Chapter 08 - Robustness and Performance

May 8, 2021

0.1 Item 65: Take Advantage of Each Block in `try/except/else/finally`

0.1.1 `finally` Blocks

- use `try/finally` when you want exceptions to propagate up but also want to run cleanup code even when exceptions occur
- one common usage of `try/finally` is for reliably closing file handles

[1]:
```
def try_finally_example(filename):
    print('* Opening file')

handle = open(filename, encoding='utf-8')
try:
    print('* Reading data')
    return handle.read()
finally:
    print('* Calling close()')
    handle.clode() # always run after try block
```

```
  File "<ipython-input-1-0c3f2d5b473d>", line 4
    handle = open(filename, encoding='utf-8)
                                            ^

SyntaxError: EOL while scanning string literal
```

0.1.2 `else` Blocks

- use `try/else` to make it clear which excaptions will be handled by your code and which exceptions will propagate up
- when the `try` block does not raise an exception, the `else` block runs
- the `else` block helps you minimize the amount of code in the `try` block

[2]:
```
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
- the `else` clause ensures that what follows the `try/except` is visually distinguished from the `except` block
- this makes the exception proagation behavior clear

0.1.3 Everything Together

- use `try/except/finally` when you want to do it all in one compound statement

- 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 exception to proapagate up
- the `finally` block cleans up the file handle

[3]:
```
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.load(data) # May raise ValueError
        print('* Performing calculation')

    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
    finally:
        print('* Calling close()')
        handle.close()         # Always runs
```

0.2 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
- mutual-exclusion locks can be used in `with` statements to indicate that he indented code block runs only while the lock is held

[7]:
```
from threading import Lock

lock = Lock()

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

- the example above is equivalent to this `try/finally` construction because the `Lock` class properly enables the `with` statement

[8]:
```
lock.acquire()
try:
    # Do something while maintaining na invariant
    pass
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 dont forget to have a corresponding `release` call for every acquire call

- its easy to make your objects and functions work in `with` statements by using the `contextlib` built-in module
- this module contains the `contextmanager` decorator 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__`

- for example, say I want a region of code to have more debug logging sometimes, we can define a function that does logging at two severity levels

[9]:
```
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

[11]:
```
my_function()
```

```
ERROR:root:Error log here
```

- we 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

[12]:
```
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` blocks context will execute
- any exceptions that happen in the `with` block will be re-raised by the `yield` expression for you to catch in the helper function

- now we 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 wont print debug message

[13]:
```
with debug_logging(logging.DEBUG):
    print('* Inside: ')
    my_function()

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

```
DEBUG:root:Some debug data
ERROR:root:Error log here
DEBUG:root:More debug data
ERROR:root:Error log here
```

```
* Inside:
* After:
```

0.3 Using `with` Targets

- the context manager passed to a `with` statement may also return an object
- the 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 ite context

- for example, say we want to write to a file and ensure that its always closed correctly
- we cand o this by passing `open` to the `with` statement

- `open` returns a file handle for the `as` target of `with` and it colses the handle when the `with` block exits

[14]:
```
with open('my_output.txt', 'w') as handle:
    handle.write('This is some data!')
```

- to enable your own function sto supply values for as targets, all you need to do is `yield` a value from your context manager
- for example, below we define a context manager to fetch a `Logger` instance, set its level and then `yield` it a s the target

[15]:
```
@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 wont print anything because the default logging severity level for the default program logger is `WARNING`

[16]:
```
with log_level(logging.DEBUG, 'my-log') as logger:
    logger.debug(f'This is a message for {logger.name}!')
    logger.debug('This will not print')
```

```
DEBUG:my-log:This is a message for my-log!
DEBUG:my-log:This will not print
```

- after the `with` statement exits, calling debug logging methods on `Logger` named 'my-log' will not print anything because the default severity level has not been restored
- error log messages will always print

[17]:
```
logger = logging.getLogger('my-log')
logger.debug('Debug will not print')
logger.error('Error will print')
```

```
ERROR:my-log:Error will print
```

- later we can change the name of the logger we want to use by simply updating the `with` statement
- this will point the `Logger` thats the `as` target in the `with` statement to a different instance, but we wont have to update any of my other code to match

[18]:
```
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')
```

```
DEBUG:other-log:This is a message for other-log!
```

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

0.3.1 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 yeilded by context manager is supplied to the `as` part of the `with` statement
- its useful for letting your code directly access the cause of a special context

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

- `datetime` built-in module works great with some hlep from the community package named `pytz`
- we dont use `time` for its platform-dependent nature

- below is the code to convert the present time in `UTC` to my computer's local time

[22]:
```python
from datetime import datetime, timezone

now = datetime(2019, 3, 16, 22, 14, 25)
now_utc = now.replace(tzinfo=timezone.utc)
now_local = now_utc.astimezone()
print(now_local)
```

```
2019-03-16 18:14:25-04:00
```

- the `datetime` module can also easily convert a local time back to a UNIX timestamp in UTC

[33]:
```python
import time

time_str = '2019-03-16 15:14:35'
time_format = '%Y-%m-%d %H:%M:%S'
now = datetime.strptime(time_str, time_format)
time_tuple = now.timetuple()
utc_now = time.mktime(time_tuple)
print(utc_now)
```

```
1552763675.0
```

- `datetime` only provides time zone operations with its `tzinfo` class and related methods
- the Python default installation is missing time zone definitions besides UTC
- to avoid having to use the `tzinfo` class we can use the `pytz` module

- to use `pytz` effectively, you should always convert local time to `UTC` first
- perform any `datetime` operations you need on the `UTC` values
- then convert to local times as a final step

- below we convert a NYC flight arrival time to a UTC `datetime`

- although some of these calls seem redundant, all of them are necessary when using `pytz`

[36]:
```
import pytz

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

arrival_nyx = '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 we have a `UTC` datetime, we can convert it to `San Franciso` local time

[38]:
```
pacific = pytz.timezone('US/Pacific')
sf_dt = pacific.normalize(utc_dt.astimezone(pacific))
print(sf_dt)
```

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

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

- `pickle` module can serialize python objects into a stream of bytes and deserialize bytes back into objects
- say we want to use a Python object to represent the state of a players progress in a game

[1]:
```
class GameState:
    def __init__(self):
        self.level = 0
        self.lives = 4
```

- the program modifies this object as the game runs

[2]:
```
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 this game to a file so it can be resumed at a later time
- we use `dump` function to write the `GameState` object to a file

[5]:
```python
import pickle

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

- later we can call the load function with the file and get back the `GameState` object as if it had never been serialized

[6]:
```python
with open(state_path, 'rb') as f:
    state_after = pickle.load(f)

print(state_after.__dict__)
```

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

- the problem with the approach is what happend as the game features expand over time
- imagine you also wanted to track players point

