Scientific Software Development with Python

Python recipes

Simon Pfreundschuh
Department of Space, Earth and Environment

CHALMERS
UNIVERSITY OF TECHNOLOGY

Agenda

CHALMERS
UNIVERSITY OF TECHNOLOGY

1. **Overview**
2. Context managers
3. Error handling
4. Serializing objects
5. Log messages
6. Type annotations

Overview

	Conceptual	Technical
Organisational	Project planning & management	Version control, testing, deployment (DevOps)
Implementational	Software design	This Lecture Python programming, scientific computing

Lecture content

- Pythonic approaches to handling common programming tasks
  - Acquiring and releasing of resources
  - Error handling
  - Storing data
  - Logging
  - Type annotations
- Some *real* hacking (exercises)

Agenda

1. Overview

2. **Context managers**

3. Error handling

4. Serializing objects

5. Log messages

6. Type annotations

The need for context managers

The problem

- Some code may require specific *setup* and *tear down* actions.

Example

- Opening and closing of files:
  - File needs to be closed to ensure that all data is written to it.[1]

```
f1 = open("test_file.txt", "w")
f1.write("hi!")

f2 = open("test_file.txt", "r")
content = f2.read()
print(content) # Prints: ''
```

[1] This is called *buffering* and implemented to minimize the number of slow hard disc accesses.

The need for context managers

- To obtain the correct result, the file must be closed by calling the `close` method:

```python
f1 = open("test_file.txt", "w")
f1.write("hi!")
f1.close()

f2 = open("test_file.txt", "r")
content = f2.read()
print(content) # Prints: 'hi!'
```

Problems with this approach

- The file is not closed if an exception is thrown between the opening and the closing of the first file

The need for context manager

- Python, of course, takes care of that for you:

```python
with open("test_file.txt", "w") as f1:
    f1.write("hi!")

f2 = open("test_file.txt", "r")
content = f2.read()
print(content) # Prints: 'hi!'
```

Context manager classes

How does this work

- The `with` statement is only *syntactic sugar* for two special methods:
  - `__enter__`: Is called when the `with` block is entered
  - `__exit__`: Is called when the `with` block is left

Example

```python
class MyContextManager:
    def __init__(self):
        print("1: Context manager created.")

    def __enter__(self):
        print("2: Entering with block.")

    def __exit__(self, exc_type, exc, exc_tb):
        print("4: Leaving with block.")

with MyContextManager():
    print("3: Inside with block.")
```

Context manager classes

Example usage

```python
with MyContextManager():
    print("3: Inside with block.")
```

Output

```
1: Context manager created.
2: Entering with block.
3: Inside with block.
4: Leaving with block.
```

Context manager classes

Example usage

- Note that the `__exit__` method is called even when an exception is raised.

```python
with MyContextManager():
    raise Exception("Uh oh. Something went wrong.)
    print("3: Inside with block.")
```

Output

```
1: Context manager created.
2: Entering with block.
4: Leaving with block.
-----------------------------------------------------------------------
Exception                                 Traceback (most recent ...)
...
Exception: Uh oh. Something went wrong.
```

Context manager classes

Error handling

- The arguments of the `__exit__` method can be used to handle errors occurring in the `with` block:
  - `exc_type`: The type of the thrown exception
  - `exc`: The thrown exception object
  - `exc_tb`: The traceback describing the program state
- To avoid an exception from propagating upwards, the `__exit__` method should return `True`

Context manager classes

Error handling

```python
class NotSoSeriousException(Exception):
    pass

class MyContextManager:
    def __init__(self):
        print("1: Context manager created.")

    def __enter__(self):
        print("2: Entering with block.")

    def __exit__(self, exc_type, exc, exc_tb):
        print("4: Leaving with block.")
        if exc_type == NotSoSeriousException:
            print("5: Something happened but it's not so bad.")
            return True
```

Context manager classes

Error handling

```python
with MyContextManager():
    raise NotSoSeriousException("Uh oh. Something went wrong.")
    print("3: Inside with block.")
```

