CUED Part IA Flood Monitoring and Warning System Documentation

Release 2024.0

Garth N. Wells

Jan 11, 2024

PROJECT SPECIFICATION

Lent Term 2024, Department of Engineering, University of Cambridge

Your team has been tasked with building the computational backend (library) to a new real-time flood warning system for England. The library should:

1. Fetch real-time river level data over the Internet from the Department for Environment Food and Rural Affairs data service.
2. Support specified data query types on river level monitoring stations.
3. Analyse monitoring station data in order to assess flood risks, and issue flood warnings for areas of the country.

The mandated development practices are listed in the *Requirements* section. The library is required to support specific query interfaces (API), as outlined in the *Deliverables* section, which form the *public interface* of the library. Another company has been contracted to build a user interface using the prescribed public interfaces to the library, hence they cannot be changed.

Development team

Your development team is your laboratory group.

CHAPTER

ONE

REQUIREMENTS

This section defines the technical requirements for the implementation.

1.1 Language and library structure

Library is to be developed in Python 3 and using multiple modules (files). Each file should collect related functionality.

1.2 Documentation

All classes, methods (a function that belongs to a class) and functions must be documented with a 'docstring'. The docstring shall explain what the class or function does. For a function, the docstring shall make clear the purpose, what arguments are expected and what is returned.

Simple examples of Python docstrings can be viewed here.

1.3 Development practices

The quality of a flood warning software library is paramount; implementation errors could put lives at risk and lead to substantial financial losses. You are therefore required to adopt software engineering best practices. Your team is required to:

- Use the Git version control system (see *Using Git*).
- Provide automated tests for your implementations using pytest to demonstrate the quality of the system (see *Test framework*)
- Use automated continuous integration testing (see *Automated testing*)

1.4 Data source

The system is to be built on the (near) real-time river level data at the nearly 2000 monitoring stations that is made available by the Department for Environment Food and Rural Affairs (DEFRA) at https://environment.data.gov.uk/. For most stations river level data is updated every 15 minutes. The data service is summarised at https://data.gov.uk/dataset/real-time-and-near-real-time-river-level-data1.

REST interface for data retrieval

Data is fetched from https://environment.data.gov.uk/ using a REST interface. With a suitably formed URL (a string), as defined in the service documentation, the server returns the requested data as a JSON object. JSON

objects are represented in Python as data structures made up of dictionaries, lists and strings. JSON objects are straightforward to manipulate from Python. The interface to the DEFRA service is documented at https://environment.data.gov.uk/flood-monitoring/doc/reference.

CHAPTER

TWO

DELIVERABLES

This section defines the project deliverables. Functionality is to be implemented in the module named `floodsystem`.

Milestones and deadlines

Project deliverables/tasks are structured into **two** milestones. Milestone 1 must be delivered by the interim marking session, and Milestone 2 by the final marking session. You may deliver early by signing off at the Help Desk.

Clarifications

Clarifications can be sought at the Help Desk.

Task completion, interfaces and demonstration programs

Each task requires the implementation of functionality that can be accessed via a specified interface, usually a function signature (function name and arguments, and return values). At the end of each task is a description of a demonstration program that must be be provided. Demonstration programs must have the structure:

```
def run():
    # Put code here that demonstrates functionality

if __name__ == "__main__":
    run()
```

You should expect to run demonstration programs during a marking session.

Important: Conforming to the specified public interface is critical as this will allow the interface team to work independently of your development (and it will allow automated testing of your work).

Testing

Write tests as you progress through the tasks (see *Test framework*) and add deliverables and tests to the automated testing system (see *Automated testing*).

Tip: To deliver on a Task, you will often want to implement more functions than just the required function interface. Use additional functions to:

- Modularise and simplify your library.
- Allow re-use of functions across tasks.
- Simplify testing.

As you work through the Tasks, look for opportunities to re-structure code in order to re-use functions.

Units

Distances in kilometres (km) and heights in metres (m).

2.1 Milestone 1

Processing of monitoring station properties.

Deadline
: Mid-term sign-up session

Points
: 4

Caution: Do not use the 'representative output' in your pytest tests. Representative output is provided to help you, but would not be part of a real contract. Moreover, you are working with real-time data which will change.

2.1.1 Task 1A: build monitoring station data

This task has been completed for you in the template repository.

1. In a submodule `station`, create a class `MonitoringStation` that represents a monitoring station, and has *attributes*:
   - Station ID (`string`)
   - Measurement ID (`string`)
   - Name (`string`)
   - Geographic coordinate (`tuple(float, float)`)
   - Typical low/high levels (`tuple(float, float)`)
   - River on which the station is located (`string`)
   - Closest town to the station (`string`)

2. Implement the *methods* `__init__` to initialise a station with data, and `__repr__` for printing a description of the station.

3. In the submodule `stationdata` implement a function that returns a list of `MonitoringStation` objects (for active stations with water level monitoring). To avoid excessive data requests, the function should save fetched data to file, and then optionally read from a cache file. The function should have the signature:

```
def build_station_list(use_cache=True):
```

The data should be retrieved from the online service documented at http://environment.data.gov.uk/flood-monitoring/doc/reference.

