```
    handle = open(filename, encoding='utf-8') # Maybe OSError
    try:
        print('* Reading data')
        return handle.read()  # Maybe UnicodeDecodeError
    finally:
        print('* Calling close()')
        handle.close()        # Always runs after try block
```

Any exception raised by the `read` method will always propagate up to the calling code, but the `close` method of `handle` in the `finally` block will run first:

```
filename = 'random_data.txt'

with open(filename, 'wb') as f:
    f.write(b'\xf1\xf2\xf3\xf4\xf5')  # Invalid utf-8

data = try_finally_example(filename)
```

```
>>>
* Opening file
* Reading data
* Calling close()
Traceback ...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf1 in
➥position 0: invalid continuation byte
```

You must call `open` before the `try` block because exceptions that occur when opening the file (like `OSError` if the file does not exist) should skip the `finally` block entirely:

```
try_finally_example('does_not_exist.txt')
```

```
>>>
* Opening file
Traceback ...
FileNotFoundError: [Errno 2] No such file or directory:
➥'does_not_exist.txt'
```

`else` Blocks

Use `try/except/else` to make it clear which exceptions will be handled by your code and which exceptions will propagate up. When the `try` block doesn't raise an exception, the `else` block runs. The `else` block helps you minimize the amount of code in the `try` block, which is good for isolating potential exception causes and improves

readability. For example, say that I want to load JSON dictionary data from a string and return the value of a key it contains:

```
import json

def load_json_key(data, key):
    try:
        print('* Loading JSON data')
        result_dict = json.loads(data)  # May raise ValueError
    except ValueError as e:
        print('* Handling ValueError')
        raise KeyError(key) from e
    else:
        print('* Looking up key')
        return result_dict[key]         # May raise KeyError
```

In the successful case, the JSON data is decoded in the `try` block, and then the key lookup occurs in the `else` block:

```
assert load_json_key('{"foo": "bar"}', 'foo') == 'bar'
```

```
>>>
* Loading JSON data
* Looking up key
```

If the input data isn't valid JSON, then decoding with `json.loads` raises a `ValueError`. The exception is caught by the `except` block and handled:

```
load_json_key('{"foo": bad payload', 'foo')
```

```
>>>
* Loading JSON data
* Handling ValueError
Traceback ...
JSONDecodeError: Expecting value: line 1 column 9 (char 8)

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

Traceback ...
KeyError: 'foo'
```

If the key lookup raises any exceptions, they propagate up to the caller because they are outside the `try` block. The `else` clause ensures that what follows the `try/except` is visually distinguished from the `except` block. This makes the exception propagation behavior clear:

```
load_json_key('{"foo": "bar"}', 'does not exist')
```

```
>>>
* Loading JSON data
* Looking up key
Traceback ...
KeyError: 'does not exist'
```

Everything Together

Use `try/except/else/finally` when you want to do it all in one compound statement. For example, say that I want to read a description of work to do from a file, process it, and then update the file in-place. Here, the `try` block is used to read the file and process it; the `except` block is used to handle exceptions from the `try` block that are expected; the `else` block is used to update the file in place and allow related exceptions to propagate up; and the `finally` block cleans up the file handle:

```
UNDEFINED = object()

def divide_json(path):
    print('* Opening file')
    handle = open(path, 'r+')   # May raise OSError
    try:
        print('* Reading data')
        data = handle.read()    # May raise UnicodeDecodeError
        print('* Loading JSON data')
        op = json.loads(data)   # May raise ValueError
        print('* Performing calculation')
        value = (
            op['numerator'] /
            op['denominator'])  # May raise ZeroDivisionError
    except ZeroDivisionError as e:
        print('* Handling ZeroDivisionError')
        return UNDEFINED
    else:
        print('* Writing calculation')
        op['result'] = value
        result = json.dumps(op)
        handle.seek(0)          # May raise OSError
        handle.write(result)    # May raise OSError
        return value
    finally:
        print('* Calling close()')
        handle.close()          # Always runs
```