Output

```
1: Context manager created.
2: Entering with block.
4: Leaving with block.
5: Something happene but it's not so bad.
```

Context manager classes

The `contextlib` module

- Provides abstract base classes for defining context managers.
- Also defines a decorator function to simplify the definition of context managers

Using the `contextmanager` decorator

```
from contextlib import contextmanager

@contextmanager
def my_context_manager():
    print("2: Entering with block.")
    try:
        yield
    finally:
        print("4: Leaving with block.")
```

Context manager classes

Using the `contextmanager` decorator

```python
with my_context_manager():
    print("3: Inside with block.")
```

Output

```
2: Entering with block.
3: Inside with block.
4: Leaving with block.
```

Exercise 1

- Exercise 1 in exercise notebook
- Time: 15 minutes

Agenda

1. Overview

2. Context managers

3. **Error handling**

4. Serializing objects

5. Log messages

6. Type annotations

Exceptions

Raising exception

- Exceptions are special objects used to signal an error occurring in a program
- Exceptions are *raised* using the `raise` keyword:

```
raise Exception("Uh oh, something went wrong.")
```

- When an exception is raised, execution of the current function and any calling functions stops. The exception propagates upwards in the call stack until it is either caught or program execution is aborted.

```
def a_fragile_function():
    print("This will be printed.")
    raise Exception("Uh oh, something went wrong.")
    print("This will not be printed.")
```

Exceptions

Tracebacks

- When an exception propagates all the way up to the interpreter it is printed together with a traceback of the call stack, which helps to identify the problem:

```
~/ssdp/lectures/10/test_module.py in a_fragile_function()
      1 def a_fragile_function():
----> 2     raise Exception("Uh oh, something went wrong.")

Exception: Uh oh, something went wrong.
```

Handling exceptions

Handling exceptions

- Error are handled using special blocks `try`, `except`, `else`, `finally`.

Basic error handling

- If we want to try something but don’t care if it works
- The `try` block designates a region of codes in which an error may occur.
- The `except` keyword is followed by the exception type that we want to catch.

```
try:
    a_fragile_function()
except Exception:
    pass
```

Handling exceptions

The problem with excepting too general exceptions

- The `Exception` class is the base class for all built-in exceptions.
- `except Exception` therefore handles *all possible exceptions*, which is seldomly what we want to achieve.

Example

- In the example below I wouldn't even realize, that there is a spelling error in the function I intended to call:

```python
try:
    a_fagile_function()
except Exception:
    pass
print("This codes executes correctly despite the spelling error.")
```

Handling exceptions

Excepting too general exceptions

`except` blocks that specify a very general exception class (or no exception class at all) are considered bad practice.

A better approach

- When you raise an exception, define a custom exception class:

```
class ExampleError(exception): pass

def a_fragile_function():
    """
    This function throws an error.

    Raises:
        ExampleError: Raised when the funciton is called.
    """
    print("This will be printed.")
    raise ExampleError("Uh oh, something went wrong.")
    print("This will not be printed.")
```

- **Note**: Exceptions raised by a function must be documented.

A better approach

- Calling code can now handle the exceptions that it really intends to handle:

```
try:
    a_fagile_function()
except ExampleError:
    pass
```

- The exception caused by the misspelled function name now propagates upwards as expected:

```
NameError: name 'a_fagile_function' is not defined
```

Handling exceptions

- A `try` block can be followed by multiple `except` blocks:

```python
from test_module import a_fragile_function, ExampleError
try:
    a_fagile_function()
except ExampleError:
    pass
except NameError:
    print("You made a spelling mistake!")
```

- The exception caused by the misspelled function name now propagates up as expected:

Output

```
You made a spelling mistake!
```

Handling exceptions

The `else` and `finally` blocks

- The `except` blocks can be followed by an `else` and a `finally` block:
  - The `else` block:
    - Executed only if no exception was encountered in `try` block.
  - The `finally` block:
    - Executed independent of outcome from `try` block
    - Useful to perform clean up operations (like `__exit__` in a context manager)

Handling exceptions

Example

```python
from test_module import a_fragile_function, ExampleError
try:
    # Acquire resources and try to obtain input. If that
    # doesn't work, continue.
    get_resources()
    input = get_input()
except ExampleError:
    pass
except NameError:
    print("You made a spelling mistake!")
else:
    # If we have obtained erroneous input propagations should
    # propagate upwards.
    check_input()
finally:
    # Always release resources again.
    release_resources()
```

Handling exceptions

What's the `else` block for?

