Hello! Olá! 你好!

- My name is Jácome Cunha
- Contact me: jacomemiguelcunha@gmail.com

Chapter 7 - File Handling

CS 172 - Computer Programming 2
Lanzhou University

These slides use many elements provided in the main bibliographic reference for these lectures:

Programming in Python 3
A Complete Introduction to the Python Language,
2nd Edition,
Mark Summerfield

Outline

Outline

File Handling

- Most programs need to save/load information to/from files
- We will now go in more detail over file handling in Python
- All the techniques we present are platform-independent
  - You can use the example programs in machines with different operating systems/architectures
  - And save a file on one machine and read it on a different one

File Handling

- The programs in this chapter all use the same data collection
  - A set of aircraft incident reports
- An incident report record holds the following data:

Name	Data Type	Notes
`report_id`	`str`	Minimum length 8 and no whitespace
`date`	`datetime.date`	
`airport`	`str`	Non-empty and no newlines
`aircraft_id`	`str`	Non-empty and no newlines
`aircraft_type`	`str`	Non-empty and no newlines
`pilot_percent_hours_on_type`	`float`	Range 0.0 to 100.0
`pilot_total_hours`	`int`	Positive and non-zero
`midair`	`bool`	
`narrative`	`str`	Multiline

- All the code we introduce is attached to the chapter
  - on file `convert-incidents.py`

File Handling

- Before moving on to saving and loading aircraft incidents
  - we go through creating and manipulating incidents
- The `Incidents.py` program defines one custom exception:

```
class IncidentError(Exception): pass
```

File Handling

- Aircraft incidents are held as `Incident` objects
  - We only show the initializer method of the `Incident` class

```
class Incident:
    def __init__(self, report_id, date, airport, aircraft_id,
                 aircraft_type, pilot_percent_hours_on_type,
                 pilot_total_hours, midair, narrative=""):
        assert len(report_id) >= 8 and len(report_id.split()) == 1, \
               "invalid report ID"
        self.__report_id = report_id
        self.date = date
        self.airport = airport
        self.aircraft_id = aircraft_id
        self.aircraft_type = aircraft_type
        self.pilot_percent_hours_on_type = pilot_percent_hours_on_type
        self.pilot_total_hours = pilot_total_hours
        self.midair = midair
        self.narrative = narrative
```

- The report ID is defined as a private attribute
  - this can be seen by the use of `__` before its name;
  - and restricts the access to `report_id`

File Handling

- In Object-Oriented Programming (OOP), it is fundamental to ensure that the internal structure of an object is kept consistent
  - this is achieved by controlling the access and manipulation of attributes
- `report_id` should be private and not changed after its creation
  - adding `__` to its name in its initialization promotes that
  - but we need to make sure users of the class can read it
  - in Python, this can be done defining a *property*:

```
@property
def report_id(self):
    return self.__report_id
```

  - Now, `report_id` will always hold a correct value, since:
    - upon creation of an `Incident`, its value is validated
    - the defined property only allows users to read it, so users cannot change its value!
    - it cannot be changed because there isn't a *setter*

File Handling

- Regarding the other data attributes of `Incident`
  - they were implemented as read/write properties;
- For example, for `date`:

```
@property
def date(self):
    "The incident date"
    return self.__date

@date.setter
def date(self, date):
    assert isinstance(date, datetime.date), "invalid date"
    self.__date = date
```

- In this case, we are also allowing to change/set the value of a date
  - as in `incident.date = datetime.date(2007, 6, 13)`
  - This will *force* the execution of the second method
    - which implements date validation
- So, the date of an `Incident` will always have a value of the correct type

File Handling

- Using setters to change the values of an incident ensures all the data of an `Incident` will always have consistent values
  - which is essential for being able of saving/loading data from a file
- All the properties follow the same pattern
  - and differ only in the details of their assertions
- Since we used assertions, the program will fail if
  - an attempt is made to create an `Incident` with invalid data, or
  - setting an existing incident's property to an invalid value

File Handling

- The collection of incidents is held as an `IncidentCollection`
  - which is a subclass of `dict`, so we get a lot of functionality *for free*:
    - support for the item access operator `[]`
    - get, set and delete incidents
- We present some of its methods:

```
class IncidentCollection(dict):

    def values(self):
        for report_id in self.keys():
            yield self[report_id]

    def items(self):
        for report_id in self.keys():
            yield (report_id, self[report_id])

    # iterator over the keys of IncidentCollection d in the form iter(d)
    def __iter__(self):
        for report_id in sorted(super().keys()):
            yield report_id

    # iterator over the keys of IncidentCollection d in the form d.keys()
    keys = __iter__
```