In the successful case, the try, else, and finally blocks run:

```
temp_path = 'random_data.json'

with open(temp_path, 'w') as f:
    f.write('{"numerator": 1, "denominator": 10}')

assert divide_json(temp_path) == 0.1
```

```
>>>
* Opening file
* Reading data
* Loading JSON data
* Performing calculation
* Writing calculation
* Calling close()
```

If the calculation is invalid, the try, except, and finally blocks run, but the else block does not:

```
with open(temp_path, 'w') as f:
    f.write('{"numerator": 1, "denominator": 0}')

assert divide_json(temp_path) is UNDEFINED
```

```
>>>
* Opening file
* Reading data
* Loading JSON data
* Performing calculation
* Handling ZeroDivisionError
* Calling close()
```

If the JSON data was invalid, the try block runs and raises an exception, the finally block runs, and then the exception is propagated up to the caller. The except and else blocks do not run:

```
with open(temp_path, 'w') as f:
    f.write('{"numerator": 1 bad data')

divide_json(temp_path)
```

```
>>>
* Opening file
* Reading data
* Loading JSON data
* Calling close()
Traceback ...
JSONDecodeError: Expecting ',' delimiter: line 1 column 17
➥(char 16)
```

This layout is especially useful because all of the blocks work together in intuitive ways. For example, here I simulate this by running the `divide_json` function at the same time that my hard drive runs out of disk space:

```
with open(temp_path, 'w') as f:
    f.write('{"numerator": 1, "denominator": 10}')

divide_json(temp_path)
```

```
>>>
* Opening file
* Reading data
* Loading JSON data
* Performing calculation
* Writing calculation
* Calling close()
Traceback ...
OSError: [Errno 28] No space left on device
```

When the exception was raised in the `else` block while rewriting the result data, the `finally` block still ran and closed the file handle as expected.

Things to Remember

- The `try/finally` compound statement lets you run cleanup code regardless of whether exceptions were raised in the `try` block.
- The `else` block helps you minimize the amount of code in `try` blocks and visually distinguish the success case from the `try/except` blocks.
- An `else` block can be used to perform additional actions after a successful `try` block but before common cleanup in a `finally` block.

Item 66: Consider `contextlib` and `with` Statements for Reusable `try/finally` Behavior

The `with` statement in Python is used to indicate when code is running in a special context. For example, mutual-exclusion locks (see Item 54: "Use `Lock` to Prevent Data Races in Threads") can be used in `with` statements to indicate that the indented code block runs only while the lock is held:

```
from threading import Lock

lock = Lock()
```

```
with lock:
    # Do something while maintaining an invariant
    ...
```

The example above is equivalent to this try/finally construction because the Lock class properly enables the with statement (see Item 65: "Take Advantage of Each Block in try/except/else/finally" for more about try/finally):

```
lock.acquire()
try:
    # Do something while maintaining an invariant
    ...
finally:
    lock.release()
```

The with statement version of this is better because it eliminates the need to write the repetitive code of the try/finally construction, and it ensures that you don't forget to have a corresponding release call for every acquire call.

It's easy to make your objects and functions work in with statements by using the contextlib built-in module. This module contains the contextmanager decorator (see Item 26: "Define Function Decorators with functools.wraps" for background), which lets a simple function be used in with statements. This is much easier than defining a new class with the special methods __enter__ and __exit__ (the standard way).

For example, say that I want a region of code to have more debug logging sometimes. Here, I define a function that does logging at two severity levels:

```
import logging

def my_function():
    logging.debug('Some debug data')
    logging.error('Error log here')
    logging.debug('More debug data')
```

The default log level for my program is WARNING, so only the error message will print to screen when I run the function:

```
my_function()
```

```
>>>
Error log here
```

I can elevate the log level of this function temporarily by defining a context manager. This helper function boosts the logging severity level before running the code in the with block and reduces the logging severity level afterward:

```
from contextlib import contextmanager

@contextmanager
def debug_logging(level):
    logger = logging.getLogger()
    old_level = logger.getEffectiveLevel()
    logger.setLevel(level)
    try:
        yield
    finally:
        logger.setLevel(old_level)
```

The yield expression is the point at which the with block's contents will execute (see Item 30: "Consider Generators Instead of Returning Lists" for background). Any exceptions that happen in the with block will be re-raised by the yield expression for you to catch in the helper function (see Item 35: "Avoid Causing State Transitions in Generators with throw" for how that works).