[7]:
```python
class GameState:
    def __init__(self):
        self.level = 0
        self.lives = 4
        self.point = 0 # New field
```

- Serializing the new version of the `GameState` class using `pickle` will work exactly as before
- we simulate the round-trip through a file by serializing to a string with `dumps` and back to an objec twith `loads`

[8]:
```python
state = GameState()
serialized = pickle.dumps(state)
state_after = pickle.loads(serialized)
print(state_after.__dict__)
```

```
{'level': 0, 'lives': 4, 'point': 0}
```

- note that the older saved `GameState` object is not returned
- we can unpickle an old game file by a program with the new definition of the `GameState` class

[9]:
```python
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
- the 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 way

- fixing these problems is straightforward using the `copyreg` built-in module
- the `copyreg` module lets you register the functions resposibility for serializing and deserializing Python objects

0.5.1 Default Attribute Values

- in the simplest case, you can use a constructor with default arguments to ensure that `GameState` objects will always have all attributes after unpickling

[18]:
```
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, we define a helper function that takes a `GameState` object and turnes 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 the unpickling function

[19]:
```
def pickle_game_state(game_state):
    kwargs = game_state.__dict__
    return unpickle_game_state, (kwargs, )
```

- now we 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
- its a tiny wrapper around the constructor

[24]:
```
def unpickle_game_state(kwargs):
    return GameState(**kwargs)
```

- we register these functions with the `copyreg` built-in module

[25]:
```
import copyreg

copyreg.pickle(GameState, pickle_game_state)
```

- after registration, serializing and deserializing works as before

[26]:
```
state = GameState()
state.points += 1000
serialized = pickle.dumps(state)
state_after = pickle.loads(serialized)
print(state_after.__dict__)
```

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

- we can now change the definition of `GameState` again and give players a count of magic spells to use

[40]:
```python
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
```

- 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` modules default behavior of saving and restoring onl the attributes that belong to an object

[41]:
```python
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}
```

0.5.2 Versioning Classes

- sometimes you need to make backward-incompatible changes to your Python objects by removing fields
- doing this prevents the default argument approach above from working
- say we remove the number of lives from the `GameState`

[42]:
```python
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

[43]:
```python
# the following causes TypeError
# pickle.loads(serialized)
```

- we can fix this by adding new version parameters to the function supplied to `copyreg`
- now serialized data will have a version of 2 specified when pickling a new `GameState` object

[44]:
```python
def pickle_game_state(game_state):
    kwargs = game_state.__dict__
    kwargs['version'] = 2
    return unpickle_game_state, (kwargs, 2)
```

- old versions of the data will noe have a `version` argument present
- this means we can manipulate the arguments passed to the `GameState` constructor accordingly

[45]:
```python
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

[46]:
```python
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}
```

- anytime we need to adapt old versions of the same class, we can go the `unpickle_game_state` function and change the `version`

0.5.3 Stable Import Paths

- other isssues with `pickle` could be breakage from renaming a class
- often over the lifecycle of a program, you'll refactor your code by renaming classes and moving them to other modules
- doing so breakes the `pickle` module unless your careful
- below we rename the `GameState` class and remove the old class from the program entirely

[47]:
```python
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 fails because the class cant be found

[48]:
```python
# lol we should be getting an error here?
pickle.loads(serialized)
```

[48]:
```
<__main__.GameState at 0x2004a652310>
```

- the cause of the exception is that the import path of the serialized objects class is encoded in the pickled data

[49]:
```python
print(serialized)
```


```
b'\x80\x04\x95L\x00\x00\x00\x00\x00\x00\x00\x8c\x08__main__\x94\x8c\x13unpickle_
game_state\x94\x93\x94}\x94(\x8c\x05level\x94K\x00\x8c\x05lives\x94K\x04\x8c\x06
points\x94M\xe8\x03u\x85\x94R\x94.'
```


- the solution is to use `copyreg` again
- we can specify a stable identifier for the function to use for unpickling an object

- this allows us to transition pickled data to a different classes with different names when its deserialized

[50]:
```
copyreg.pickle(BetterGameState, pickle_game_state)
```

- after we use `copyreg` you can see that the import path to `unpickled_game_state` is encoded int hat serialized data instead of `BetterGameState`

[51]:
```
state = BetterGameState()
serialized = pickle.dumps(state)
print(serialized)
```


```
b'\x80\x04\x95Y\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\x0
5magic\x94K\x05\x8c\x07version\x94K\x02uK\x02\x86\x94R\x94.'
```


- remember you cant change the path of the module in which the `unpickle_game_state` function is present

0.6 Item 69: Use `decimal` When Precision is Paramount

- if you need a precise number with a eplision of `0.0001`, you could use the `round` function but due to floating point error, rounding to the nearest whole cent could reduce the final cost
- 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

[52]:
```
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 is by passing `str` containing the number to the `Decimal` constructor
- this ensures that there is no loss of precision
- the second way is by directly passing a `float` or an `int` instance to the constructor
- perfer `str` over exact value

[53]:
```
print(Decimal('1.45'))
print(Decimal(1.45))
```

```
1.45
1.4499999999999999555910790149937383830547332763671875
```

- lets suppose we ant to support short phone calls between places that are cheap
- we can compute the charge for a phone call that was 5 seconds long with a rate of `0.05/min`

[54]:
```
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 we try to round ti to the nearest whole cent

[57]:
```
print(round(small_cost, 2))
```

```
0.00
```

- the `Decimal` class has a built-in function for rounding to exactly the decimal palce needed with the desired rounding behavior

[58]:
```
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` values

[60]:
```
rounded = small_cost.quantize(Decimal('0.01'),
                              rounding=ROUND_UP)

print(f'Rounded {small_cost} to {rounded}')
```

```
Rounded 0.004166666666666666666666666667 to 0.01
```

- for representing rational numbers with no limit to precision, consider using the `Fraction` class from the `fractions` built-in module

0.7 Item 70: Profile Before Optimizing

- dynamic nature of Python causes surprising behaviors in its runtime performance
- operations you might assume would be slow are fast
  - string manipulation, generators, etc.
- operations that you would assume would be fast are actually slow
  - attribute accesses, fucntion calls, etc.
- Python provides a built-in `profiler` for determining which parts of a program are responsible for its execution time
- the `cProfile` is better than the `profile` because of its minimal impact on the performance of your program while its being profiled

  be sure that what you're measuring is the code itself and not external systems. Beware of functions that access the network or resources on disk. These may appear to have large impact on your programs exectuion time because of the slowness of the underlying systems. If your program uses a cache to make the latency of slow resources like these, you should ensure that its properly warmed up before you start profiling

- say I want to determine why an algorithm is low
- we can define a function that sorts a `list` of data using an insertion sort