Demonstration program

In the program file `Task1A.py`, use the function `stationdata.build_station_list` to build a list of monitoring stations. Print the total number of stations, and a summary of the stations named 'Bourton Dickler', 'Surfleet Sluice' and 'Gaw Bridge'. Representative output is:

```
Number of stations: 1840
Station name:     Bourton Dickler
   id:            http://environment.data.gov.uk/flood-monitoring/id/stations/1029TH
   measure id:    http://environment.data.gov.uk/flood-monitoring/id/measures/1029TH-
↪level-stage-i-15_min-mASD
   coordinate:    (51.874767, -1.740083)
   town:          Little Rissington
   river:         Dikler
   typical range: (0.068, 0.42)
Station name:     Surfleet Sluice
   id:            http://environment.data.gov.uk/flood-monitoring/id/stations/E2043
   measure id:    http://environment.data.gov.uk/flood-monitoring/id/measures/E2043-
↪level-stage-i-15_min-mASD
   coordinate:    (52.845991, -0.100848)
   town:          Surfleet Seas End
   river:         River Glen
   typical range: (0.15, 0.895)
Station name:     Gaw Bridge
   id:            http://environment.data.gov.uk/flood-monitoring/id/stations/52119
   measure id:    http://environment.data.gov.uk/flood-monitoring/id/measures/52119-
↪level-stage-i-15_min-mASD
   coordinate:    (50.976043, -2.793549)
   town:          Kingsbury Episcopi
   river:         River Parrett
   typical range: (0.231, 0.971)
```

2.1.2 Task 1B: sort stations by distance

1. In the submodule `geo` implement a function that, given a list of station objects and a coordinate *p*, returns a list of `(station, distance)` tuples, where `distance` (`float`) is the distance of the `station` (`MonitoringStation`) from the coordinate *p*. The returned list should be sorted by distance. The required function signature is:

```
def stations_by_distance(stations, p):
```

where `stations` is a list of `MonitoringStation` objects and `p` is a tuple of floats for the coordinate *p*.

Tip: The distance between two geographic coordinates (latitude/longitude) is computed using the haversine formula. You could program the haversine formula, or you could use a Python library to perform the computation for you, e.g. https://pypi.org/project/haversine/.

Hint: Build a list of all `(station, distance)` tuples, and use the provided function `utils.sort_by_key` to produce a list that is sorted by the second entry in the tuple.

Demonstration program

Provide a program file `Task1B.py` that uses `geo.stations_by_distance` and prints a list of tuples (station name, town, distance) for the 10 closest and the 10 furthest stations from the Cambridge city centre, (52.2053, 0.1218). The closest 10 entries (e.g., `x[:10]`) in the list may be:

```
[('Cambridge Jesus Lock', 'Cambridge', 0.8402364350834995), ('Bin Brook', 'Cambridge',␣
↪2.502274086951454), ("Cambridge Byron's Pool", 'Grantchester', 4.0720438555077125), (
↪'Cambridge Baits Bite', 'Milton', 5.115589516578674), ('Girton', 'Girton', 5.
↪227070345811418), ('Haslingfield Burnt Mill', 'Haslingfield', 7.044388165868453), (
↪'Oakington', 'Oakington', 7.128249171700346), ('Stapleford', 'Stapleford', 7.
↪265694306995238), ('Comberton', 'Comberton', 7.7350743760373675), ('Dernford',
↪'Great Shelford', 7.993861351711722)]
```

and the furthest 10 (e.g., `x[-10:]`):

```
[('Boscadjack', 'Wendron', 440.0026482838576), ('Gwithian', 'Gwithian', 442.
↪05491558132354), ('Helston County Bridge', 'Helston', 443.37824966454974), ('Loe Pool
↪', 'Helston', 445.07184458260684), ('Relubbus', 'Relubbus', 448.64944322554413), (
↪'St Erth', 'St Erth', 449.0341571188601 5), ('St Ives Consols Farm', 'St Ives', 450.
↪0734690482922), ('Penzance Tesco', 'Penzance', 456.3857579793324), ('Penzance␣
↪Alverton', 'Penzance', 458.5766422710278), ('Penberth', 'Penberth', 467.
↪53367291629183)]
```

2.1.3 Task 1C: stations within radius

1. In the submodule `geo` implement a function that returns a list of all stations (type `MonitoringStation`) within radius r of a geographic coordinate x. The required function signature is:

```
def stations_within_radius(stations, centre, r):
```

where `stations` is a list of `MonitoringStation` objects, `centre` is the coordinate x and `r` is the radius.

Demonstration program

Provide a program file `Task1C.py` that uses the function `geo.stations_within_radius` to build a list of stations within 10 km of the Cambridge city centre (coordinate (52.2053, 0.1218)). Print the names of the stations, listed in alphabetical order. Representative output:

```
['Bin Brook', 'Cambridge Baits Bite', "Cambridge Byron's Pool",
 'Cambridge Jesus Lock', 'Comberton', 'Dernford', 'Girton',
 'Haslingfield Burnt Mill', 'Lode', 'Oakington', 'Stapleford']
```

2.1.4 Task 1D: rivers with a station(s)

1. In the submodule `geo` develop a function that, given a list of station objects, returns a container (`list`/`tuple`/`set`) with the names of the rivers with a monitoring station. The function should have the signature:

```
def rivers_with_station(stations):
```

where `stations` is a list of `MonitoringStation` objects. The returned container should not contain duplicate entries.