File Handling

- The collection of incidents is held as an `IncidentCollection`
- It was not necessary to re-implement the initializer
  - `dict.__init__()` is sufficient
- The keys are report IDs and the values are `Incidents`
- We have re-implemented `values()`, `items()`, and `keys()`
  - so that their iterators work in report ID order
    - This works because `values()` and `items()` iterate over the keys returned by `IncidentCollection.keys()`
    - and this method, which is just another name for `IncidentCollection.__iter__()`, iterates in sorted order over the keys provided by `dict.keys()`

File Handling

- Here's an example where we create an `IncidentCollection`

```
>>> kwargs = dict(report_id="2007061289X")
>>> kwargs["date"] = datetime.date(2007, 6, 12)
>>> kwargs["airport"] = "Los Angeles"
>>> kwargs["aircraft_id"] = "8184XK"
>>> kwargs["aircraft_type"] = "CVS91"
>>> kwargs["pilot_percent_hours_on_type"] = 17.5
>>> kwargs["pilot_total_hours"] = 1258
>>> kwargs["midair"] = False
>>> incidents = IncidentCollection()
>>> incident = Incident(**kwargs)
>>> incidents[incident.report_id] = incident
>>> kwargs["report_id"] = "2007061989K"
>>> kwargs["date"] = datetime.date(2007, 6, 19)
>>> kwargs["pilot_percent_hours_on_type"] = 20
>>> kwargs["pilot_total_hours"] = 17521
>>> incident = Incident(**kwargs)
>>> incidents[incident.report_id] = incident
>>> kwargs["report_id"] = "2007052989V"
>>> kwargs["date"] = datetime.date(2007, 5, 29)
>>> kwargs["pilot_total_hours"] = 1875
>>> incident = Incident(**kwargs)
>>> incidents[incident.report_id] = incident
>>> for incident in incidents.values():
...     print(incident.report_id, incident.date.isoformat())
2007052989V 2007-05-29
2007061289X 2007-06-12
2007061989K 2007-06-19
```

Outline

Writing and Reading Binary Data

- Binary formats usually take up the least amount of disk space
- And are usually the fastest to save and load
- The easiest of all possibilities is to use pickles

Introduction to Pickle Module

- The `pickle` module in Python is used for serializing (pickling) and deserializing (unpickling) Python objects
- Serialization refers to the process of converting a Python object into a sequence of bytes
- Deserialization is the inverse operation, where a sequence of bytes is converted back into a Python object

Pickling Objects

```
import pickle

# Object to be pickled
data = {'a': 1, 'b': 2, 'c': 3}

# Pickling the object
with open('data.pkl', 'wb') as fil:
    pickle.dump(data, fil)
```

- The `pickle.dump()` function is used to pickle an object
- The object is stored in a file ('data.pkl' in this case)

Unpickling Objects

```
import pickle

# Unpickling the object
with open('data.pkl', 'rb') as fil:
    data = pickle.load(fil)

print(data)
```

- The `pickle.load()` function is used to unpickle an object
- The object is loaded from a file ('data.pkl' in this case)

Security Concerns

- Pickles offer the simplest approach
- But offer no security mechanisms
  - no encryption, no digital signature
- As pickles can import modules and call functions
  - this raises a potential security threat
  - it might be dangerous to use a pickle from an untrusted source
  - malicious data can cause arbitrary code execution
- Be cautious when unpickling objects from untrusted sources
- Use `pickle.load()` only with trusted data sources

Good afternoon! Boa tarde! 下午好!

