

# Context Managers and Other Topics





# try/finally for resource management

- We used to use the finally statement for resource management, such as closing files and releasing locks
- If an exception is raised in the try block the finally block executes and then the exception is raised

```
handle = open(filename)
try:
    data = handle.read()
finally:
    handle.close()
```

```
from threading import Lock
lock = Lock()
```

```
lock.acquire()
try:
    x += 1
finally:
    lock.release()
```

# with statement

- Since Python 2.5 we use the with statement and "context managers"

```
with open(filename) as handle:  
    data = handle.read()
```

```
from threading import Lock  
lock = Lock()
```

```
with lock:  
    x += 1
```

```
with context_manager as resource:  
    resource.use()
```

- Used with database sessions, connections, anything with a defined life or span of effect

More concise and easier to read than try/finally and the indented block visually shows the life of the resource. The resource is cleaned up even if an exception is raised.

# Context Managers and Exceptions

- Context managers are able to handle exceptions
- It's more common for them to re-raise exceptions

```
>>> import pytest
>>> with pytest.raises(TypeError):
...     1 + '2'
...
>>> with pytest.raises(ValueError):
...     [0, 1, 2][5]
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError: list index out of range
```

# Context Managers Return the Resource

- The context manager returns an object to be used in the with statement, typically a resource or context or session

```
>>> import time
>>> from unittest.mock import patch
>>>
>>> with patch('time.sleep') as mock_sleep:
...     mock_sleep.return_value = None
...     time.sleep(100)
...
>>> mock_sleep
<MagicMock name='sleep' id='140122800561248'>
>>> mock_sleep.assert_called_once_with(100)
```

# With Statement and Context Managers

- File handles, locks (etc) are "context managers" and implement the context manager protocol
- There are three phases of use of a resource in a with statement
  1. Preparing (entering the with statement)
  2. Using the resource (inside the with statement)
  3. Cleaning up (exiting the with statement)

There are two protocol methods to correspond to phases 1 and 2 with the `__enter__` and `__exit__` protocol methods.

- `__enter__` returns the resource
- `__exit__` is able to handle an exception or re-raise it

# A Simple Context Manager

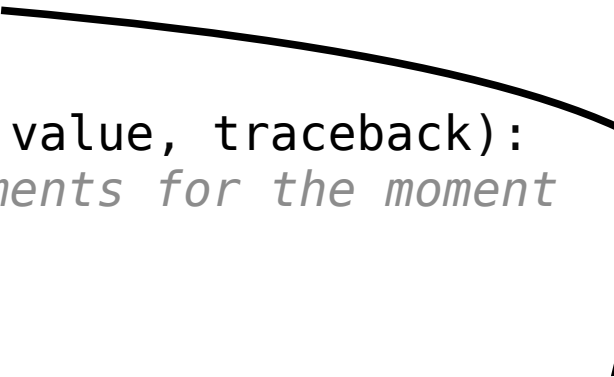
```
class File:
    def __init__(self, filename, method='rt'):
        self.handle = open(filename, method)

    def __enter__(self):
        return self.handle

    def __exit__(self, typ, value, traceback):
        # Ignore these arguments for the moment
        self.handle.close()

with File('contextmanagers.pdf', 'wt') as handle:
    assert not handle.closed
    data = handle.read()

assert handle.closed
```



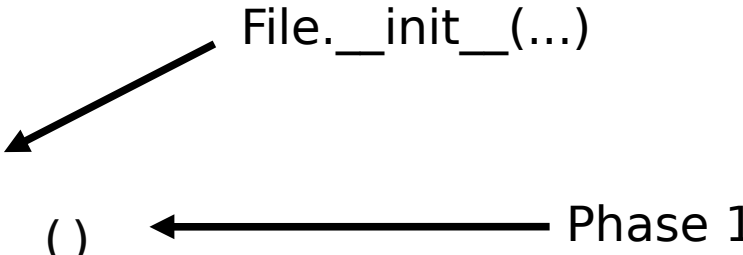

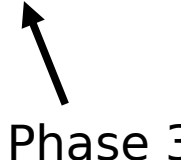


# The Context Manager Protocol

- This code:

```
with File(filename) as handle:  
    data = handle.read()
```

- Is equivalent to:

```
import sys  
ctx_mgr = File(filename)  
handle = ctx_mgr.__enter__()   
  
try:  
    data = handle.read()   
except Exception:  
    typ, value, traceback = sys.exc_info()  
    suppress = ctx_mgr.__exit__(typ, value, traceback)   
    if typ is not None and not suppress:  
        raise
```