Tip: Consider returning a Python set object. A set contains only unique elements. This is useful for building a collection of river names since a set will never contain duplicate entries, no matter how many times a river name is added. A brief example of using a `set` is available here.

---

2. In the submodule `geo` implement a function that returns a Python `dict` (dictionary) that maps river names (the 'key') to a list of station objects on a given river. The function should have the signature:

```
def stations_by_river(stations):
```

where `stations` is a list of `MonitoringStation` objects.

Demonstration program

Provide a program file `Task1D.py` that:

- Uses `geo.rivers_with_station` to print how many rivers have at least one monitoring station (Representative result: 843) and prints the first 10 of these rivers in alphabetical order. Representative output:

```
843 stations. First 10 - ['Addlestone Bourne', 'Adur', 'Aire Washlands',
↪'Alconbury Brook',
 'Aldbourne', 'Aller Brook', 'Alre', 'Alt', 'Alverthorpe Beck', 'Ampney Brook']
```

- Uses `geo.stations_by_river` to print the names of the stations located on the following rivers in alphabetical order:
  - 'River Aire'

    Representative output:

    ```
    ['Airmyn', 'Apperley Bridge', 'Armley', 'Beal Weir Bridge', 'Bingley',
    ↪'Birkin Holme Washlands', 'Carlton Bridge', 'Castleford', 'Chapel Haddlesey
    ↪', 'Cononley', 'Cottingley Bridge', 'Ferrybridge Lock', 'Fleet Weir',
    ↪'Gargrave', 'Kildwick', 'Kirkstall Abbey', 'Knottingley Lock', 'Leeds Crown␣
    ↪Point', 'Saltaire', 'Snaygill', 'Stockbridge']
    ```

  - 'River Cam'

    Representative output:

    ```
    ['Cam', 'Cambridge', 'Cambridge Baits Bite', 'Cambridge Jesus Lock', 'Dernford
    ↪', 'Weston Bampfylde']
    ```

  - 'River Thames'

    Representative output:

```
['Abingdon Lock', 'Bell Weir', 'Benson Lock', 'Boulters Lock', 'Bray Lock',
↪'Buscot Lock', 'Caversham Lock', 'Chertsey Lock', 'Cleeve Lock', 'Clifton␣
↪Lock', 'Cookham Lock', 'Cricklade', 'Culham Lock', 'Days Lock', 'Ewen',
↪'Eynsham Lock', 'Farmoor', 'Godstow Lock', 'Goring Lock', 'Grafton Lock',
↪'Hannington Bridge', 'Hurley Lock', 'Iffley Lock', 'Kings Lock', 'Kingston',
↪ 'Maidenhead', 'Mapledurham Lock', 'Marlow Lock', 'Marsh Lock', 'Molesey␣
↪Lock', 'Northmoor Lock', 'Old Windsor Lock', 'Osney Lock', 'Penton Hook',
↪'Pinkhill Lock', 'Radcot Lock', 'Reading', 'Romney Lock', 'Rushey Lock',
↪'Sandford-on-Thames', 'Shepperton Lock', 'Shifford Lock', 'Shiplake Lock',
↪'Somerford Keynes', 'Sonning Lock', 'St Johns Lock', 'Staines', 'Sunbury ␣
↪Lock', 'Sutton Courtenay', 'Teddington Lock', 'Thames Ditton Island',
↪'Trowlock Island', 'Walton', 'Whitchurch Lock', 'Windsor Park']
```

2.1.5 Task 1E: rivers by number of stations

1. Implement a function in `geo` that determines the N rivers with the greatest number of monitoring stations. It should return a `list` of *(river name, number of stations)* tuples, sorted by the number of stations. In the case that there are more rivers with the same number of stations as the N th entry, include these rivers in the list. The function should have the signature:

   ```
   def rivers_by_station_number(stations, N):
   ```

   where `stations` is a list of `MonitoringStation` objects.

Demonstration program

Provide a program file `Task1E.py` that prints the list of (river, number stations) tuples when $N = 9$. Representative result is:

```
[('Thames', 55), ('River Great Ouse', 31), ('River Avon', 30), ('River Calder', 24), (
↪'River Aire', 21), ('River Severn', 20), ('River Derwent', 18), ('River Stour', 16),␣
↪('River Wharfe', 14), ('River Trent', 14), ('Witham', 14)]
```

The above list has more then 9 entries since a number of rivers have 14 stations.

2.1.6 Task 1F: typical low/high range consistency

It is suspected that some stations have inconsistent data for typical low/high ranges, namely that (i) no data is available; or (ii) the reported typical high range is less than the reported typical low. This needs to be checked so that stations with inconsistent data are not used erroneously in flood warning analysis.