Now, I can call the same logging function again but in the debug_logging context. This time, all of the debug messages are printed to the screen during the with block. The same function running outside the with block won't print debug messages:

```
with debug_logging(logging.DEBUG):
    print('* Inside:')
    my_function()

print('* After:')
my_function()
```

```
>>>
* Inside:
Some debug data
Error log here
More debug data
* After:
Error log here
```

Using with Targets

The context manager passed to a with statement may also return an object. This object is assigned to a local variable in the as part of the

compound statement. This gives the code running in the with block the ability to directly interact with its context.

For example, say I want to write a file and ensure that it's always closed correctly. I can do this by passing open to the with statement. open returns a file handle for the as target of with, and it closes the handle when the with block exits:

```
with open('my_output.txt', 'w') as handle:
    handle.write('This is some data!')
```

This approach is more Pythonic than manually opening and closing the file handle every time. It gives you confidence that the file is eventually closed when execution leaves the with statement. By highlighting the critical section, it also encourages you to reduce the amount of code that executes while the file handle is open, which is good practice in general.

To enable your own functions to supply values for as targets, all you need to do is yield a value from your context manager. For example, here I define a context manager to fetch a Logger instance, set its level, and then yield it as the target:

```
@contextmanager
def log_level(level, name):
    logger = logging.getLogger(name)
    old_level = logger.getEffectiveLevel()
    logger.setLevel(level)
    try:
        yield logger
    finally:
        logger.setLevel(old_level)
```

Calling logging methods like debug on the as target produces output because the logging severity level is set low enough in the with block on that specific Logger instance. Using the logging module directly won't print anything because the default logging severity level for the default program logger is WARNING:

```
with log_level(logging.DEBUG, 'my-log') as logger:
    logger.debug(f'This is a message for {logger.name}!')
    logging.debug('This will not print')

>>>
This is a message for my-log!
```

After the with statement exits, calling debug logging methods on the Logger named 'my-log' will not print anything because the default

logging severity level has been restored. Error log messages will always print:

```
logger = logging.getLogger('my-log')
logger.debug('Debug will not print')
logger.error('Error will print')
```

```
>>>
Error will print
```

Later, I can change the name of the logger I want to use by simply updating the `with` statement. This will point the `Logger` that's the `as` target in the `with` block to a different instance, but I won't have to update any of my other code to match:

```
with log_level(logging.DEBUG, 'other-log') as logger:
    logger.debug(f'This is a message for {logger.name}!')
    logging.debug('This will not print')
```

```
>>>
This is a message for other-log!
```

This isolation of state and decoupling between creating a context and acting within that context is another benefit of the `with` statement.

Things to Remember

- The `with` statement allows you to reuse logic from `try/finally` blocks and reduce visual noise.
- The `contextlib` built-in module provides a `contextmanager` decorator that makes it easy to use your own functions in `with` statements.
- The value yielded by context managers is supplied to the `as` part of the `with` statement. It's useful for letting your code directly access the cause of a special context.

Item 67: Use `datetime` Instead of `time` for Local Clocks

Coordinated Universal Time (UTC) is the standard, time-zone-independent representation of time. UTC works great for computers that represent time as seconds since the UNIX epoch. But UTC isn't ideal for humans. Humans reference time relative to where they're currently located. People say "noon" or "8 am" instead of "UTC 15:00 minus 7 hours." If your program handles time, you'll probably find yourself converting time between UTC and local clocks for the sake of human understanding.

Python provides two ways of accomplishing time zone conversions. The old way, using the time built-in module, is terribly error prone. The new way, using the datetime built-in module, works great with some help from the community-built package named pytz.

You should be acquainted with both time and datetime to thoroughly understand why datetime is the best choice and time should be avoided.