# Handling Exceptions

```
class File:
    def __init__(self, filename, method='rt'):
        self.handle = open(filename, method)
    def __enter__(self):
        return self.handle
    def __exit__(self, typ, value, traceback):
        if typ is not None:
            print(f"Exception {value!r} has been handled")
        self.handle.close()
        return True

>>> with File('contextmanagers.pdf', 'wt') as handle:
...     handle.missing_method()
...
Exception AttributeError("'_io.TextIOWrapper' object has no
attribute 'missing_method'") has been handled
```

# Async Context Managers

- When you need to await on entering or leaving you create an async context manager and provide `__aenter__` and `__aexit__` methods instead

```
class Connection:
    def __init__(self, host, port):
        self.host = host
        self.port = port

    async def __aenter__():
        #setting up a connection
        self.conn = await get_conn(self.host, self.port)
        return conn

    async def __aexit__(self, exc_type, exc, tb):
        #closing the connection
        await self.conn.close()

async with Connection('localhost', 500) as conn:
    data = await conn.recv()
```

# A Simpler Way: contextlib

- Instead of writing context managers use `contextlib.contextmanager (*)` and write a generator
- <https://docs.python.org/3/library/contextlib.html>

```
from contextlib import contextmanager
```

```
@contextmanager
def managed_file(filename, suppress=False):
    resource = open(filename)
    try:
        yield resource
    except Exception:
        if not suppress:
            raise
    finally:
        resource.close()
```

Phase 1

Phase 2

Phase 3

`contextmanager` turns the generator into a context manager with `__enter__` and `__exit__` methods.

(\*) or `asynccontextgenerator` or perhaps `contextdecorator (**)` (\*\*) which I wrote



# Class Decoration via Inheritance

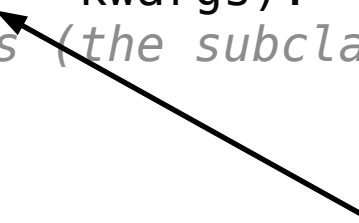
- Base classes can customise subclasses with the `__init_subclass__` method
- Can also be used for class registration, a framework tool

```
class Parent:
    def __init_subclass__(cls, **kwargs):
        # Do something with cls (the subclass)
        ...
```

```
class Child1(Parent):
    pass
```

```
class Child2(Parent, a=1, b=2):
    pass
```

The parent receives the child class as the first argument, `cls`, along with any keyword arguments in the child definition



- Like an implicit/inherited class decorator
- Class decorators and `__init_subclass__` replace most use-cases for metaclasses

# Class Level Cache

- From dataclasses-envloader project
- You build Config classes defining a set of environment variables along with defaults
- `Config.from_env()` builds a config instance from the environment, including reading `.env` files (compatible with pipenv and Docker)
- `from_env` has a cache, on first call, so that subsequent calls are cheap

```
@dataclass
class Config(EnvLoaderMixin):
    API_USERNAME: str
    API_PASSWORD: str
    SMTP_SERVER: str = "smtp.office365.com"
    SMTP_PORT: int = 587

config = Config.from_env()
```

# `__init_subclass__` Example

- A simplified view of the code:

```
class EnvLoaderMixin:
```

```
    def __init_subclass__(cls, prefix="", **_):  
        cls._cached_env = None  
        # Stuff about prefixes omitted
```

```
    @classmethod  
    def from_env(cls, prefix=None, cached=True):  
        if cls._cached_env is None:  
            config = ...  
            cls._cached_env = config  
        return cls._cached_env
```

- When you create a Config class which subclasses EnvLoaderMixin, it is `__init_subclass__` that sets up the cache for that class






# for and tuples

- You can have multiple iteration variables

```
points = [  
    (1, 4), (10, 40), (23, 14), (5, 6), (7, 8)  
]  
  
for x, y in points:  
    # Loops with x = 1, y = 4  
    #           x = 10, y = 40  
    #           x = 23, y = 14  
    #           ...
```



tuples are exanded

- Here each tuple is unpacked into a set of iteration variables

# enumerate() function

- `enumerate(sequence [, start=0])`
- Provides a loop counter

```
names = ['Elwood', 'Jake', 'Curtis']
for i, name in enumerate(names):
    # Loops with i = 0, name = 'Elwood'
    #             i = 1, name = 'Jake'
    #             i = 2, name = 'Curtis'
    ...
```

- Example: keeping track of line number

```
with open(filename) as f:
    for lineno, line in enumerate(f, start=1):
        ...
```

# enumerate() function

- enumerate() is a nice shortcut

```
for i, x in enumerate(s):  
    statements
```