1. Add a *method* to the `MonitoringStation` class that checks the typical high/low range data for consistency. The method should return `True` if the data is consistent and `False` if the data is inconsistent or unavailable. The method should have the signature:

   ```
   def typical_range_consistent(self):
   ```

2. Implement in the submodule `station` a function that, given a list of station objects, returns a list of stations that have inconsistent data. The function should use `MonitoringStation.typical_range_consistent`, and should have the signature:

   ```
   def inconsistent_typical_range_stations(stations):
   ```

   where `stations` is a list of `MonitoringStation` objects.

Demonstration program

Provide a program file `Task1F.py` that builds a list of all stations with inconsistent typical range data. Print a list of station names, in alphabetical order, for stations with inconsistent data. The representative result (at the time of writing) is:

```
['Addlestone', 'Airmyn', 'Allerford', 'Arundel Queen St Bridge', 'Blacktoft', 'Braunton
↪', 'Brentford', 'Broomfleet Weighton Lock', 'East Hull Hedon Road', 'Eccelsfield␣
↪Morrisons', 'Fleetwood', 'Goole', 'Gravesend', 'Hedon Thorn Road Bridge', 'Hedon␣
↪Westlands Drain', 'Hull Barrier Victoria Pier', 'Hull High Flaggs, Lincoln Street',
↪"King's Lynn", 'Littlehampton', 'Paull', 'Salt end', 'Silloth Docks', 'Stone Creek',
↪'Templers Road', 'Topsham', 'Totnes', 'Truro Harbour', 'Weare Giffard', 'Westbrook␣
↪Mill', 'Wilfholme PS', 'Wilfholme PS Hull Level']
```

2.1.7 Optional extensions

1. Display the location of each station on a map (perhaps from Google Maps). Suitable Python libraries tools for this include Bokeh and Plotly.
2. Explore what other station information is available in the retrieved data. The function `stationdata.build_station_list` is a good place to start. Extend `MonitoringStation` to store any interesting station data as attributes.
3. *Advanced*: The `MonitoringStation` attributes (station data) are properties of the station and will not generally change. However, we could accidentally and mistakenly change an attribute in our code. For flood forecasting and warning, such an error could have dire consequences. Use property decorators to prevent accidental modification of the attributes.

2.2 Milestone 2

The focus of the Milestone 2 is processing monitoring station real-time data to warn of flood risks.

Deadline
End-of-term sign-up session

Points
8

Caution: Representative output for each demonstration program is provided as a guide. You will be working with real-time data, so the precise output will change with time.

2.2.1 Task 2A: fetch real-time river levels

This task has been completed for you in the template repository.

1. Extend the `MonitoringStation` class with an attribute `latest_level` (`float`), and implement in the `stationdata` submodule a function that updates the latest water level for all stations in a list using data fetched from the Internet. If level data is not available, the attribute `latest_level` should be set to `None`. The function should have the signature:

```
def update_water_levels(stations):
```

   where `stations` is a list of `MonitoringStation` objects.

Demonstration program

Provide a program file `Task2A.py` that sets the latest water level for all stations, and prints the latest levels at the stations 'Bourton Dickler', 'Surfleet Sluice', 'Gaw Bridge', 'Hemingford' and 'Swindon'. Typical output is:

```
Station name and current level: Bourton Dickler, 0.146
Station name and current level: Surfleet Sluice, 0.84
Station name and current level: Gaw Bridge, 0.463
Station name and current level: Hemingford, 0.197
Station name and current level: Swindon, 1.192
```

2.2.2 Task 2B: assess flood risk by level

1. Add a method to `MonitoringStation` that the returns the latest water level as a fraction of the typical range, i.e. a ratio of 1.0 corresponds to a level at the typical high and a ratio of 0.0 corresponds to a level at the typical low. The method should have the signature:

```
def relative_water_level(self):
```

   If the necessary data is not available or is inconsistent, the function should return `None`.

2. In the submodule `flood`, implement a function that returns a list of tuples, where each tuple holds (i) a station (object) at which the latest relative water level is over `tol` and (ii) the relative water level at the station. The returned list should be sorted by the relative level in descending order. The function should have the signature:

```
def stations_level_over_threshold(stations, tol):
```

where `stations` is a list of `MonitoringStation` objects. Consider only stations with consistent typical low/high data.

Demonstration program

Provide a program file `Task2B.py` that prints the name of each station at which the current relative level is over 0.8, with the relative level alongside the name (do not forget to handle the cases of inconsistent range data). Typical output will be of the form:

```
Ledgard Bridge 3.982
Godstow Lock 1.56198347107438
Windyridge Road 1.4470588235294117
Castle Mill (Bedford) 1.3333333333333328
Newark Weir 1.249999999999988
Cam 1.1813725490196074
Hayes Basin 1.166666666666667
Eckington Sluice 1.0875462392108504
Romney Lock 1.0829268292682928
Pinkhill Lock 1.0524475524475525
.
.
```

Explore your implementation for different tolerances.