The time Module

The localtime function from the time built-in module lets you convert a UNIX timestamp (seconds since the UNIX epoch in UTC) to a local time that matches the host computer's time zone (Pacific Daylight Time in my case). This local time can be printed in human-readable format using the strftime function:

```
import time

now = 1552774475
local_tuple = time.localtime(now)
time_format = '%Y-%m-%d %H:%M:%S'
time_str = time.strftime(time_format, local_tuple)
print(time_str)
```

```
>>>
2019-03-16 15:14:35
```

You'll often need to go the other way as well, starting with user input in human-readable local time and converting it to UTC time. You can do this by using the strptime function to parse the time string, and then calling mktime to convert local time to a UNIX timestamp:

```
time_tuple = time.strptime(time_str, time_format)
utc_now = time.mktime(time_tuple)
print(utc_now)
```

```
>>>
1552774475.0
```

How do you convert local time in one time zone to local time in another time zone? For example, say that I'm taking a flight between San Francisco and New York, and I want to know what time it will be in San Francisco when I've arrived in New York.

I might initially assume that I can directly manipulate the return values from the time, localtime, and strptime functions to do time zone conversions. But this is a very bad idea. Time zones change all the time due to local laws. It's too complicated to manage yourself, especially if you want to handle every global city for flight departures and arrivals.

Many operating systems have configuration files that keep up with the time zone changes automatically. Python lets you use these time zones through the `time` module if your platform supports it. On other platforms, such as Windows, some time zone functionality isn't available from `time` at all. For example, here I parse a departure time from the San Francisco time zone, Pacific Daylight Time (PDT):

```
import os

if os.name == 'nt':
    print("This example doesn't work on Windows")
else:
    parse_format = '%Y-%m-%d %H:%M:%S %Z'
    depart_sfo = '2019-03-16 15:45:16 PDT'
    time_tuple = time.strptime(depart_sfo, parse_format)
    time_str = time.strftime(time_format, time_tuple)
    print(time_str)
```

```
>>>
2019-03-16 15:45:16
```

After seeing that `'PDT'` works with the `strptime` function, I might also assume that other time zones known to my computer will work. Unfortunately, this isn't the case. `strptime` raises an exception when it sees Eastern Daylight Time (EDT), which is the time zone for New York:

```
arrival_nyc = '2019-03-16 23:33:24 EDT'
time_tuple = time.strptime(arrival_nyc, time_format)
```

```
>>>
Traceback ...
ValueError: unconverted data remains:  EDT
```

The problem here is the platform-dependent nature of the `time` module. Its behavior is determined by how the underlying C functions work with the host operating system. This makes the functionality of the `time` module unreliable in Python. The `time` module fails to consistently work properly for multiple local times. Thus, you should avoid using the `time` module for this purpose. If you must use `time`, use it only to convert between UTC and the host computer's local time. For all other types of conversions, use the `datetime` module.

The `datetime` Module

The second option for representing times in Python is the `datetime` class from the `datetime` built-in module. Like the `time` module, `datetime` can be used to convert from the current time in UTC to local time.

Here, I convert the present time in UTC to my computer's local time, PDT:

```
from datetime import datetime, timezone

now = datetime(2019, 3, 16, 22, 14, 35)
now_utc = now.replace(tzinfo=timezone.utc)
now_local = now_utc.astimezone()
print(now_local)
```

```
>>>
2019-03-16 15:14:35-07:00
```

The datetime module can also easily convert a local time back to a UNIX timestamp in UTC:

```
time_str = '2019-03-16 15:14:35'
now = datetime.strptime(time_str, time_format)
time_tuple = now.timetuple()
utc_now = time.mktime(time_tuple)
print(utc_now)
```

```
>>>
1552774475.0
```

Unlike the time module, the datetime module has facilities for reliably converting from one local time to another local time. However, datetime only provides the machinery for time zone operations with its tzinfo class and related methods. The Python default installation is missing time zone definitions besides UTC.

Luckily, the Python community has addressed this gap with the pytz module that's available for download from the Python Package Index (see Item 82: "Know Where to Find Community-Built Modules" for how to install it). pytz contains a full database of every time zone definition you might need.

To use pytz effectively, you should always convert local times to UTC first. Perform any datetime operations you need on the UTC values (such as offsetting). Then, convert to local times as a final step.