[61]:
```
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 peice of data

[63]:
```
def insert_value(array, value):
    for i, existing in enumerate(array):
        if existing > value:
            array.insert(i, value)
            return
    array.append(value)
```

- to profile `insertion_sort` and `insert_value`, we create a data set of random number and define a `test` function to pass to the profiler

[64]:
```
from random import randint
max_size = 10**4
data = [randint(0, max_size) for _ in range(max_size)]
test = lambda: insertion_sort(data)
```

- here we instantiate a `Profile` object from the `cProfile` module and run the test function through it using the `runcall` method

[67]:
```
from cProfile import Profile

profiler = Profile()
profiler.runcall(test)
```

[67]:
```
[0,
 0,
 3,
 4,
 5,
 5,
 5,
 6,
 8,
 11,
 12,
```

```
13,
14,
14,
14,
14,
16,
17,
17,
18,
18,
19,
19,
20,
20,
22,
22,
23,
23,
23,
25,
28,
28,
29,
29,
29,
30,
30,
35,
36,
37,
39,
41,
42,
43,
43,
43,
44,
48,
48,
50,
50,
52,
53,
54,
55,
55,
56,
```

```
62,
63,
65,
65,
65,
65,
68,
71,
71,
71,
73,
73,
75,
76,
77,
77,
78,
78,
78,
78,
78,
81,
82,
85,
86,
88,
88,
89,
90,
91,
95,
95,
96,
96,
97,
98,
100,
101,
102,
102,
103,
103,
104,
104,
104,
105,
105,
```

```
107,
109,
110,
110,
111,
111,
111,
112,
114,
115,
115,
115,
117,
118,
120,
121,
122,
123,
126,
126,
126,
127,
128,
129,
129,
129,
129,
132,
135,
135,
136,
136,
136,
136,
137,
137,
137,
137,
138,
139,
139,
141,
143,
147,
147,
147,
149,
```

```
149,
149,
149,
152,
154,
155,
157,
157,
157,
158,
158,
158,
159,
161,
161,
162,
162,
163,
166,
167,
170,
170,
171,
172,
172,
172,
177,
179,
179,
179,
180,
180,
180,
180,
183,
186,
188,
191,
191,
193,
194,
196,
197,
197,
199,
199,
200,
```

```
200,
200,
201,
203,
203,
205,
205,
206,
206,
207,
207,
208,
209,
210,
214,
214,
217,
219,
219,
219,
222,
225,
225,
226,
228,
228,
228,
228,
229,
230,
230,
230,
230,
233,
235,
235,
236,
238,
240,
240,
241,
241,
242,
242,
243,
243,
243,
```

```
244,
245,
245,
245,
246,
248,
250,
251,
253,
253,
253,
254,
254,
256,
256,
257,
258,
261,
261,
261,
262,
263,
263,
266,
267,
268,
268,
269,
269,
274,
275,
275,
276,
276,
276,
277,
281,
281,
281,
282,
284,
284,
285,
287,
287,
290,
290,
```

291,
293,
293,
294,
295,
296,
296,
299,
300,
302,
302,
307,
307,
307,
308,
310,
311,
311,
312,
312,
314,
314,
315,
317,
317,
317,
319,
319,
320,
320,
321,
322,
322,
326,
326,
327,
328,
329,
330,
330,
331,
331,
331,
331,
331,
333,
333,

```
336,
336,
336,
338,
339,
340,
343,
343,
347,
347,
350,
350,
351,
351,
351,
351,
352,
353,
354,
354,
355,
356,
356,
359,
360,
360,
362,
363,
364,
364,
365,
365,
366,
366,
368,
368,
369,
370,
371,
372,
372,
373,
373,
375,
376,
377,
377,
```

378,
378,
378,
379,
379,
379,
381,
381,
382,
382,
383,
384,
385,
387,
391,
391,
392,
393,
394,
394,
394,
395,
395,
397,
397,
398,
400,
403,
405,
407,
409,
409,
410,
411,
411,
414,
415,
418,
420,
423,
425,
427,
428,
430,
432,
432,
434,

435,
435,
436,
437,
438,
438,
438,
439,
440,
441,
442,
442,
443,
443,
444,
446,
446,
446,
447,
448,
450,
450,
452,
452,
453,
454,
454,
456,
457,
457,
458,
460,
460,
463,
463,
464,
466,
469,
470,
471,
471,
472,
472,
473,
473,
474,
475,

```
478,
479,
480,
481,
482,
483,
483,
483,
483,
484,
484,
485,
485,
487,
488,
489,
493,
495,
495,
498,
499,
503,
503,
505,
505,
506,
506,
506,
507,
507,
508,
508,
509,
510,
513,
513,
515,
516,
517,
518,
518,
518,
519,
520,
521,
521,
523,
```

523,
524,
525,
525,
525,
526,
526,
528,
530,
532,
532,
533,
534,
535,
536,
537,
539,
541,
542,
543,
545,
546,
547,
548,
548,
549,
552,
552,
553,
553,
554,
555,
558,
559,
559,
560,
560,
562,
562,
562,
563,
563,
563,
563,
564,
566,
567,

567,
568,
569,
569,
574,
575,
576,
579,
579,
580,
581,
588,
588,
589,
590,
592,
594,
594,
595,
598,
598,
601,
602,
602,
604,
605,
605,
609,
609,
609,
611,
612,
613,
613,
614,
614,
616,
617,
620,
621,
621,
622,
622,
622,
623,
624,
624,

625,
627,
628,
628,
629,
629,
630,
631,
631,
632,
634,
635,
635,
637,
639,
640,
640,
642,
642,
642,
643,
643,
644,
644,
645,
645,
646,
647,
648,
648,
649,
650,
650,
651,
651,
653,
654,
654,
654,
655,
656,
657,
658,
660,
661,
661,
662,

663,
664,
664,
664,
665,
665,
666,
667,
668,
668,
669,
669,
670,
670,
671,
672,
676,
676,
677,
677,
678,
679,
679,
680,
683,
686,
686,
689,
689,
689,
690,
692,
692,
692,
693,
694,
694,
695,
697,
698,
698,
699,
700,
701,
701,
704,
704,

704,
704,
705,
708,
709,
710,
710,
712,
713,
714,
714,
717,
717,
718,
722,
723,
723,
725,
725,
727,
727,
727,
728,
729,
729,
729,
730,
730,
731,
732,
735,
735,
735,
738,
738,
738,
739,
740,
740,
740,
742,
743,
745,
746,
750,
751,
753,

753,
754,
755,
758,
758,
760,
762,
763,
764,
765,
766,
766,
768,
768,
769,
770,
772,
772,
773,
775,
777,
777,
778,
778,
779,
779,
781,
782,
783,
784,
785,
787,
787,
788,
789,
790,
791,
792,
794,
794,
796,
797,
797,
799,
800,
801,
801,

801,
802,
802,
806,
807,
807,
808,
810,
811,
813,
816,
817,
817,
817,
819,
820,
821,
821,
822,
822,
823,
823,
823,
824,
824,
824,
824,
824,
825,
825,
825,
826,
828,
828,
830,
830,
831,
832,
833,
834,
835,
835,
836,
841,
842,
846,
846,

846,
847,
848,
849,
849,
849,
850,
851,
851,
851,
852,
853,
854,
855,
856,
856,
857,
858,
859,
859,
860,
861,
862,
863,
864,
865,
868,
869,
869,
869,
872,
872,
873,
873,
874,
876,
877,
878,
879,
880,
880,
880,
880,
881,
881,
882,
883,

884,
884,
884,
885,
885,
885,
886,
887,
887,
888,
891,
893,
897,
899,
900,
901,
902,
904,
905,
905,
906,
906,
908,
908,
909,
911,
915,
916,
917,
918,
918,
919,
920,
922,
926,
926,
928,
929,
931,
931,
931,
935,
935,
936,
937,
938,
939,

939,
943,
943,
944,
944,
944,
945,
947,
948,
949,
950,
951,
951,
952,
954,
955,
957,
958,
958,
958,
959,
960,
961,
963,
964,
965,
966,
966,
966,
970,
970,
970,
971,
971,
973,
975,
976,
977,
978,
978,
979,
979,
980,
981,
981,
981,
982,