2.2.3 Task 2C: most at risk stations

1. Implement a function in the submodule `flood` that returns a list of the N stations (objects) at which the water level, relative to the typical range, is highest. The list should be sorted in descending order by relative level. The function should have the signature:

```
def stations_highest_rel_level(stations, N):
```

where `stations` is a list of `MonitoringStation` objects.

Demonstration program

Provide a program file `Task2C.py` that prints the names of the 10 stations at which the current relative level is highest, with the relative level beside each station name. Typical output will be of the form:

```
Ledgard Bridge 3.982
Godstow Lock 1.56198347107438
Windyridge Road 1.4470588235294117
Castle Mill (Bedford) 1.3333333333333328
Newark Weir 1.249999999999988
Cam 1.1813725490196074
Hayes Basin 1.166666666666667
Eckington Sluice 1.0875462392108504
Romney Lock 1.0829268292682928
Pinkhill Lock 1.0524475524475525
```

2.2.4 Task 2D: level data time history

This task has been completed for you in the template repository.

1. Implement in the submodule `datafetcher` a function that retrieves from the Internet the water level data for a given station 'measure id' over the period from the current time back to d days ago. It should return a tuple with the first entry being the sample times and the second entry being the water levels. The function should have the signature:

```
def fetch_measure_levels(measure_id, dt):
```

Typical use to retrieve the level data at a station over the past 10 days would be:

```
import datetime
dt = 10
dates, levels = fetch_measure_levels(station.measure_id,
                                     dt=datetime.timedelta(days=dt))
```

Demonstration program

Provide a program file `Task2D.py` that fetches and prints the level history at the station 'Cam' over the past 2 days. Typical output:

```
2017-01-08 04:00:00+00:00 0.819
2017-01-08 03:45:00+00:00 0.819
2017-01-08 03:30:00+00:00 0.819
2017-01-08 03:15:00+00:00 0.819
2017-01-08 03:00:00+00:00 0.819
2017-01-08 02:45:00+00:00 0.819
2017-01-08 02:30:00+00:00 0.819
2017-01-08 02:15:00+00:00 0.819
2017-01-08 02:00:00+00:00 0.82
2017-01-08 01:45:00+00:00 0.82
.
.
```

2.2.5 Task 2E: plot water level

1. Implement in a submodule `plot` a function that displays a plot (using Matplotlib) of the water level data against time for a station, and include on the plot lines for the typical low and high levels. The axes should be labelled and use the station name as the plot title. The function should have the signature:

```
def plot_water_levels(station, dates, levels):
```

where `station` is a `MonitoringStation` object.

Hint: Example code to display a plot using Matplotlib:

```
import matplotlib.pyplot as plt
from datetime import datetime, timedelta
```

(continues on next page)

(continued from previous page)

```
t = [datetime(2016, 12, 30), datetime(2016, 12, 31), datetime(2017, 1, 1),
     datetime(2017, 1, 2), datetime(2017, 1, 3), datetime(2017, 1, 4),
     datetime(2017, 1, 5)]
level = [0.2, 0.7, 0.95, 0.92, 1.02, 0.91, 0.64]

# Plot
plt.plot(t, level)

# Add axis labels, rotate date labels and add plot title
plt.xlabel('date')
plt.ylabel('water level (m)')
plt.xticks(rotation=45);
plt.title("Station A")

# Display plot
plt.tight_layout()  # This makes sure plot does not cut off date labels

plt.show()
```

2. *Optional:* In place of Matplotlib, try using a web-centric plotting library such as Bokeh or Plotly.
3. *Optional extension:* Generalise your implementation such that it takes a list of up to 6 stations displays the level at each station as subplot inside a single plot.

Demonstration program

Provide a program file `Task2E.py` that plots the water levels over the past 10 days for the 5 stations at which the current relative water level is greatest.

2.2.6 Task 2F: function fitting

Least-squares polynomial fit

A least-squares polynomial fit is finding a polynomial that 'best' fits data points in the least-squares sense, i.e. for a set of n data points

$$((x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1}))$$

the best-fit polynomial $f(x)$ minimises the error

$$e = \sum_{i=0}^{n-1} (y_i - f(x_i))^2.$$

Details of how to compute least-squares fits is covered in Part IB.

1. In a submodule `analysis` implement a function that given the water level time history (dates, levels) for a station computes a least-squares fit of a polynomial of degree *p* to water level data. The function should return a tuple of (i) the polynomial object and (ii) any shift of the time (date) axis (see below). The function should have the

signature:

```
def polyfit(dates, levels, p):
```

Typical usage for a polynomial of degree 3 would be:

```
poly, d0 = polyfit(dates, levels, 3)
```

where `poly` is a numpy.poly1d object an `d0` is any shift of the date (time) axis.