For example, here I convert a New York City flight arrival time to a UTC datetime. Although some of these calls seem redundant, all of them are necessary when using pytz:

```
import pytz

arrival_nyc = '2019-03-16 23:33:24'
nyc_dt_naive = datetime.strptime(arrival_nyc, time_format)
```

```
eastern = pytz.timezone('US/Eastern')
nyc_dt = eastern.localize(nyc_dt_naive)
utc_dt = pytz.utc.normalize(nyc_dt.astimezone(pytz.utc))
print(utc_dt)
```

```
>>>
2019-03-17 03:33:24+00:00
```

Once I have a UTC `datetime`, I can convert it to San Francisco local time:

```
pacific = pytz.timezone('US/Pacific')
sf_dt = pacific.normalize(utc_dt.astimezone(pacific))
print(sf_dt)
```

```
>>>
2019-03-16 20:33:24-07:00
```

Just as easily, I can convert it to the local time in Nepal:

```
nepal = pytz.timezone('Asia/Katmandu')
nepal_dt = nepal.normalize(utc_dt.astimezone(nepal))
print(nepal_dt)
```

```
>>>
2019-03-17 09:18:24+05:45
```

With `datetime` and `pytz`, these conversions are consistent across all environments, regardless of what operating system the host computer is running.

Things to Remember

- Avoid using the `time` module for translating between different time zones.
- Use the `datetime` built-in module along with the `pytz` community module to reliably convert between times in different time zones.
- Always represent time in UTC and do conversions to local time as the very final step before presentation.

Item 68: Make `pickle` Reliable with `copyreg`

The `pickle` built-in module can serialize Python objects into a stream of bytes and deserialize bytes back into objects. Pickled byte streams shouldn't be used to communicate between untrusted parties. The purpose of `pickle` is to let you pass Python objects between programs that you control over binary channels.

Note

The pickle module's serialization format is unsafe by design. The serialized data contains what is essentially a program that describes how to reconstruct the original Python object. This means a malicious pickle payload could be used to compromise any part of a Python program that attempts to deserialize it.

In contrast, the json module is safe by design. Serialized JSON data contains a simple description of an object hierarchy. Deserializing JSON data does not expose a Python program to additional risk. Formats like JSON should be used for communication between programs or people who don't trust each other.

For example, say that I want to use a Python object to represent the state of a player's progress in a game. The game state includes the level the player is on and the number of lives they have remaining:

```
class GameState:
    def __init__(self):
        self.level = 0
        self.lives = 4
```

The program modifies this object as the game runs:

```
state = GameState()
state.level += 1  # Player beat a level
state.lives -= 1  # Player had to try again

print(state.__dict__)
```

```
>>>
{'level': 1, 'lives': 3}
```

When the user quits playing, the program can save the state of the game to a file so it can be resumed at a later time. The pickle module makes it easy to do this. Here, I use the dump function to write the GameState object to a file:

```
import pickle

state_path = 'game_state.bin'
with open(state_path, 'wb') as f:
    pickle.dump(state, f)
```

Later, I can call the load function with the file and get back the GameState object as if it had never been serialized:

```
with open(state_path, 'rb') as f:
    state_after = pickle.load(f)

print(state_after.__dict__)
```

```
>>>
{'level': 1, 'lives': 3}
```

The problem with this approach is what happens as the game's features expand over time. Imagine that I want the player to earn points toward a high score. To track the player's points, I'd add a new field to the `GameState` class

```
class GameState:
    def __init__(self):
        self.level = 0
        self.lives = 4
        self.points = 0  # New field
```

Serializing the new version of the `GameState` class using `pickle` will work exactly as before. Here, I simulate the round-trip through a file by serializing to a string with `dumps` and back to an object with `loads`:

```
state = GameState()
serialized = pickle.dumps(state)
state_after = pickle.loads(serialized)
print(state_after.__dict__)
```

```
>>>
{'level': 0, 'lives': 4, 'points': 0}
```

But what happens to older saved `GameState` objects that the user may want to resume? Here, I unpickle an old game file by using a program with the new definition of the `GameState` class:

```
with open(state_path, 'rb') as f:
    state_after = pickle.load(f)

print(state_after.__dict__)
```

```
>>>
{'level': 1, 'lives': 3}
```

The `points` attribute is missing! This is especially confusing because the returned `object` is an instance of the new `GameState` class:

```
assert isinstance(state_after, GameState)
```

This behavior is a byproduct of the way the `pickle` module works. Its primary use case is making object serialization easy. As soon as your use of `pickle` moves beyond trivial usage, the module's functionality starts to break down in surprising ways.