```
 984,
 987,
 …]
```

- when the test function has finished running, we can extract statistics about its performance by using the `pstat` module and its `Stats` class
- various methods on a `Stats` object adjust how to select and sort the `profiling` information to show only the things I case about

[68]:
```python
from pstats import Stats

stats = Stats(profiler)
stats.strip_dirs()
stats.sort_stats('cumulative')
stats.print_stats()
```

```
        20003 function calls in 1.845 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    1.845    1.845 <ipython-
input-64-2fd13660a10c>:4(<lambda>)
        1    0.005    0.005    1.845    1.845 <ipython-
input-61-cbe174de868f>:1(insertion_sort)
    10000    1.820    0.000    1.840    0.000 <ipython-
input-63-c4fc1ad100fb>:1(insert_value)
     9990    0.019    0.000    0.019    0.000 {method 'insert' of 'list'
objects}
       10    0.000    0.000    0.000    0.000 {method 'append' of 'list'
objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
```

[68]:
```
<pstats.Stats at 0x2004a60eee0>
```

- `ncalls`: the number of calls to the function during the profiling
- `tottime`: the number of seconds spent executing the function exlcuding time spent executing other function calls
- `tottime percall`: the average number of seconds spent executing the function excluding time spent executing other functions it calls; this is `tottime` divided by `ncalls`
- `cumtime`: the cumulative number of seconds spent executing the function, including time spent in all other functions it calls
- `cumtime percall`the average number of seconds spent in the function call each time its called, including time spent in all other functions each time it is called; this is `cumtime` divided by `ncalls`

- you can also call `stats.print_callers()` to show what is being called

0.8 Item 71: Perfer deque for Producer_consumer Queues

- imagine we have a program that processing incomming emails for long-term archival, and its using a `list` for producer-consuming queue
- we define a class represent the message

[1]:
```python
class Email:
    def __init__(self, sender, receiver, message):
        self.sender = sender
        self.receiver = receiver
        self.message = message
```

- we define a placeholder function for receiving a single email, presumably from a socket, the file system or some other type of `I/O` system
- the implementation of this function does not matte, whats important is the interface: it either returns an `Email` instance or raise a `NoEmailError` exception

[3]:
```python
class NoEmailError(Exception):
    pass

def try_receive_email():
    # Returns an Email instance or raises NoEmailError
    pass
```

- the producing function receives emails and enqueues them to be consumed at a later time
- the function uses the `append` method on the `list` to add new messages to the end of the queue so they are processed after all messages that were previously received

[4]:
```python
def produce_emails(queue):
    while True:
        try:
            email = try_receive_email()
        except NoEmailError:
            return
        else:
            queue.append(email) # Producer
```

- the consuming function does something useful with the emails
- the function calls `pop(0)` on the queue, which removes the very first time from the `list` and returns it to the caller
- this perserves the order in which emails were received

[6]:
```python
def consume_one_email(queue):
    if not queue:
        return
    email = queue.pop(0) # Consumer
```

```python
    # Index the message for long-term archival
```

- finally we need a looping function that conenctgs the pieces together
- this function alternates between producing and consuming unitl the `keep_running` functions returns `False`

[7]:
```python
def loop(queue, keep_running):
    while keep_running():
        produce_emails(queue)
        consume_one_email(queue)

def my_end_func():
    pass

loop([], my_end_func)
```

- the reson we do not process each `Email` message in `produce_emails` as it's returned by `try_receive_email` is because of the trade-off between latency and throughput
- when using producer-consumer queues, you want to minimize the latency of accpeting new items so they can be collected as fast as possible
- the consumer can then process through the backlog og items at a consistent pace- one item per loop
- this provides a stable performance profile and consistent throughput at the cost of end-to-end latency

- using a `list` for a producer-consumer queue like this works fine up to a point, but as the `cardinality`- the number of items in the list increases, the `list` type's performance can degrade superlineraly
- if we `micro-benchmark` using the `timeit` module we can analyze the performance
- we define a benchmark for the performance of adding new items to the queue using the `append` method of `list` (matching the producer function's usage)

[15]:
```python
import timeit

def print_results(count, tests):
    avg_iteration = sum(tests) / len(tests)
    print(f'Count {count:>5,} takes {avg_iteration:.6f}s')
    return count, avg_iteration

def list_append_benchmark(count):
    def run(queue):
        for i in range(count):
            queue.append(i)

    tests = timeit.repeat(
        setup='queue = []',
        stmt='run(queue)',
        globals=locals(),
        repeat=1000,
```

```python
        number=1)
    return print_results(count, tests)
```

- running this benchmark function with different levels of cardinality lets us compare its performance in relationship to data size

[16]:
```python
def print_delta(before, after):
    before_count, before_time = before
    after_count, after_time = after
    growth = 1 + (after_count - before_count) / before_count
    slowdown = 1 + (after_time - before_time) / before_time
    print(f'{growth:>4.1f}x data size, {slowdown:>4.1f}x time')

baseline = list_append_benchmark(500)
for count in (1_000, 2_000, 3_000, 4_000, 5_000):
    comparison = list_append_benchmark(count)
    print_delta(baseline, comparison)
```