Hint: To work with dates as function arguments, e.g. a polynomial that depends on time, the dates need to be converted to floats. Matplotlib has a function date2num that from a list of `datetime` objects returns a list of `float`, where the floats are the time in days (including fractions of days) since the year 0001:

```
import matplotlib
x = matplotlib.dates.date2num(dates)
```

Hint: NumPy has tools for computing least-squares fits to data. The below example computes a least-squares fit for some data points, and plots the data points and the least-squares polynomial:

```
import numpy as np
import matplotlib.pyplot as plt

# Create set of 10 data points on interval (0, 2)
x = np.linspace(0, 2, 10)
y = [0.1, 0.09, 0.23, 0.34, 0.78, 0.74, 0.43, 0.31, 0.01, -0.05]

# Find coefficients of best-fit polynomial f(x) of degree 4
p_coeff = np.polyfit(x, y, 4)

# Convert coefficient into a polynomial that can be evaluated,
# e.g. poly(0.3)
poly = np.poly1d(p_coeff)

# Plot original data points
plt.plot(x, y, '.')

# Plot polynomial fit at 30 points along interval
x1 = np.linspace(x[0], x[-1], 30)
plt.plot(x1, poly(x1))

# Display plot
plt.show()
```

Caution: In the above example, if we changed the `x` interval (0, 2) to (10000, 10002), i.e.:

```
x = np.linspace(10000, 10002, 10)
```

NumPy prints the warning message:

```
RankWarning: Polyfit may be poorly conditioned warnings.warn(msg, RankWarning)
```

This message is warning that floating point round-off errors will be significant and will affect accuracy. In simple terms, the issues is that when we raise a number between 10000 and 10002 to a power, small but important differences are effectively 'lost'.

This issues arises if we work with dates converted to floats using `matplotlib.dates.date2num` since it returns the number of days since the origin of the Gregorian calendar. The numbers will therefore be large. A way to improve the situation is with a change-of-variable:

```
import numpy as np
import matplotlib.pyplot as plt

# Create set of 10 data points on interval (1000, 1002)
x = np.linspace(10000, 10002, 10)
y = [0.1, 0.09, 0.23, 0.34, 0.78, 0.74, 0.43, 0.31, 0.01, -0.05]

# Using shifted x values, find coefficient of best-fit
# polynomial f(x) of degree 4
p_coeff = np.polyfit(x - x[0], y, 4)

# Convert coefficient into a polynomial that can be evaluated
# e.g. poly(0.3)
poly = np.poly1d(p_coeff)

# Plot original data points
plt.plot(x, y, '.')

# Plot polynomial fit at 30 points along interval (note that polynomial
# is evaluated using the shift x)
x1 = np.linspace(x[0], x[-1], 30)
plt.plot(x1, poly(x1 - x[0]))

# Display plot
plt.show()
```

2. In the submodule `plot`, add a function that plots the water level data and the best-fit polynomial. The function must have the signature:

```
def plot_water_level_with_fit(station, dates, levels, p):
```

where `station` is a `MonitoringStation` object.

Demonstration program

Provide a program file `Task2F.py` that for each of the 5 stations at which the current relative water level is greatest and for a time period extending back 2 days, plots the level data and the best-fit polynomial of degree 4 against time. Show the typical range low/high on your plot.

Caution: Fitting high-degree polynomials to data is notoriously tricky, especially if the data is not very smooth (as will often be the case with water level data). The problem is that oscillations can appear at the ends of the interval. The is known as Runge's phenomenon. You can observe this with the river level data by increasing the polynomial degree, say to 10, and the time interval, say to 10 days.

2.2.7 Task 2G: issuing flood warnings for towns

1. Using your implementation, list the **towns** where you assess the risk of flooding to be greatest. Explain the criteria that you have used in making your assessment, and rate the risk at 'severe', 'high', 'moderate' or 'low'.

Note: This task is an opportunity to demonstrate your creativity and technical insights.

Tip: Consider how you could forecast whether the water level at a station is rising or falling.

2.2.8 Optional extensions

1. Show all stations on a map, and indicate by colour stations that are (i) below the typical range; (ii) within the typical range; (iii) above the typical range; or (iv) for which there is not level data.
2. Provide a web-based interface to your flood warning system.
3. Incorporate rainfall data from http://environment.data.gov.uk/flood-monitoring/doc/reference into your system.
4. Explore what other data from http://environment.data.gov.uk/flood-monitoring/doc/reference you could use to improve your monitoring and warning system. To start, examine the data that is already being retrieved but has not been used.

CHAPTER

THREE

GETTING STARTED

1. Read the *Requirements* section.
2. Install, configure and test your development environment (*Development environment*).
3. Create a Git repository for your team/project (*Creating a team development repository*) from the provided template.
4. Read *Using Git*.
5. Read the *Deliverables* section, and with your team consider dependencies between 'tasks' in the deliverables and allocate independent tasks to a team member (*Project planning*).
6. Start implementing your tasks (*Editing and executing Python code*).

Tip: Start simple and work in small steps. It is much easier to add functionality to a working program than to fix a complicated non-working program.

Note: When developing your programs, you may need to review the activity notebooks from Michaelmas Term.

3.1 Development environment

Note: Experienced developers have their preferred tools and development environments. If you are experienced with Git, Python and using editors, you are free to use your preferred tools.

The following procedures and tools are suggested.

3.1.1 Option 1: Web-based environment

You can use GitHub Codespaces, which provides a development environment in your browser.

