Exceptions and the Call Stack

Michael Foord
https://agileabstractions.com

- Python Trainer
- Core Python Developer
- Author of IronPython in Action
- Creator of unittest.mock
- Twitter: @voidspace

Exception Handling

- Errors are reported as exceptions
- Unhandled exceptions terminate the program

```
>>> value = int('invalid')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10:
'invalid'
```

- The exception has a type, a message, and a traceback telling you where in the code it occurred
- If the exception happens deep in a program the traceback will show you the whole call stack leading to the error - vital for debugging

The try...except Construct

- Some errors we know how to deal with and can handle
- For this we use the "try-except" statement

```
for line in f:
    fields = line.split()
    try:
        shares = int(fields[1])
    except ValueError:
        print("Couldn't parse", line)
```

- The exception type must match (exact type or subclass) for it to be handled
- If a matching error occurs inside the try block the code in the except block runs

Raising Exceptions

- Sometimes we need to signal an error, for this we use the raise statement

```
raise ValueError("All your data is bad")
```

- If the exception is unhandled the program terminates with message and traceback

```
$ python explode.py
Traceback (most recent call last):
  File "explode.py", line 21, in baz
    bam()
  File "explode.py", line 27, in bam
    raise ValueError("All your data is bad")
ValueError: All your data is bad
```

The Exception Variable

- As well as a message the exception object may have important information
- The except statement can assign the exception object to a variable, using the as syntax

```
try:
    value = int(line[1])
except ValueError as e:
    print("The exception message is:", e)
```

Note: The exception variable ("e" by convention) is cleared after the except finishes. This is to avoid leaking memory, but it's unusual in Python.

Handling Multiple Error Types

- Handling multiple exception types the same way

```
try:
    ...
except (ValueError, TypeError, RuntimeError) as e:
    ...
```

- Handling multiple exception types differently

```
try:
    ...
except ValueError as e:
    ...
except TypeError as e:
    ...
except RuntimeError as e:
    ...
```

- On error the except blocks are checked in order

The else Section

- If there is code you only want to run if there *isn't* an exception you can put it in an else block
- This can help minimise the code inside the try block

```
try:
    value = int(line[1])
except (ValueError, IndexError) as e:
    logger.error('Error parsing "%s" because %s', line, e)
else:
    data.append(value) # only runs if there is no error
```

- To avoid overbroad exception handling follow two important rules
  - Minimise code protected in the try block
  - Be as specific as possible about the error types

The finally Statement

- The finally block runs whether or not there is an error
- Typically used for resource management, like releasing locks and closing connections
- Largely, but not entirely, made obsolete by the with statement

```python
lock = Lock()
lock.acquire()
try:
    ...
finally:
    lock.release()
```

```python
lock = Lock()
with lock:
    ...
```

What Exceptions to Handle?

- Exception handling is for when you are able to deal with the error
- Only handle exceptions you can recover from
- This error here is the *right* exception to signal the problem – leave the caller to handle it if they can

```
>>> read_data('missing.xml')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "data.py", line 7, in read_data
    h = open(filename)
FileNotFoundError: [Errno 2] No such file or directory:
'missing.xml'
```

Re-raising Exceptions

- Sometimes you need to catch an exception, do some processing (close connections or free resources) and then re-raise the exception to let it propagate

```
allocate_resources()
try:
    # Complex operations
    ...
except LookupError as e:
    logger.error("Operation failed: %s", e)
    free_resources()
    raise
```

- A bare raise inside an except re-raises the current exception

Exceptions and the Stack

- Exceptions propagate up the stack
- They can be handled at any level
- Execution continues from the except
- *Or* Python terminates with a non-zero error code and a traceback
- Running code with "python -i script.py" drops you into an interpreter even on error
- "import pdb;pdb.pm()" launches the debugger into the stack frame where the error occurred

Demo with "explode.py"

The Debugger

- pdb is the Python debugger
- pdb.pm() for "postmortem" investigation of errors
- breakpoint() in code or test drops you into the debugger - equivalent of pdb.set_trace()
- Investigate local variables, execute code, step through code or set new breakpoints and continue

pdb commands

help	Get help	c(ontinue)	Continue execution
w(here)	Print stack trace	s(tep)	Execute a single line
d(own)	Move down a stack level	n(ext)	Execute a single line (**)
u(p)	Up a stack level	l(ist)	List the source code
b(reak) loc	Set breakpoint at loc (*)	!statement	Execute statement

(*) e.g. "b 45" set a breakpoint at line 45. "b file.py 45" line 45 in *file.py*
(**) Step through code in the *current function*, don't step into functions calls.