- Code that should execute *before the* `finally` block but for which you don't want to catch error should go in the `else` block
- This is better than adding more statements to the `try` block because it avoids exceptions from being *swallowed*.

Custom exceptions

Define a root exception for your module

- If you define custom exceptions in your package, it is **good practice** to define a root exception.
- The root exception should be the base class for all exception classes defined in your package
- This allows calling code distinguish exception from your code from other exceptions.

```python
class TestModuleException(Exception):
    """Base class for all exceptions from the ``test_module``."""

class ExampleError(TestModuleException):
    """Example error raised from the ``test_modul``."""
```

Agenda

CHALMERS
UNIVERSITY OF TECHNOLOGY

1. Overview

2. Context managers

3. Error handling

4. **Serializing objects**

5. Log messages

6. Type annotations

Serializing objects

The problem

- How do we store custom classes to disk?
- Serialization: Converting a class hierarchy to a 1-dimensional data stream

Naive approach

- Define `load` and `save` methods which store and read objects to and from disk using primitive data types (numbers and strings).

Serializing objects

The pythonic approach: `pickle`

- The `pickle` module allows storing *most* Python classes as binary data.

```python
import random
import pickle

class MyClass:
    def __init__(self, n):
        self.data = list(range(n))

my_object = MyClass(10)
print(my_object.data)        # Prints: [0, ..., 9]

# Note: Need to open file in binary mode ("wb")!
with open("my_class.pckl", "wb") as file:
    pickle.dump(my_object, file)

# Note: Need to open file in binary mode ("rb")!
with open("my_class.pckl", "rb") as file:
    my_loaded_object = pickle.load(file)

print(my_loaded_object.data) # Prints: [0, ..., 9]
```

Pickle

Restrictions

- Functions and classes are pickled by name reference
  - Pickle only stores class data, not the the code defining the class
  - They must be importable from the environment where the unpickling is performed
- Certain types that interact with the computing environment cannot be pickled

Pickle

Example

```python
class MyClass:

    def __init__(self, filename):
        self.file_handle = open(filename, "w")

    def __del__(self):
        if (self.file_handle):
            self.file_handle.close()
            self.file_handle = None

my_object = MyClass("some_file.txt")

with open("my_object.pckl", "wb") as file:
    pickle.dump(my_object, file)
```

Output

```
TypeError: cannot serialize '_io.TextIOWrapper' object
```

Customizing pickling behavior

- To avoid these problems pickling behavior can be customized using the `__setstate__` and `__getstate__` special methods.

```
class MyClass:

    def __init__(self, filename):
        self.filename = filename
        self.file_handle = open(filename, "w")

    def __setstate__(self, state):
        self.file_handle = open(state["filename"])

    def __getstate__(self):
        return {"filename": self.filename}

    def __del__(self):
        if (self.file_handle):
            self.file_handle.close()
            self.file_handle = None
```

Pickle

Warning

Unpickling data is a security risk. Only unpickle data from trusted sources.[2]

[2] We'll see more of this later.