3.1.2 Option 2: Local software installation

1. Install Visual Studio Code (https://code.visualstudio.com/).

   Visual Studio Code will provide instructions on how to install `git` and `python` when you need them. Otherwise, instructions at https://code.visualstudio.com/docs/sourcecontrol/overview and https://code.visualstudio.com/docs/languages/python.

3.1.3 Testing your Python installation

1. Create a file in VS Code with the extension `.py` and enter some simple Python code, e.g.:

   ```
   print("Testing Python install")
   ```

2. Click the 'play' button at the top of the open file.

3.2 Creating a team development repository

1. Log into GitHub (create an account using your @cam.ac.uk email address, or use any other GitHub account you wish).
2. **One team member only:** The template start code is at https://github.com/CambridgeEngineering/PartIA-Flood-Warning-System. Click on the green "Use this template" button, select "Create a new repository" give your new repository a name. Make your repository "private". In the "Settings" section for your repository add your team members as "Collaborators" and share the name of the repository with team members.
3. Clone your team's repository using VS Code "Source control".
4. From VS Code, execute file `Task1A.py`. You should see some output on river level monitoring stations.

Note: The Python code uses some modules (`requests` and `dateutil`) that are not part of the Python standard library. If you see an error that a module is missing, you can install the module using `pip`. Use:

```
pip install requests python-dateutil
```

in the terminal window.

3.3 Editing and executing Python code

1. Launch VS Code and open your local code repository directory.
2. Open/create the files you wish to edit. 'Module' files should go in the directory `floodsystem/`. The `Task*.py` files should go in the root directory of the repository.
3. Use right-click -> 'Run Python File in Terminal' on the program text in VS Code to run the Python code.

As you develop you programs, commit your changes (using Git) and push these to your shared online repository. If you are unsure how often to commit and push changes, err on the side of committing and pushing frequently. *Commit at least upon the completion of each task.*

3.4 Automated testing

The starter template repository includes the file `.github/workflows/pythonapp.yml` which configures automated testing, known as *continuous integration* (CI), on GitHub. Against each commit you will see on the GitHub repository page whether or not the tests are passing.

Edit `.github/workflows/pythonapp.yml` to run your deliverables in the test system and to add code tests to your test suite.

3.5 Project planning

1. Examine the first few project deliverables, and divide independent tasks amongst team members. Each team member can then work on tasks independently.
2. Communicate frequently with team members to update them on your progress, and seek help from a team member if required.
3. As tasks are completed review each others work and provide feedback.
4. As you progress through the tasks, periodically assess which tasks are independent and allocate these to a team member.

CHAPTER

FOUR

DEVELOPMENT TOOLS AND PRACTICES

4.1 Working in a team

Most software is developed in teams, and working effectively in a development team requires certain skills and practices. At a planning level:

- Examine the required tasks, then discuss and decide on the dependencies between tasks. To start, allocate independent tasks to team members.
- Let your team know when a task or piece of functionality is complete.
- Discuss frequently.

At the implementation level:

- Use a version control system, such as Git. With Git:
  - Work that is committed cannot be lost (unless you try really hard) - your team members cannot accidentally delete your code.
  - Commit changes frequently and in small chunks. This makes clear to others what you are working on, and any conflicts will be easier to resolve.
  - It is easy to switch between computers.
- Add tests as functionality is developed. This:
  - Builds confidence that your implementation is correct.
  - Can detect if a change by you or a team member has affected your implementations. (One of the most frustrating situations in team development is when a change by another team members breaks your carefully constructed functionality.)

4.2 Using Git

Git is modern widely used *version control system* (VCS). A version control system tracks changes to source code. It can show what has changed, and who has made changes and when they made them. Git is very powerful and can be challenging to learn. Elementary Git usage for getting started is summarised below.

Git is generally used from the command line (terminal), but here are tools that provide graphical interfaces and some editors (e.g. VS Code) have built-in Git support.

4.2.1 VS Code

VS Code provides helpers for the operations in the following section.

4.2.2 Command-line use

Creating or cloning a repository

To clone a repository (typically hosted by an online service), e.g.:

```
git clone https://github.com/CambridgeEngineering/PartIA-Computing-Michaelmas.git
```

The location for a particular repository can be found on the online repository page.

To create a new repository, create a directory and execute in the directory the command:

```
git init
```

Adding a new file or adding file changes to the staging area

The command:

```
git add myfile.py
```

instructs Git that we want to track the file `myfile.py`, or if the file is already tracked that we will want to add any changes to the repository history.

Committing changes to the project history

The `commit` command commits changes to the project history, and each commit has a 'commit message' associated with it:

```
git commit -m "Complete Task 1C"
```

It is possible at any time to see the changes between any two commits, and to revert a repository to a particular commit.

Collaborating: merging changes

To fetch remote changes into your repository, e.g. changes made by your team mate:

```
git pull
```

In general, you should `commit` your changes before using `pull`.

To send your changes to the remote server:

```
git push
```

If team members have 'pushed' changes, you will need to use `git pull` before you can push. Once you have pushed changes, other team members will receive your changes when they next 'pull'.