```
Count   500 takes 0.000063s
Count 1,000 takes 0.000108s
 2.0x data size,  1.7x time
Count 2,000 takes 0.000167s
 4.0x data size,  2.6x time
Count 3,000 takes 0.000229s
 6.0x data size,  3.6x time
Count 4,000 takes 0.000320s
 8.0x data size,  5.1x time
Count 5,000 takes 0.000402s
10.0x data size,  6.4x time
```

- this shows that the `append` method takes roughly constant time for the `list` type and the total time for enqueueing scales linearly as the data size increases
- there is overhead for the `list` type to increase its capacity under the covers as new items are added, but its reaonably low and is amortized across repeated calls to `append`

- below we define a similar benchmark for the `pop(0)` call that removes items from the beginning of the queue (matching the consumer function's usage)

[21]:
```python
def list_pop_benchmark(count):
    def prepare():
        return list(range(count))

    def run(queue):
        while queue:
            queue.pop(0)

    tests = timeit.repeat(
        setup='queue = prepare()',
        stmt='run(queue)',
```

```
        globals=locals(),
        repeat=1000,
        number=1)

    return print_results(count, tests)
```

- we can similarly run this benchmark for queues of different sizes to see how performance is affected by cardinality

[22]:
```
baseline = list_pop_benchmark(500)
for count in (1_000, 2_000, 3_000, 4_000, 5_000):
    comparison = list_pop_benchmark(count)
    print_delta(baseline, comparison)
```

```
Count   500 takes 0.000112s
Count 1,000 takes 0.000233s
 2.0x data size,  2.1x time
Count 2,000 takes 0.000734s
 4.0x data size,  6.5x time
Count 3,000 takes 0.001450s
 6.0x data size, 12.9x time
Count 4,000 takes 0.002731s
 8.0x data size, 24.3x time
Count 5,000 takes 0.004136s
10.0x data size, 36.8x time
```

- this shows that the total time for dequeuing items from a `list` with `pop(0)` scales quadratically as the lenght of the queue increases
- the causes is that `pop(0)` needs to move every item in the `list` back an index, effectively ressagning the entire list's contents
- we need to call `pop(0)` for every item in the `list` and thus I end up doing roughly `len(queue) * len(queue)` operations to consume the queue; this does not scale

- python provides the `deque` class from the collections module
- `deque` is a double-ended queue implementation
- it provides constant time operations for insetrting or removing items from it beginning or end
- this makes it ideal for `FIFO` queues

- to use the `deque` class, we call to `append` in `produce_emails` can stay the same as it was when using a `list` for queue
- the `list.pop` method call in `consume_one_email` must change to call the `deque.popleft` method with no arguments instead of a `list`
- we redifine the one function affected to use the new method and run `loop` again

[26]:
```
import collections

def consume_one_email(queue):
    if not queue:
        return
    email = queue.popleft() # Consumer
```

```python
    # Process the email message

def my_end_func():
    pass

loop(collections.deque(), my_end_func)
```

- we can run another version of the benchmark to verify that `append` perfromance has stayed roughly the same (module a constant factor)

[29]:
```python
def deque_append_benchmark(count):
    def prepare():
        return collections.deque()

    def run(queue):
        for i in range(count):
            queue.append(i)

    tests = timeit.repeat(
        setup='queue = prepare()',
        stmt='run(queue)',
        globals=locals(),
        repeat=1000,
        number=1)

    return print_results(count, tests)

baseline = deque_append_benchmark(500)
for count in (1_000, 2_000, 3_000, 4_000, 5_000):
    comparison = deque_append_benchmark(count)
    print_delta(baseline, comparison)
```

```
Count   500 takes 0.000052s
Count 1,000 takes 0.000102s
 2.0x data size,  2.0x time
Count 2,000 takes 0.000155s
 4.0x data size,  3.0x time
Count 3,000 takes 0.000254s
 6.0x data size,  4.9x time
Count 4,000 takes 0.000331s
 8.0x data size,  6.3x time
Count 5,000 takes 0.000383s
10.0x data size,  7.3x time
```

- the `popleft` usage scales linearly instead of displaying the super-linear behavior of `pop(0)`

0.9 Item 72: Consider Searching Sorted Sequences with `bisect`

- searching for a specific value in a `list` takes linear time proportional to the list's lenght when you call the `index` method

[30]:
```
data = list(range(10**5))
index = data.index(91234)
assert index == 91234
```

- if you do not know what you are searching for in alist, then you may want to search for the closest index that is equal or exceeds our goal value

[31]:
```
def find_closest(sequence, goal):
    for index, value in enumerate(sequence):
        if goal < value:
            return index
    raise ValueError(f'{goal} is out of bounds')

index = find_closest(data, 91234.56)
assert index == 91235
```

- pythons built-in `bisect` module provides better ways to accomplush these types of searches though ordered `lists`
- you can use the `bisect_left` function to do an eficient binary search through any sequence of sorted items
- the index it returns will either be where the item is already present in the `list` or where you'd want to insert the item in the `list` to keep it in sorted order

[33]:
```
from bisect import bisect_left

index = bisect_left(data, 91234) # Exact match
assert index == 91234

index = bisect_left(data, 91234.56) # Closest match
assert index == 91235
```

- the complexity of the binary search algorithm used by the `bisect` module is logarithmic
- this means searching in a `list` of lenght 1 million takes roughly the same anount of time with `bisect` as linearly searching a list of lenght 20 using `list.index` method

- we can verify the speed improvement for `bisect` by using `timeit` module to run a micro-benchmark

[35]:
```
import random
import timeit

size = 10**5
iterations = 1000

data = list(range(size))
```

```python
to_lookup = [random.randint(0, size)
             for _ in range(iterations)]

def run_linear(data, to_lookup):
    for index in to_lookup:
        data.index(index)

def run_bisect(data, to_lookup):
    for index in to_lookup:
        bisect_left(data, index)

baseline = timeit.timeit(
    stmt='run_linear(data, to_lookup)',
    globals=globals(),
    number=10)
print(f'Linear search takes {baseline:.6f}s')

comparison = timeit.timeit(
    stmt='run_bisect(data, to_lookup)',
    globals=globals(),
    number=10)
print(f'Bisect search takes {comparison:.6f}s')

slowdown = 1 + ((baseline - comparison) / comparison)
print(f'{slowdown:.1f}x time')
```

