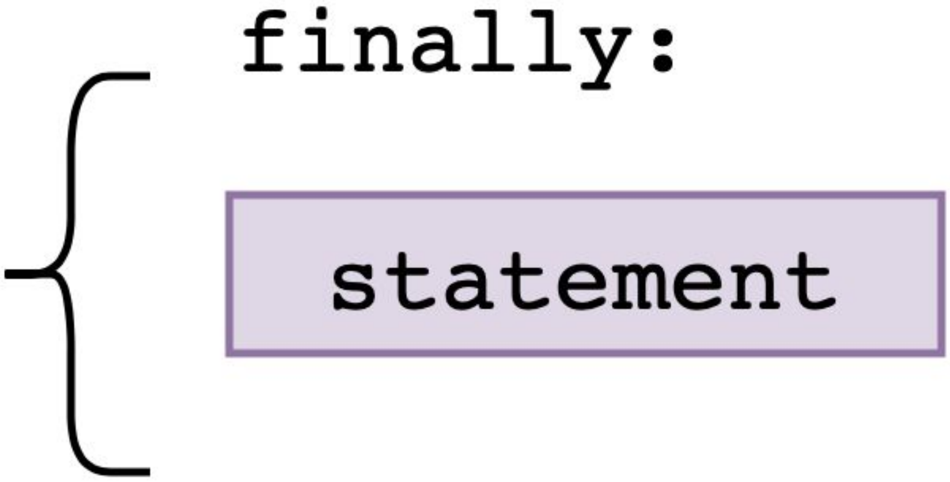
```python
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

else

The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn’t raised by the code being protected by the try … except statement.

finally

If a finally clause is present, the finally clause will execute as the last task before the try statement completes


finally:

statement

finally

finally optional clause is intended to define clean-up actions that must be executed under all circumstances. For example:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

finally

The following points discuss more complex cases when an exception occurs:

- If an exception occurs during execution of the try clause, the exception may be handled by an except clause. If the exception is not handled by an except clause, the exception is re-raised after the finally clause has been executed.
- An exception could occur during execution of an except or else clause. Again, the exception is re-raised after the finally clause has been executed.

finally

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")

>>> divide("1", "2")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'

unsupported operand type(s) for /: 'str' and 'str'
```

finally

The following points discuss more complex cases when an exception occurs:

- If the try statement reaches a break, continue or return statement, the finally clause will execute just prior to the break, continue or return statement's execution.

```
>>> def func():
...     try:
...         c = 0
...         while True:
...             c += 1
...             if c == 5:
...                 break
...     finally:
...         print(c)

>>> func()
5
```

finally

The following points discuss more complex cases when an exception occurs:

- If a finally clause includes a return statement, the returned value will be the one from the finally clause's return statement, not the value from the try clause's return statement.

```
>>> def func():
...     try:
...         return "Mystring"
...     finally:
...         return "No way"

>>> func()
'No way'
```