JSON

Serialization using `json`

- Uses JavaScript Object Notation (JSON) format
- Stores data in (human-readable) text files
- Cross-language compatibility
- Works only for lists, dicts and primitive types
- Considered safe

```python
import json

data = [1, 2, "data"]

with open("data.json", "w") as file:
    json.dump(data, file)

with open("data.json", "rt") as file:
    data_loaded = json.load(file)

print(data_loaded)
```

JSON

Serializing custom classes with `json`

- Can define custom *encoder* and *decoder* classes
- Provided to `dump` and `load` methods using the `cls` argument.

```
import json
from json import JSONEncoder, JSONDecoder

class MyClassEncoder(JSONEncoder):
    ...

class MyClassDecoder(JSONDecoder):
    ...

my_object = MyClass("some_file.txt")
with open("my_object.json", "w") as file:
    json.dump(my_object, file, cls=MyClassEncoder)

with open("my_object.json", "r") as file:
    my_object = json.load(file, cls=MyClassDecoder)
```

JSON

The custom encoder

- Should from `JSONEncoder` base class
- Should override `default(obj)` method, which turns an object into a representation of `json` serializable datatypes.
- Should handle objects of targeted class and delegate reset to base class implementation.

```python
class MyClassEncoder(JSONEncoder):
    def default(self, obj):
        if isinstance(obj, MyClass):
            return {"MyClass": obj.filename}
```

The custom decoder

- Should inherit from `JSONDecoder` base class.
- Should provide custom `object_hook` to `__init__` call of base class.
- `object_hook` should check if a loaded `json` object should be turned into an object of the custom class.

```python
class MyClassDecoder(JSONDecoder):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, object_hook=self.object_hook, **kwargs)

    def object_hook(self, obj):
        if "MyClass" in obj:
            return MyClass(obj["MyClass"])
        return obj
```

Notes on `pickle` and `json`

- Both `pickle` and `json` also provide the `dumps` and `loads` methods, which write and read data to and from stream object instead of file handles, respectively.
- Prefer specialized data formats when storing large data (NetCDF, HDF5).

Excercise 2

- Exercise 2 on exercise sheet
- Time: 15 minutes

Agenda

CHALMERS
UNIVERSITY OF TECHNOLOGY

1. Overview

2. Context managers

3. Error handling

4. Serializing objects

5. Log messages

6. Type annotations

Log messages

The problem

- For diagnostic purposes it is often useful to provide messages from different parts of a program

The solution

- The `logging` module provides a standardized solution to handle logging of information

Log messages

Types of messages

`DEBUG` Detailed information for diagnosing problems

`INFO` General information

`WARNING` Something unexpected happened but things still work.

`ERROR` Something unexpected happened and the program was not able to perform a certain function[3].

`CRITICAL` A very serious error

[3] This should only be used when the program can resume execution. Otherwise throw an exception.

Log messages

Example

```python
import logging
logger = logging.getLogger("test_logger")
logger.debug("A debug message.")
logger.info("An info message.")
logger.warning("A warning.")
logger.error("An error.")
logger.critical("A critical error.")
```

Output

- By default, only messages with levels higher or equal than warning are printed.

```
A warning.
An error.
An critical error.
```

Log messages

Controlling the output level

- The logging behavior should be customized using the `basicConfig` function upon program start.

```
import logging
loggin.basicConfig(level=logging.DEBUG)
logger = logging.getLogger("test_logger")
logger.debug("A debug message.")
```

Output

```
DEBUG:test_logger:A debug message.
```

Log messages

Controlling message formatting

- The message format can be customized by providing a custom format string:

```
import logging
format_string = "{name} ( {levelname:10} ) :: {message} "
loggin.basicConfig(level=logging.DEBUG,
                    format=format_string,
                    style="{")
logger = logging.getLogger("test_logger")
logger.debug("A debug message.")
```

Output

```
__main__ ( DEBUG      ) :: A debug message.
```

Log messages

Logging to a file

- This will store log output to `log.txt`.

```python
import logging
loggin.basicConfig(level=logging.DEBUG, filename="log.txt", mode="w")
logger = logging.getLogger("test_logger")
logger.debug("A debug message.")
```

Log messages

CHALMERS
UNIVERSITY OF TECHNOLOGY

Handling output from different modules

- It is useful to separate output from different modules by using different logger objects[4]:

```
import logging
logger = logging.getLogger(__name__)
```

[4] The `__name__` attribute of contains the filename of the current file.