- Compare to:

```
i = 0  
for x in s:  
    statements  
    i += 1
```

- Less typing and enumerate() runs slightly faster

# zip() function

- Makes an iterator that combines sequences

```
columns = ['name', 'shares', 'price']  
values = ['GOOG', 100, 490.1]
```

```
pairs = zip(a, b)  
# ('name', 'GOOG'), ('shares', 100), ('price', 490.1)
```

- To get the result, you must iterate

```
for name, value in pairs:  
    ...
```

- Common use: making dictionaries

```
d = dict(zip(columns, values))
```

# List Comprehensions

- Creates a new list by applying an operation to each element in a sequence

```
>>> a = [1, 2, 3, 4, 5]
>>> b = [2*x for x in a]
>>> b
[2, 4, 6, 8, 10]
```

- Another example:

```
>>> names = ['Elwood', 'Jake']
>>> a = [name.lower() for name in names]
>>> a
['elwood', 'jake']
```

# List Comprehensions

- A list comprehension can also filter

```
>>> a = [1, -5, 4, 2, -2, 10]
>>> b = [2*x for x in a if x > 0]
>>> b
[2, 8, 4, 20]
```

- Another example

```
>>> f = open('stockreport', 'r')
>>> goog = [line for line in f if 'GOOG' in line]
```

# List Comprehensions

- General syntax

*[expression for names in sequence if condition]*

- List comprehensions come from maths

$a = \{ x^2 \mid x \in s, x > 0 \}$       *# Math*

- What it means

```
result = []
for names in sequence:
    if condition:
        result.append(expression)
```

- Can be used anywhere a sequence is expected

```
>>> a = [1, 2, 3, 4]
>>> sum([x*x for x in a])
30
```



# List Comp: Examples

- List comprehensions are hugely useful
- Collecting the values of a specific field

```
stocknames = [s['name'] for s in stocks]
```

- Performing database-like queries

```
a = [s for s in stocks if s['price'] > 100  
                        and s['shares'] > 50 ]
```

- Data reductions over sequences

```
cost = sum([s['shares']*s['price'] for s in stocks])
```

# Generator Expressions

- List comprehensions are "eager", they consume their input and produce a list
- Many of the builtin functions in Python are "lazy", they produce iterable objects instead of executing immediately

```
>>> range(100)
range(0, 100)
>>> zip(['name', 'shares', 'prices'], ['GOOG', 100, 490.10])
<zip object at 0x7f503b5d80c0>
>>> enumerate(nums)
<enumerate object at 0x7f503b6a8840>
```

- Generator expressions are a lazy version of list comprehensions

# Generator Expressions

- Generator expressions produce "one shot" generators
- The syntax is very similar to list comprehensions

```
>>> a = [2, 4, 6, 8, 10]
>>> b = (x**2 for x in a)
>>> b
<generator object <genexpr> at 0x7f503b78f760>
>>> for result in b:
...     print(result)
...
4
16
36
64
100
```

- They don't produce a list, so the whole result set doesn't need to be in memory
- They can't be reused

# Generator Expressions

- General syntax (very similar to list comprehensions)  
(*expression for names in iterable if conditional*)
- They look better than list comprehensions in function calls

```
sum(x*x for x in a)
```

- Can be applied to any iterable and even chained together

```
>>> a = [1,2,3,4]
>>> b = (x*x for x in a)
>>> c = (-x for x in b)
>>> for i in c:
...     print(i, end=' ')
...
-1 -4 -9 -16
```



# Positional and Keyword Only Arguments

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

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

# Ternary Expressions

- Also known as "conditional expressions"
- A concise way of having an expression evaluate to a value based on a condition

```
>>> email_address = None
>>> send_email = True if email_address is not None else
False
>>> send_email
False
>>> email_address = 'michael@python.org'
>>> send_email = True if email_address is not None else
False
>>> send_email
True
```

- Very terse syntax, it maybe clearer to use if/else
- Like list comprehensions it can be helpful to start reading them in the middle (the true condition is on the left, the false is on the right and the if is in the middle)

# The Walrus Operator

- Assignment as an expression: `x := 3`
- New in Python 3.8
- Useful where you need to test a value immediately after setting it
- Can be used to write inscrutable code!
- Old code (regular expressions):

```
>>> match = re.match(r'\w+@(\w+\.\w+)', email_address)
>>> if match is not None:
...     domain = match.groups(1)
... 
```

- With the walrus operator:

```
>>> if match := re.match(r'\w+@(\w+\.\w+)', email_address):
...     domain = match.groups(1)
... 
```