Fixing these problems is straightforward using the `copyreg` built-in module. The `copyreg` module lets you register the functions responsible

for serializing and deserializing Python objects, allowing you to control the behavior of pickle and make it more reliable.

Default Attribute Values

In the simplest case, you can use a constructor with default arguments (see Item 23: "Provide Optional Behavior with Keyword Arguments" for background) to ensure that GameState objects will always have all attributes after unpickling. Here, I redefine the constructor this way:

```
class GameState:
    def __init__(self, level=0, lives=4, points=0):
        self.level = level
        self.lives = lives
        self.points = points
```

To use this constructor for pickling, I define a helper function that takes a GameState object and turns it into a tuple of parameters for the copyreg module. The returned tuple contains the function to use for unpickling and the parameters to pass to the unpickling function:

```
def pickle_game_state(game_state):
    kwargs = game_state.__dict__
    return unpickle_game_state, (kwargs,)
```

Now, I need to define the unpickle_game_state helper. This function takes serialized data and parameters from pickle_game_state and returns the corresponding GameState object. It's a tiny wrapper around the constructor:

```
def unpickle_game_state(kwargs):
    return GameState(**kwargs)
```

Now, I register these functions with the copyreg built-in module:

```
import copyreg

copyreg.pickle(GameState, pickle_game_state)
```

After registration, serializing and deserializing works as before:

```
state = GameState()
state.points += 1000
serialized = pickle.dumps(state)
state_after = pickle.loads(serialized)
print(state_after.__dict__)
```

```
>>>
{'level': 0, 'lives': 4, 'points': 1000}
```

With this registration done, now I'll change the definition of GameState again to give the player a count of magic spells to use. This change is similar to when I added the points field to GameState:

```
class GameState:
    def __init__(self, level=0, lives=4, points=0, magic=5):
        self.level = level
        self.lives = lives
        self.points = points
        self.magic = magic  # New field
```

But unlike before, deserializing an old GameState object will result in valid game data instead of missing attributes. This works because unpickle_game_state calls the GameState constructor directly instead of using the pickle module's default behavior of saving and restoring only the attributes that belong to an object. The GameState constructor's keyword arguments have default values that will be used for any parameters that are missing. This causes old game state files to receive the default value for the new magic field when they are deserialized:

```
print('Before:', state.__dict__)
state_after = pickle.loads(serialized)
print('After: ', state_after.__dict__)
```

```
>>>
Before: {'level': 0, 'lives': 4, 'points': 1000}
After:  {'level': 0, 'lives': 4, 'points': 1000, 'magic': 5}
```

Versioning Classes

Sometimes you need to make backward-incompatible changes to your Python objects by removing fields. Doing so prevents the default argument approach above from working.

For example, say I realize that a limited number of lives is a bad idea, and I want to remove the concept of lives from the game. Here, I redefine the GameState class to no longer have a lives field:

```
class GameState:
    def __init__(self, level=0, points=0, magic=5):
        self.level = level
        self.points = points
        self.magic = magic
```

The problem is that this breaks deserialization of old game data. All fields from the old data, even ones removed from the class, will be passed to the GameState constructor by the unpickle_game_state function:

```
pickle.loads(serialized)
```

```
>>>
Traceback ...
TypeError: __init__() got an unexpected keyword argument
➥'lives'
```

I can fix this by adding a version parameter to the functions supplied to copyreg. New serialized data will have a version of 2 specified when pickling a new GameState object:

```
def pickle_game_state(game_state):
    kwargs = game_state.__dict__
    kwargs['version'] = 2
    return unpickle_game_state, (kwargs,)
```

Old versions of the data will not have a version argument present, which means I can manipulate the arguments passed to the GameState constructor accordingly:

```
def unpickle_game_state(kwargs):
    version = kwargs.pop('version', 1)
    if version == 1:
        del kwargs['lives']
    return GameState(**kwargs)
```

Now, deserializing an old object works properly:

```
copyreg.pickle(GameState, pickle_game_state)
print('Before:', state.__dict__)
state_after = pickle.loads(serialized)
print('After: ', state_after.__dict__)
```

```
>>>
Before: {'level': 0, 'lives': 4, 'points': 1000}
After:  {'level': 0, 'points': 1000, 'magic': 5}
```

I can continue using this approach to handle changes between future versions of the same class. Any logic I need to adapt an old version of the class to a new version of the class can go in the unpickle_game_state function.