Log message

Application example

- Finally, you can combine the `logging` module with `argparse` to interactively control the logging behavior of your command line application:

```
import argparse
import logging

parser = argparse.ArgumentParser(description='Logging example.')
parser.add_argument('--verbose', action='store_true')

args = parser.parse_args()
if args.verbose:
    logging_level = logging.DEBUG
else:
    logging_level = logging.WARNING

format_string = "{name} ( {levelname:10} ) :: {message} "
logging.basicConfig(level=logging_level, format=format_string, style="{")
logger = logging.getLogger(__name__)
logger.debug("A debug message.")
logger.info("An info message.")
logger.warning("A warning.")
...
```

Log message

Non-verbose output

```
$ python logging_example.py
__main__ ( WARNING    ) :: A warning.
__main__ ( ERROR      ) :: An error.
__main__ ( CRITICAL   ) :: A critical error.
```

Verbose output

```
$ python logging_example.py --verbose
__main__ ( DEBUG      ) :: A debug message.
__main__ ( INFO       ) :: An info message.
__main__ ( WARNING    ) :: A warning.
__main__ ( ERROR      ) :: An error.
```

Agenda

1. Overview
2. Context managers
3. Error handling
4. Serializing objects
5. Log messages
6. **Type annotations**

Type annotations

Example from last lecture

- Python supports type hints since version 3.5:

```
from dataclasses import dataclass
@dataclass
class Record:
  id: int
  name: str
  properties: list
  record = Record(1, "name", [])
```

Type annotations

Example from last lecture

- However, type annotation are not enforced:

```
# This is valid although the first to arguments
# are swapped.
record = Record("name", 1, [])
```

Type annotations

CHALMERS
UNIVERSITY OF TECHNOLOGY

Checking types with `mypy`

- To check types an external tool such as `mypy` is required:

```
$ pip install mypy
```

- Then types can be checked as follows:

```
python -m mypy type_example.py
type_example.py:9: error: Argument 1 to "Record" has incompatible type "str"; expected "int"
type_example.py:9: error: Argument 2 to "Record" has incompatible type "int"; expected "str"
Found 2 errors in 1 file (checked 1 source file)
```

Type annotations

Annotating functions

- Types hints can be used to specify types for arguments as well as return type
- Example from Python docs[5]:

```python
Vector = list[float] # Type alias

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]
```

[5] https://docs.python.org/3/library/typing.html

Type annotations

Advanced type annotations

- The `typing` module provides special type objects to specify types for e.g. callables and sequences

```python
from typing import List, Sequence, Callable

Vector = List[float]
Functional = Callable[[Vector], float]

def dot_product(x: Vector, y: Vector) -> float:
    return sum([a * b for a, b in zip(x, y)])

def apply(f: Functional, vectors: Sequence[Vector]) -> Sequence[float]:
    return [f(v) for v in vectors]

apply(dot_product, [[1.0, 0.0], [0.0, 1.0]]) # Doesn't pass type check.
apply(lambda x: dot_product(x, x), [[1.0, 0.0], [0.0, 1.0]])
```

Type annotations

Advantages

- Type hints allow static type checkers to catch logical errors such as this one[6]:

```python
from typing import List, Sequence, Callable

def do_something(input List[int]):
    for i in input:
        i.something() # Error: int has not attribute something.
```

- Type hints make your code easier to understand

[6] Using `mypy` in your IDE will thus help you catch logical errors while programming.

Excercise 3

- Exercise 3 on exercise sheet
- Time: 15 minutes

Summary

What we have learned

- Special syntax to handle errors and context
- Object oriented approaches to storing data, logging
- Using types hints for *gradual typing*

Conclusions

CHALMERS
UNIVERSITY OF TECHNOLOGY

Conclusions

- Python is complex and keeps changing
  - There is a lot to learn
- General principles:
  - Object orientation
  - Customizing behavior using special methods