Pickles with Optional Compression

- It is often easier to start by writing the saving code before the loading code

```
def export_pickle(self, filename, compress=False):
    fh = None
    try:
        if compress:
            fh = gzip.open(filename, "wb")
        else:
            fh = open(filename, "wb")
        pickle.dump(self, fh, pickle.HIGHEST_PROTOCOL)
        return True
    except (OSError, pickle.PicklingError) as err:
        print("{0}: export error: {1}".format(
              os.path.basename(sys.argv[0]), err))
        return False
    finally:
        if fh is not None:
            fh.close()
```

Pickles with Optional Compression

- To write compressed data, we use `gzip.open()` to open the file
  - Otherwise, we use the built-in `open()` function
- We must use *write binary* mode `wb` when pickling data in binary format
- In Python 3.0 and 3.1, `pickle.HIGHEST_PROTOCOL` is protocol 3, in Python 3.4 is protocol 4, and Python 3.8 is protocol 5
  - a compact binary pickle format
  - the best to use for data shared among Python 3 programs
  - note that newer versions of Python will handle older versions of the protocol
  - but older versions of Python will not handle new versions of the protocol
- We have used a `finally` block to ensure the file is closed at the end
  - whether there was an error or not
- It is important to note that the pickled data is a `dict`
  - and the dictionary's values are `Incident` custom class objects
  - pickles are able to serve objects of most custom classes *for free*

Pickles with Optional Compression

- To read back the data, we need to distinguish between a compressed and an uncompressed pickle
- Any file that is compressed by gzip begins with a *magic number*
  - a magic number is a sequence of one or more bytes
  - it's located at the beginning of a file, used to indicate the file's type
- For gzip files the magic number is the two bytes `0x1F 0x8B`
  - which we store in a `bytes` variable: `b"\x1F\x8B"`

Pickles with Optional Compression

- The code for reading an incidents pickle file is:

```
GZIP_MAGIC = b"\x1F\x8B"
def import_pickle(self, filename):
    fh = None
    try:
        fh = open(filename, "rb")
        magic = fh.read(len(GZIP_MAGIC))
        if magic == GZIP_MAGIC:
            fh.close()
            fh = gzip.open(filename, "rb")
        else:
            fh.seek(0)
        self.clear()
        self.update(pickle.load(fh))
        return True
    except (OSError, pickle.UnpicklingError) as err:
        print("{0}: import error: {1}".format(
              os.path.basename(sys.argv[0]), err))
        return False
    finally:
        if fh is not None:
            fh.close()
```

Pickles with Optional Compression

- We begin by opening the file in *read binary* mode (`rb`)
- We then we read its first bytes
  - we read the number of bytes used by the magic number (hence the `len`)
  - if they are the same as the gzip magic number
    - we close the file
    - and create a new file object using the `gzip.open()` function
  - if not, we call `seek()` to restore the file pointer to the beginning
    - so that the next read, inside `pickle.load()` will be from the start
- We can't assign to `self` since that would wipe out the `IncidentCollection` object that is in use
  - we use `self.clear()` to clear all the incidents and make the dictionary empty
  - and then use `dict.update()` to populate the dictionary with the data loaded from the pickle

Outline

Writing Text

- Writing text is easy but reading it back can be problematic
- We need to choose the structure carefully
- We will store aircraft incidents in the following text format:

```
[20070927022009C]
date=2007-09-27
aircraft_id=1675B
aircraft_type=DHC-2-MK1
airport=MERLE K (MUDHOLE) SMITH
pilot_percent_hours_on_type=46.1538461538
pilot_total_hours=13000
midair=0
.NARRATIVE_START.
    ACCORDING TO THE PILOT, THE DRAG LINK FAILED DUE TO AN OVERSIZED
    TAIL WHEEL TIRE LANDING ON HARD SURFACE.
.NARRATIVE_END.
```

Writing Text

- Each incident record begins with the report ID enclosed in brackets []
- The remaining data is stored in `key=value` form
  - Each item stored in one line
- The multiline narrative text is delimited with special markers
  - `.NARRATIVE_START.` and `.NARRATIVE_START.`,
  - and we indent all the text in between to ensure no line of text could be confused with a marker

```
[20070927022009C]
date=2007-09-27
aircraft_id=1675B
aircraft_type=DHC-2-MK1
airport=MERLE K (MUDHOLE) SMITH
pilot_percent_hours_on_type=46.1538461538
pilot_total_hours=13000
midair=0
.NARRATIVE_START.
    ACCORDING TO THE PILOT, THE DRAG LINK FAILED DUE TO AN OVERSIZED
    TAIL WHEEL TIRE LANDING ON HARD SURFACE.
.NARRATIVE_END.
```