Stable Import Paths

One other issue you may encounter with pickle is breakage from renaming a class. Often over the life cycle of a program, you'll refactor your code by renaming classes and moving them to other modules. Unfortunately, doing so breaks the pickle module unless you're careful.

Here, I rename the GameState class to BetterGameState and remove the old class from the program entirely:

```
class BetterGameState:
    def __init__(self, level=0, points=0, magic=5):
        self.level = level
        self.points = points
        self.magic = magic
```

Attempting to deserialize an old GameState object now fails because the class can't be found:

```
pickle.loads(serialized)
```

```
>>>
Traceback ...
AttributeError: Can't get attribute 'GameState' on <module
➥'__main__' from 'my_code.py'>
```

The cause of this exception is that the import path of the serialized object's class is encoded in the pickled data:

```
print(serialized)
```

```
>>>
b'\x80\x04\x95A\x00\x00\x00\x00\x00\x00\x00\x8c\x08__main__
➥\x94\x8c\tGameState\x94\x93\x94)\x81\x94}\x94(\x8c\x05level
➥\x94K\x00\x8c\x06points\x94K\x00\x8c\x05magic\x94K\x05ub.'
```

The solution is to use copyreg again. I can specify a stable identifier for the function to use for unpickling an object. This allows me to transition pickled data to different classes with different names when it's deserialized. It gives me a level of indirection:

```
copyreg.pickle(BetterGameState, pickle_game_state)
```

After I use copyreg, you can see that the import path to unpickle_game_state is encoded in the serialized data instead of BetterGameState:

```
state = BetterGameState()
serialized = pickle.dumps(state)
print(serialized)
```

```
>>>
b'\x80\x04\x95W\x00\x00\x00\x00\x00\x00\x00\x8c\x08__main__
➥\x94\x8c\x13unpickle_game_state\x94\x93\x94}\x94(\x8c
➥\x05level\x94K\x00\x8c\x06points\x94K\x00\x8c\x05magic\x94K
➥\x05\x8c\x07version\x94K\x02u\x85\x94R\x94.'
```

The only gotcha is that I can't change the path of the module in which the `unpickle_game_state` function is present. Once I serialize data with a function, it must remain available on that import path for deserialization in the future.

Things to Remember

- The `pickle` built-in module is useful only for serializing and deserializing objects between trusted programs.
- Deserializing previously pickled objects may break if the classes involved have changed over time (e.g., attributes have been added or removed).
- Use the `copyreg` built-in module with `pickle` to ensure backward compatibility for serialized objects.

Item 69: Use decimal When Precision Is Paramount

Python is an excellent language for writing code that interacts with numerical data. Python's integer type can represent values of any practical size. Its double-precision floating point type complies with the IEEE 754 standard. The language also provides a standard complex number type for imaginary values. However, these aren't enough for every situation.

For example, say that I want to compute the amount to charge a customer for an international phone call. I know the time in minutes and seconds that the customer was on the phone (say, 3 minutes 42 seconds). I also have a set rate for the cost of calling Antarctica from the United States ($1.45/minute). What should the charge be?

With floating point math, the computed charge seems reasonable

```
rate = 1.45
seconds = 3*60 + 42
cost = rate * seconds / 60
print(cost)
```

```
>>>
5.364999999999999
```

The result is 0.0001 short of the correct value (5.365) due to how IEEE 754 floating point numbers are represented. I might want to round up this value to 5.37 to properly cover all costs incurred by the customer. However, due to floating point error, rounding to the nearest whole cent actually reduces the final charge (from 5.364 to 5.36) instead of increasing it (from 5.365 to 5.37):

```
print(round(cost, 2))
```

```
>>>
5.36
```

The solution is to use the `Decimal` class from the `decimal` built-in module. The `Decimal` class provides fixed point math of 28 decimal places by default. It can go even higher, if required. This works around the precision issues in IEEE 754 floating point numbers. The class also gives you more control over rounding behaviors.