Seeing changes in your working directory

The command:

```
git diff
```

shows any changes to your code since the last commit. The command:

```
git status
```

will show any changes to files that are (a) tracked but have changed since the most recent commit, and (b) files that are not tracked (have not been added using `git add`).

Project history

The log of project commits is displayed by the command:

```
git log
```

The output will include the commit messages and the author of each commit.

Project history is shown by online services, like GitHub, and this the simplest way to examine project change. It is also possible to add comments and suggestions on particular code changes to discuss with team members.

How often should I commit changes?

Often. Structure your work into small chunks, and commit after completing each 'chunk'. At the very least, you should commit changes at the completion of each *Task* in the *Deliverables* section.

Also, pull and push frequently.

Getting help with Git

There are many online resources for learning Git, and search engines for very useful. Helpful tutorials for beginners are:

- https://guides.github.com/introduction/git-handbook/
- https://code.visualstudio.com/docs/sourcecontrol/overview
- https://learngitbranching.js.org/
- https://swcarpentry.github.io/git-novice/

4.3 Test framework

Testing is critical for high quality software development, and there are many tools for helping with this. In this project you will use pytest. Some tests are in the project starter repository.

Write tests as you go, and run the tests frequently to check that nothing has been inadvertently broken.

4.3.1 Running tests

`pytest` is very simple to use:

1. Put tests in files starting with `test_`, e.g. `test_data.py`.
2. In the test file, prefix test function with `test_`, e.g.:

```
def test_sum():
    a, b = 2, 3
    assert a + b == 5
```

3. To run all tests in all `test_*.py` files in a directory, use:

```
pytest .
```

To run all tests in the file `test_data,py`:

```
pytest test_data.py
```

`pytest` will print a summary of the number of tests run, with the number that pass and the number that fail.

4.3.2 Writing tests

Aim to have at least one test for every function in your library. Some tests will just check that a function can be called successfully, e.g.:

```
import mymodule

def test_call():
    x = mymodule.do_something(4)
```

More useful test will check results, e.g.:

```
import mymodule

def test_my_sum():
    sum = mymodule.sum(7, -8)
    assert sum == -1
```

Take care when comparing floating point values, since round-off errors can make precise comparison difficult. Use rounding to compare floats, e.g:

```
import math

def test_math_sine():

    x = math.sin(0.0)
    assert round(x, 8) == 0  # 'round' keep 8 digits after the decimal point

    pi = 3.14159265359
    x = math.sin(pi)
    assert round(x, 8) == 0
```

(continues on next page)

(continued from previous page)

```
    pi = 3.14159265359
    x = math.sin(pi/2.0)
    assert round(x - 1, 8) == 0
```

CHAPTER

FIVE

HELP AND FEEDBACK

Get started early to give your team time to seek feedback and resolve issues. Issues/bugs are a feature of all software development and engineering.

There is a significant *design* component to this project. There is no one 'best' solution.

5.1 Help

Help channels for the activity are:

1. Peer support - this is encouraged, but be sure that you understand what you are doing.
2. Moodle forum.
3. Help Desk (see Moodle page for details).

5.2 Feedback

You can get feedback on your work from demonstrators at the Help Desk.

CHAPTER

SIX

SUGGESTIONS FOR EXPERIENCED DEVELOPERS

These development topics are optional, but are suggested for those who are already experienced with Git and Python and those who wish to develop their skills further.

Branching with Git and pull requests

Use a Git *branch* for each task, and merge your topic branch into `master` once it is complete and tests pass. Use merge requests to merge code into the `master` branch.

Code style

Use flake8 for static analysis and to check your code for style.

Test coverage

Check your test coverage using pytest-cov.

Installing the module and using from a Jupyter notebook

The template repository has a `setup.py` file which allows the `floodsystem` module to be installed. Install the module from the project directory using:

```
pip install . --user
```

Once the module has been installed, you should be able to import it from any location. Try using your module from a Jupyter notebook.

CHAPTER

SEVEN

LEARNING OBJECTIVES AND ASSESSMENT

7.1 Learning objectives

Development skills

1. Approaches for working in teams.
2. Designing a working library for specific technical requirements.
3. Working to a realistic project specification.
4. Effective use of version control.
5. Devising tests.

Programming skills

1. Reinforcement of skills developed in Michaelmas Term.
2. Introduction to user modules and multi-file library implementations.
3. Working with user-defined objects.

7.2 Assessment guidelines

The following points will be used in assessing your implementation. Markers will want to view your Git log.

Code

1. Programs should execute without error.
2. Interfaces should conform to the specification in the *Deliverables*.
3. Programs should be correct.
4. Clarity and structure of the implementations.
5. Appropriate re-use of functions.

Documentation and process

1. Documentation of the library (both docstrings and comments in the code).
2. Unit tests.
3. Effective use of version control (commits of small steps with clear messages).
4. Balance of work within the team (as shown by the Git log).
5. Use of continuous integration.

Document license and copyright

Documentation repository

These documents are managed at https://github.com/CambridgeEngineering/PartIA-Computing-Lent-doc.