```
Linear search takes 8.595663s
Bisect search takes 0.006658s
1291.0x time
```

- the best part about `bisect` is that its not limited to the `list` type
- you can use it with any Python object that acts like a sequence

0.10 Item 73: Know How to Use `heapq` for Priority Queues

- sometimes you need a program to process items in order of relative importance instead
- to accomplush this, a `priority queue` is the right tool for the job
- lets assume I am writting a program to manage books borrowed form a library
  - there are people constantly borrowing new books
  - there are people returning their borrowed book on time
  - there are people who need to be reminded to return their overdue books

[47]:
```python
class Book:
    def __init__(self, title, due_date):
        self.title = title
        self.due_date = due_date
```

- I need a system that will send reminder messages when each book passes its due date

- we cant use a FIFO queue for this because the amount of time each book is allowed to be borrowed varies based on its recency, popularity and other factors
- for example, a book that is borrowed today may be due back later than a book thats borrowed tomorrow
- we can achieve this behavior by using a standard `list` and sorting it by `due_date` each time a new `Book` is added

[48]:
```python
def add_book(queue, book):
    queue.append(book)
    queue.sort(key=lambda x: x.due_date, reverse=True)

queue = []
add_book(queue, Book('Don Quixote', '2019-06-07'))
add_book(queue, Book('Frankenstein', '2019-06-05'))
add_book(queue, Book('Les Misérables', '2019-06-08'))
add_book(queue, Book('War and Peace', '2019-06-03'))
```

- if we assume that the queue of borrowed books is always in sorted order, than all I need to do to check for overdue books is to inspect the final element in the `list`
- below we define a function to return the next overdue book, if any and remove it from the queue

[49]:
```python
class NoOverdueBooks(Exception):
    pass

def next_overdue_book(queue, now):
    if queue:
        book = queue[-1]
        if book.due_date < now:
            queue.pop()
            return book
    raise NoOverdueBooks

now = '2019-06-10'
found = next_overdue_book(queue, now)

(found.title)

found = next_overdue_book(queue, now)
print(found.title)
```

```
Frankenstein
```

- if a book is returned before the due date, we can remove the scheduled reminder message by removing the Book from the `list`

[50]:
```python
def return_book(queue, book):
    queue.remove(book)
```

```python
queue = []
book = Book('Treasure Island', '2019-06-04')

add_book(queue, book)
print('Before return', [x.title for x in queue])

return_book(queue, book)
print('After return ', [x.title for x in queue])
```

```
Before return ['Treasure Island']
After return  []
```

- we can confirm that when all books are returned, the `return_book` function will raise the right exception

[52]:
```python
try:
    next_overdue_book(queue, now)
except NoOverdueBooks:
    pass # Excepted
else:
    assert False # Doesn't happen
```

- the problem is that the computational complexity of this solution isn't ideal
- although checking for and removing an overdue book has a constant cost, every time I add a book, I pay the cost of sorting the whole `list` again
- if I have `len(queue)` books to add, and the cost of sorthing them is roughly `len(queue) * math.log(len(queue))`, the time it takes to add books will grow superlinearly `len(queu) * len(queue) * math.log(len(queue))`

- we define a micro-benchmark to measure this performance behavior experimentally by using the `timeit` module

[61]:
```python
import random
import timeit

def print_results(counts, tests):
    pass

def print_delta(before, after):
    pass

def list_overdue_benchmark(count):
    def prepare():
        to_add = list(range(count))
        random.shuffle(to_add)
        return [], to_add

    def run(queue, to_add):
        for i in to_add:
```

```
            queue.append(i)
            queue.sort(reverse=True)

    while queue:
        queue.pop()

    tests = timeit.repeat(
        setup='queue, to_add = prepare()',
        stmt=f'run(queue, to_add)',
        globals=locals(),
        repeat=100,
        number=1)

    return print_results(count, tests)
```

- we can veify that the runtime of adding and removing books form the queue scales superlinearly as the number of books being borrowed increases

[62]:
```
baseline = list_overdue_benchmark(500)
for count in (1_000, 1_500, 2_000):
    comparison = list_overdue_benchmark(count)
    print_delta(baseline, comparison)
```

```
Count 500 takes 0.001138s

Count 1,000 takes 0.003317s
 2.0x data size, 2.9x time

Count 1,500 takes 0.007744s
 3.0x data size, 6.8x time

Count 2,000 takes 0.014739s
 4.0x data size, 13.0x time
```

- when a book is returned before the due date, we need to do a linear scal inorder to find the book in the queue and remove it
- removing a books causes all subsequent items in the `list` to be shifted back an index
- this has a high cost that also scales superlinearly
- belowe we defien another micro-benchmark to test performance of returning a book using this function

[66]:
```
def list_return_benchmark(count):
    def prepare():
        queue = list(range(count))
        random.shuffle(queue)

        to_return = list(range(count))
        random.shuffle(to_return)
```

```
        return queue, to_return

    def run(queue, to_return):
        for i in to_return:
            queue.remove(i)

    tests = timeit.repeat(
        setup='queue, to_return = prepare()',
        stmt=f'run(queue, to_return)',
        globals=locals(),
        repeat=100,
        number=1)

    return print_results(count, tests)
```

[67]:
```
baseline = list_return_benchmark(500)
for count in (1_000, 1_500, 2_000):
    comparison = list_return_benchmark(count)
    print_delta(baseline, comparison)
```

```
Count 500 takes 0.000898s
Count 1,000 takes 0.003331s
 2.0x data size, 3.7x time
Count 1,500 takes 0.007674s
 3.0x data size, 8.5x time
```

- using the methods of `list` may work for a tiny library, but it certianly wont scale to the size of a the new york public library
- Pyhon h as a built-in method module called `heap` that solves this problem by implementing priority queue efficiently
- a `heap` is a data structure that allows for a `list` of items to be maintained where the computational complexity of adding a new item or removing the smallest item has logarithmic computational complexity
- in our library example, smallest means the book with the earliest due date
- below we reimplement the `add_book` function using the `heapq` module
- the queue is still a plain `list`
- the `heappush` function replaces the `list.append` call from before
- and we no longer have to call `list.sort` on the queue

[68]:
```
from heapq import heappush

def add_book(queue, book):
    heappush(queue, book)
```

- if we try to use this with the `Book` class as previously defined, we get this somewhat cryptic error:

[69]:
```python
queue = []
add_book(queue, Book('Little Women', '2019-06-05'))
add_book(queue, Book('The Time Machine', '2019-05-30'))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-69-a242f54708a8> in <module>
      1 queue = []
      2 add_book(queue, Book('Little Women', '2019-06-05'))
----> 3 add_book(queue, Book('The Time Machine', '2019-05-30'))

<ipython-input-68-54e4fd61a978> in add_book(queue, book)
      2
      3 def add_book(queue, book):
----> 4     heappush(queue, book)

TypeError: '<' not supported between instances of 'Book' and 'Book'
```

- the `heapq` module requires items in the priority queue to be comparable and have a natural sort order
- you can quickly give the `Book` class this behavior by using the `total_ordering` class decorator from the `functools` built-in module
- we redefine the class with a less-than method that simply compares the `due_date` fields between two `Book` instances

[72]:
```python
import functools