Exception Wrapping

- Sometimes we need to signal an error that comes from some underlying different exception
- Using exception wrapping we can raise an exception without losing the original error or traceback

```
try:
    execute_task(task)
except Exception as e:
    raise TaskError("Task failed") from e
```

- The exception chain can be unwrapped by the caller

```
try:
    send_task(task)
except TaskError as e:
    original_error = e.__cause__
```

Custom Exception Types

- All exception types derive from Exception
- Common for libraries and applications to define custom exceptions (often in a hierarchy)
- Usually all that is required is inheriting from Exception

```python
class NetworkError(Exception):
    pass

class HTTPError(NetworkError):
    pass

class FTPError(NetworkError):
    pass
```

ExceptionGroup

- Raising multiple exceptions:

```
def function():
    e = ExceptionGroup("multiple exceptions", [
            FileNotFoundError("unknown file file1.txt"),
            ValueError("Something went wrong"),
            KeyError("key")
    ])
    raise e
```

- `ExceptionGroup` is an exception:

```
>>> issubclass(ExceptionGroup, Exception)
True
```

Catching Multiple Exceptions

- Syntax new in Python 3.11
- Unpack/handle multiple exceptions
- More than one handler (except block) may run
- except*

```
try:
    some_function()
except* FileNotFoundError as e:
    print("Missing file")
except* ValueError as e:
    print("Something went wrong!")
except* ZeroDivisionError as e:
    print("A different thing went wrong!")
```

The Four Most Confusing Error Messages in Python

- `UnboundLocalError`
- `TypeError`: multiple bases have instance lay-out conflict
- `TypeError`: Cannot create a consistent method resolution order (MRO) for bases
- `TypeError`: metaclass conflict

Default Arguments

- Sometimes you want an optional argument

```
def read_prices(filename, debug=False):
    ...
```

- If a default value is assigned, the argument is optional in function calls

```
d = read_prices('prices.csv')
e = read_prices('prices.dat', True)
```

- Note: arguments with defaults must appear at the end of the argument list (all required arguments go first)

Calling a Function

- Consider a simple function

```
def read_prices(filename, debug):
    ...
```

- Calling with "positional" args

```
prices = read_prices('prices.csv', True)
```

- Calling with "keyword" arguments

```
prices = read_prices(filename='prices.csv',
                     debug=True)
```

- Calling with mixed arguments

```
prices = read_prices('prices.csv', debug=True)
```

Optional/Keyword Arguments

- Arguments with default values are useful for functions that have optional features/flags

```
def parse_data(data, debug=False, ignore_errors=False):
    ...
```

- Compare and contrast calling styles:

```
parse_data(data, False, True)          # ?????

parse_data(data, ignore_errors=True)
parse_data(data, debug=True)
parse_data(data, debug=True, ignore_errors=True)
```

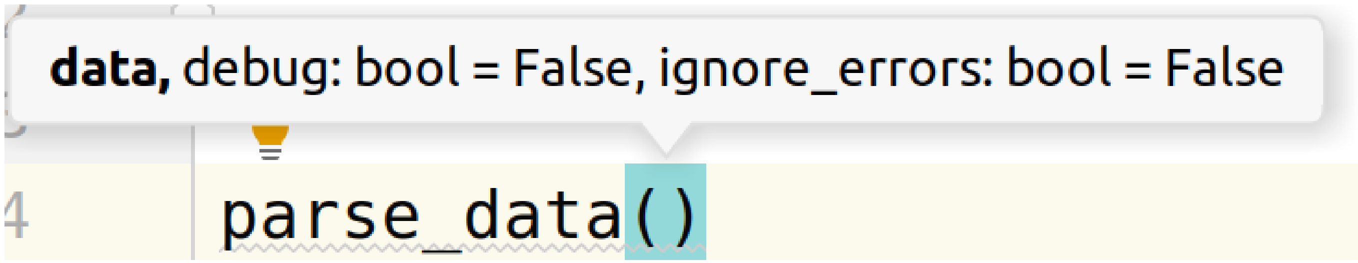
- Keyword arguments improve code clarity
- Optional arguments can be added to functions without breaking existing uses (backwards compatibility)

Design Tip

- Always give short meaningful names to function arguments
- The argument names are part of the API of the function, a design consideration
- Someone using a function may want to use the keyword calling style

```
d = read_prices('prices.csv', debug=True)
```

- Python development tools will show the names in help features and documentation



Return Values

- return returns a value

```
def square(x):
    return x*x
```

- return without a value returns None

```
def bar(x):
    statements
    return

a = bar(4)      # a = None
```

- A function without an explicit return, returns None

```
def foo(x):
    statements
    statements

a = foo(9)      # a = None
```

Multiple Return Values

- A function may return multiple values by returning a tuple

```
def divide(a,b):
    q = a // b      # Quotient
    r = a % b       # Remainder
    return q, r     # Return a tuple
```

- Usage examples:

```
x, y = divide(37, 5)        # x = 7, y = 2

x = divide(37, 5)           # x = (7, 2)
```

- Unpacking the returned tuple in the call looks like multiple return values

Positional and Keyword Only Arguments

- Python function signatures are now very rich
- We can now express positional and keyword only arguments
- Positional only arguments (mostly for compatibility with C functions) added in Python 3.8
- Keyword only arguments were new in Python 3.0

```
>>> def foo(data, /, *, debug=False):
...     pass
...
>>> foo(1, debug=True)
>>> foo(data=2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got some positional-only arguments passed
as keyword arguments: 'data'
>>> foo(3, False)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes 1 positional argument but 2 were
given
```