For example, redoing the Antarctica calculation with `Decimal` results in the exact expected charge instead of an approximation:

```
from decimal import Decimal

rate = Decimal('1.45')
seconds = Decimal(3*60 + 42)
cost = rate * seconds / Decimal(60)
print(cost)
```

```
>>>
5.365
```

`Decimal` instances can be given starting values in two different ways. The first way is by passing a `str` containing the number to the `Decimal` constructor. This ensures that there is no loss of precision due to the inherent nature of Python floating point numbers. The second way is by directly passing a `float` or an `int` instance to the constructor. Here, you can see that the two construction methods result in different behavior.

```
print(Decimal('1.45'))
print(Decimal(1.45))
```

```
>>>
1.45
1.449999999999999955591079014993738383054733276367187 5
```

The same problem doesn't happen if I supply integers to the `Decimal` constructor:

```
print('456')
print(456)
```

```
>>>
456
456
```

If you care about exact answers, err on the side of caution and use the `str` constructor for the `Decimal` type.

Getting back to the phone call example, say that I also want to support very short phone calls between places that are much cheaper to connect (like Toledo and Detroit). Here, I compute the charge for a phone call that was 5 seconds long with a rate of $0.05/minute:

```
rate = Decimal('0.05')
seconds = Decimal('5')
small_cost = rate * seconds / Decimal(60)
print(small_cost)
```

```
>>>
0.004166666666666666666666666667
```

The result is so low that it is decreased to zero when I try to round it to the nearest whole cent. This won't do!

```
print(round(small_cost, 2))
```

```
>>>
0.00
```

Luckily, the `Decimal` class has a built-in function for rounding to exactly the decimal place needed with the desired rounding behavior. This works for the higher cost case from earlier:

```
from decimal import ROUND_UP

rounded = cost.quantize(Decimal('0.01'), rounding=ROUND_UP)
print(f'Rounded {cost} to {rounded}')
```

```
>>>
Rounded 5.365 to 5.37
```

Using the `quantize` method this way also properly handles the small usage case for short, cheap phone calls:.

```
rounded = small_cost.quantize(Decimal('0.01'),
                              rounding=ROUND_UP)
print(f'Rounded {small_cost} to {rounded}')
```

```
>>>
Rounded 0.004166666666666666666666666667 to 0.01
```

While `Decimal` works great for fixed point numbers, it still has limitations in its precision (e.g., 1/3 will be an approximation). For representing rational numbers with no limit to precision, consider using the `Fraction` class from the `fractions` built-in module.

Things to Remember

- Python has built-in types and classes in modules that can represent practically every type of numerical value.
- The `Decimal` class is ideal for situations that require high precision and control over rounding behavior, such as computations of monetary values.
- Pass `str` instances to the `Decimal` constructor instead of `float` instances if it's important to compute exact answers and not floating point approximations.

Item 70: Profile Before Optimizing

The dynamic nature of Python causes surprising behaviors in its runtime performance. Operations you might assume would be slow are actually very fast (e.g., string manipulation, generators). Language features you might assume would be fast are actually very slow (e.g., attribute accesses, function calls). The true source of slowdowns in a Python program can be obscure.

The best approach is to ignore your intuition and directly measure the performance of a program before you try to optimize it. Python provides a built-in *profiler* for determining which parts of a program are responsible for its execution time. This means you can focus your optimization efforts on the biggest sources of trouble and ignore parts of the program that don't impact speed (i.e., follow Amdahl's law).

For example, say that I want to determine why an algorithm in a program is slow. Here, I define a function that sorts a `list` of data using an insertion sort:

```
def insertion_sort(data):
    result = []
    for value in data:
        insert_value(result, value)
    return result
```

The core mechanism of the insertion sort is the function that finds the insertion point for each piece of data. Here, I define an extremely inefficient version of the `insert_value` function that does a linear scan over the input array:

```
def insert_value(array, value):
    for i, existing in enumerate(array):
        if existing > value:
            array.insert(i, value)
```