@functools.total_ordering
class Book:
    def __init__(self, title, due_date):
        self.title = title
        self.due_date = due_date

    def __lt__(self, other):
        return self.due_date < other.due_date
```

- now we can add books to the priority queue by using the `heapq.heappush` function without issues

[73]:
```python
queue = []
add_book(queue, Book('Pride and Prejudice', '2019-06-01'))
add_book(queue, Book('The Time Machine', '2019-05-30'))
add_book(queue, Book('Crime and Punishment', '2019-06-06'))
add_book(queue, Book('Wuthering Heights', '2019-06-12'))
```

- alternatively we can create a `list` with all of the books in any order and then use `sort` method of `list` to produce the heap

[74]:
```
queue = [
    Book('Pride and Prejudice', '2019-06-01'),
    Book('The Time Machine', '2019-05-30'),
    Book('Crime and Punishment', '2019-06-06'),
    Book('Wuthering Heights', '2019-06-12'),
]
queue.sort()
```

- or we can use the `heap.heapify` function to create a heap in linear time as opposed to the sort methods `len(queue) * log(len(queue))` complexity

[ ]:
```
from heapq import heapify
queue = [
    Book('Pride and Prejudice', '2019-06-01'),
    Book('The Time Machine', '2019-05-30'),
    Book('Crime and Punishment', '2019-06-06'),
    Book('Wuthering Heights', '2019-06-12'),
]
heapify(queue)
```

- to check for overdue books, we inspect the first element in the `list` instead of the last and then we use the `heapq.heappop` function instead of the `list.pop` function

[76]:
```
from heapq import heappop
def next_overdue_book(queue, now):
    if queue:
        book = queue[0] # Most overdue first
        if book.due_date < now:
            heappop(queue) # Remove the overdue book
            return book

    raise NoOverdueBooks
```

- now we can find and remove overdue books in order untill there are none left for the current time

[77]:
```
now = '2019-06-02'
book = next_overdue_book(queue, now)
print(book.title)

book = next_overdue_book(queue, now)
print(book.title)

try:
    next_overdue_book(queue, now)
except NoOverdueBooks:
    pass # Expected
else:
    assert False # Doesn't happen
```

```
The Time Machine
Pride and Prejudice
```

- we can write another micro-benchmark to test the performance of this implementation that uses the `heapq` module

[78]:
```python
def heap_overdue_benchmark(count):
    def prepare():
        to_add = list(range(count))
        random.shuffle(to_add)
        return [], to_add
    def run(queue, to_add):
        for i in to_add:
            heappush(queue, i)
        while queue:
            heappop(queue)

    tests = timeit.repeat(
        setup='queue, to_add = prepare()',
        stmt=f'run(queue, to_add)',
        globals=locals(),
        repeat=100,
        number=1)

    return print_results(count, tests)
```

- the benchmark verifies that the heap-based priority queue implementation scales much better (roughly `len(queue) * math.log(len(queue))` without superlinearly degrading performance

[79]:
```python
baseline = heap_overdue_benchmark(500)
for count in (1_000, 1_500, 2_000):
    comparison = heap_overdue_benchmark(count)
    print_delta(baseline, comparison)
```

```
Count 500 takes 0.000150s

Count 1,000 takes 0.000325s
 2.0x data size, 2.2x time

Count 1,500 takes 0.000528s
 3.0x data size, 3.5x time

Count 2,000 takes 0.000658s
 4.0x data size, 4.4x time
```

- with the `heapq` implementation the question that remains is: how should we handle `returns` that are on time?
- the solution is to never remove a book from the priority queue untill its due date

- at that time, it will be the first item in the `list`, and we can simply ignore the book if its already been returned
- we implement this behavior by adding a new field to track the book's return status

[81]:
```python
@functools.total_ordering
class Book:
    def __init__(self, title, due_date):
        self.title = title
        self.due_date = due_date
        self.returned = False # New field


    def __lt__(self, other):
        return self.due_date < other.due_date
```

- then we change the `next_overdue_book` function to repeatedly ignore any book thats already been returned

[82]:
```python
def next_overdue_book(queue, now):
    while queue:
        book = queue[0]
        if book.returned:
            heappop(queue)
            continue
        if book.due_date < now:
            heappop(queue)
            return book

        break

    raise NoOverdueBooks
```

- this approach makes the `return_book` function extremely fast because it makes no modifications to the priority queue

[83]:
```python
def return_book(queue, book):
    book.returned = True
```

- the downside to the `heapq` is that the storage overhead may take signifcant memory
- you should plan for the worst-case by doing something like imposing a maximum numner of simultaneously lent books

0.11 Consider `memoryview` and `bytearray` for Zero-Copy Interactions with `bytes`

- its easy to use `I/O` tools the wrong way and reach the conclusion that the language is too slow for even `I/O`-bound workloads
- suppose we build a media server to stream television or movies over a network to users so they can watch without having to download the video data in advance

- one key feature of such a system is the ability for users to move forward or backward in the video playback so they can skip or repeat parts
- in client program, we can implement this by requesting a chunk of data from server corresponding to the new time index selected by the user

[12]:
```
def timecode_to_index(video_id, timecode):
    # Returns the byte offset in the video data
    pass

def request_chunk(video_id, byte_offset, size):
    # Returns the byte of video_id's data from the offset
    pass

video_id = ""
timecode = '01:09:14:28'
byte_offset = timecode_to_index(video_id, timecode)
size = 20 * 1024 * 1024
video_data = request_chunk(video_id, byte_offset, size)
```

- we also need to implement the server-side handler that receives the `request_chunk` request and returns the corresponding 20MB chunk of video data
- for sake of simplicity, we can assume that the command and control parts of the server have already been hooked up
- we will focus on the last steps where the requested chunk is extracted from gigabytes of video data thats cached in memeory and is then sent over the socket back to the client

[13]:
```
def requested_chunk_extraction():
    socket = ... # socket connection to client
    video_data =  ...
    byte_offset = ...
    size = 20 * 1024 * 1024 # Requested chunk size

    chunk = video_data[byte_offset:byte_offset + size]
    socket.send(chunk)
```

- the latency and throughput of this code will come down to two factors: how much time it takes to slice the `20MB` video chunk from `video_data` and how much time the `socket` takes to transmit that data to the client
- if we assume that the socket is infinitely fast, we can run a micro-benchmark by using `timeit` built in module to understand the performance characteristics of slicing `bytes` instance this way to create chunks

[21]:
```
import timeit

def micro_benchmark():
    def run_test():
        chunk = video_data[byte_offset:byte_offset + size]
        # Call socket.send(chunk), but ignoring for benchmark
```

```
    result = timeit.timeit(
        stmt='run_test()',
        globals=globals(),
        number=100) / 100

    print(f'{result:0.9f} seconds')

# >>>
# 0.004925669 seconds
```

- it takes rough 5 miliseconds to extract the `20MB` slice of data to transmit to the client
- this means the overall throughput of my server is limited to a theoretical maximum of `20MB`/`5 miliseconds` = `7.3 GB`/`second`, since that is the fastest we can extract the video data from memory
- our server will also be limited to `1 CPU-seconds` / `5 miliseconds` = `200` clients requesting new chuncks in parallel, which is tiny compared to the tens of thousands of simultaneous connections that tools like the `asyncio` built-in module can support

  the problem is that slicing `bytes` instane causes the underlying data to be copied, which takes CPU time

- the better way to write this code is by using python builtin `memoryview` type
- this exposes `CPython's` high-performance `buffer protocol` to the programs
- the buffer protocol is a `low-level C` API that allows the Python runtime and `C` extensions to access the underlying data buffers that are behind objects like `bytes` instances
- the best part about `memoryview` instance is that slicing them results in another `memoryview` instance without copying the underlying data
- below we create a `memoryview` wrapping a `bytes` instance and inspect a slice of it

[22]:
```
data = b'shave and a haircut, two bits'
view = memoryview(data)
chunk = view[12:19]
print(chunk)
print('Size: ', chunk.nbytes)
print('Data in view: ', chunk.tobytes())
print('Underlying data:', chunk.obj)
```

```
<memory at 0x000001A573B44DC0>
Size:  7
Data in view:  b'haircut'
Underlying data: b'shave and a haircut, two bits'
```

- by enabling `zero-copy` operations, `memoryview` can provide enormous speedups for code that need to quickly process large amounts of memory, such as numerical `C` extensions like `NumPy` and `I/O`-bount programs like the one below

- below we replace the simple `bytes` slicing from the above example with `memoryview` slicing instead and repeat the micro-benchmark

[29]:
```
video_data = b'shave and a haircut, two bits'
video_view = memoryview(video_data)
byte_offset = 12
byte_offset = 12
size = 12

def run_test():
    chunk = video_view[byte_offset:byte_offset + size]
    # Call socket.send(chunk), but ignoring for benchmark

result = timeit.timeit(
    stmt='run_test()',
    globals=globals(),
    number=100) / 100
print(f'{result:0.9f} seconds')
```

```
0.000000289 seconds
```

- the result is 250 nanoseconds
- now the theorethical max throughput of the server is `20MB/250 nanosecods` = `164TB/second`
- for parallel clients, we can theortically support up to `1CPU-second/250 nanoseconds` = `4 million`
- this means that the program is entirely bound by the underlying performance of the socket connection to the client, not by CPU constraints

- imagine that the data must flow in the other direction, where osme clients are sending live video streams to the server in order to broadcast them to other users
- in order to do this, we need to store the latest video data from the user in a cache that other clients can read from
- heres what the implementation of reading `1MB` of new data from the incoming client looks like

[31]:
```
def live_streaming():
    socket = ... # socket connection to the client
    video_cache = ... # Cache of incoming video stream
    byte_offset = ... # Incoming buffer position
    size = 1024 * 1024 # Incoming chunk size
    chunk = socket.recv(size)
    video_view = memoryview(video_cache)
    before = video_view[:byte_offset]
    after = video_view[byte_offset + size:]
    new_cache = b''.join([before, chunk, after])
```

- the `socket.recv` method returns a `byte` instance
- we can splice the new data with the existing cache at the current `byte_offset` by using simple slicing operations and the `bytes.join` method
- to understand the performance of this, we can run another `micro-benchmark`

[34]:
```
def live_stream_microbenchmark():
```

```python
def run_test():
    chunk = socket.recv(size)
    before = video_view[:byte_offset]
    after = video_view[byte_offset + size:]
    new_cache = b''.join([before, chunk, after])

result = timeit.timeit(
    stmt='run_test()',
    globals=globals(),
    number=100) / 100

print(f'{result:0.9f} seconds')
```

- it takes 33 miliseconds to receive `1MB` and update the video cache
- this means my maximum receive through put is `1MB/33 miliseconds` = `31MB /second` and I am limited to `31MB/1MB` = `31 simultaneous` clients streaming in the video data this way and does not scale

- a better way to write this code is to use Pythons built-in `bytearray` type in conjunction with `memoryview`
- one limitation with `bytes` instance is that they are read-only and don't allow for individual indexes to be updates

[35]:
```python
my_bytes = b'hello'
my_bytes[0] = b'\x79'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-35-ddb62563245f> in <module>
      1 my_bytes = b'hello'
----> 2 my_bytes[0] = b'\x79'

TypeError: 'bytes' object does not support item assignment
```

- the `bytearray` type is a mutable version of `bytes` that allows for arbitrary positions to be overwritten
- `bytearray` uses integers for its values instead of `bytes`

[36]:
```python
my_array = bytearray(b'hello')
my_array[0] = 0x79
print(my_array)
```

```
bytearray(b'yello')
```

- a `memoryview` can also be used to wrap a `bytearray`
- when you slice such a `memoryview` the resulting object can be used to assign data to a particular portion of the underlying buffer
- this eliminates the copying costs from above that were required to splice the `bytes` instance back together after the data was received from the client

[37]:
```
my_array = bytearray(b'row, row, row your boat')
my_view = memoryview(my_array)
write_view = my_view[3:13]
write_view[:] = b'-10 bytes-'
print(my_array)
```

```
bytearray(b'row-10 bytes- your boat')
```

- many libraries such as `socket.recv_into` and `RawIOBase.readinto`, using the buffer protocol to receive or read data quickly
- the benefit of these methods is that they avoid allocating memory and creating another copy of the data;
- what's received goes straight into an existing buffer
- Here I use the `socket.recv_into` along with a `memoryview` slice to receive data into an underlying `bytearray` without the need for splicing

[40]:
```
def memoryview_slice():
    video_array = bytearray(video_cache)
    write_view = memoryview(video_array)
    chunk = write_view[byte_offset:byte_offset + size]
    socket.recv_into(chunk)
```

- we can run another bencmark to compare performance of this approach to the earlier example that used `socket.recv`

[42]:
```
def memoryview_slice_benchmark():
    def run_test():
        chunk = write_view[byte_offset:byte_offset + size]
        socket.recv_into(chunk)

    result = timeit.timeit(
        stmt='run_test()',
        globals=globals(),
        number=100) / 100

    print(f'{result:0.9f} seconds')

# 0.000033925 seconds
```

- it took 33 microseconds to receive a `1MB` video transmission
- this means server can support `1MB/33 microseconds` = `31BG/second` of max throughput and `31GB/1MB` = `31,000` parallel streaming clients

0.11.1 Things to Remember

- the `memoryview` built-in type provides a zero-copy interface for reading and writing slices of objects that support Python's high-performance buffer protocol
- the `bytearray` built-in type provides a mutable `bytes-like` type that can be used for `zero-copy` data reads with functions like `socket.recv_from`

- a `memoryview` can wrap a `bytearray`, allowing for received data to be spliced into an arbitrary buffer location without copying costs