Writing Text

- The code for the `export_text()` function is as follows
  - excluding the `except` and `finally` blocks, similar to previous ones

```
def export_text(self, filename):
    wrapper = textwrap.TextWrapper(initial_indent="    ",
                                   subsequent_indent="    ")
    fh = None
    try:
        fh = open(filename, "w", encoding="utf8")
        for incident in self.values():
            narrative = "\n".join(wrapper.wrap(incident.narrative.strip()))
            fh.write("[{0.report_id}]\n"
                     "date={0.date!s}\n"
                     "aircraft_id={0.aircraft_id}\n"
                     "aircraft_type={0.aircraft_type}\n"
                     "airport={airport}\n"
                     "pilot_percent_hours_on_type="
                     "{0.pilot_percent_hours_on_type}\n"
                     "pilot_total_hours={0.pilot_total_hours}\n"
                     "midair={0.midair:d}\n"
                     ".NARRATIVE_START.\n{narrative}\n"
                     ".NARRATIVE_END.\n\n".format(incident,
                airport=incident.airport.strip(),
                narrative=narrative))
        return True
```

Writing Text

- We begin by creating a `textwrap.TextWrapper` object
  - initialized with the indentation we want to use
    - 4 spaces for the first and subsequent lines
  - by default, the object will wrap lines to a width of 70 characters
- The `wrap()` method takes a string as input
  - and returns a list of strings with suitable indentation
  - and each no longer than the wrap width
- We join this list of lines into a single string using `"\n"` as separator
- The incident date is held as a `datetime.date` object
  - `date!s` forces the string representation of the date when writing
    - this produces the date in ISO 8601, i.e. `YYYY-MM-DD`
- The midair `bool` is formatted as an integer (`midair:d`)
  - this produces 1 for `True` and 0 for `False`
- `str.format()` makes writing text very easy

Parsing Text

- The method for reading and parsing text is longer
- When reading the file we could be in one of several states:
  1. We could be in the middle of reading narrative lines;
  2. We could be at a `key=value` line; or
  3. We could be at a report ID line at the start of a new incident;
- We will look at the `import_text_manual()` method in 8 small parts

Parsing Text

```
def import_text_manual(self, filename):
    fh = None
    try:
        fh = open(filename, encoding="utf8")
        self.clear()
        data = {}
        narrative = None
        ...
```

- The method begins by opening the file in *read text* mode
- Then we clear the dictionary of incidents
- And create the `data` dictionary to hold the data for a single incident
- The `narrative` variable is used for two purposes
  - as a state indicator:
    - if its value is `None`, we are not currently reading a narrative
    - if it is a string (even an empty one), it means that we are reading narrative lines

Parsing Text

- We read text line by line and keep track of the current line number
  - which is used to provide more informative error messages

```
...
for lino, line in enumerate(fh, start=1):
    line = line.rstrip()
    if not line and narrative is None:
        continue
    ...
```

- We begin by stripping off any trailing whitespace from the line
  - if we get an empty line, and we are not in the middle of a narrative,
  - we simply skip to the next line.
  - This means the number of blank lines between incidents doesn't matter
    - but that we preserve any blank lines within the narrative texts

Parsing Text

- If the `narrative` is not `None`, we know we are in a narrative

```
for lino, line in enumerate(fh, start=1):
    ...
    if narrative is not None:
        if line == ".NARRATIVE_END.":
            data["narrative"] = textwrap.dedent(narrative).strip()
            if len(data) != 9:
                raise IncidentError("missing data on "
                                    "line {0}".format(lino))
            incident = Incident(**data)
            self[incident.report_id] = incident
            data = {}          # clear the data dictionary
            narrative = None   # reset narrative ready for the next record
        else:
            narrative += line + "\n"
    ...
```

- If the line is the narrative end marker, we know that we have finished reading the narrative and the incident
- In this case, we put the narrative text into the `data` dictionary, and
- If we have 9 pieces of data, we create an incident and store it
  - otherwise, we raise an error
- If the line isn't the narrative end marker, we append it to the narrative

Parsing Text

```
for lino, line in enumerate(fh, start=1):
    ...
    elif (not data and line[0] == "[" and line[-1] == "]"):
        data["report_id"] = line[1:-1]
```

- If the `narrative` is `None`, then we are either:
  - reading a new report ID
  - or reading some other data
- We can only be at a new report ID if the `data` dictionary is empty, **and** if the line begins with `[` and ends with `]`
  - in this case, we put the report ID into the `data` dictionary
  - this `elif` condition will not be `True` again until `data` is cleared

Parsing Text

```
for lino, line in enumerate(fh, start=1):
    ...
    elif "=" in line:
        key, value = line.split("=", 1)
        if key == "date":                # value is text, needs to be converted
            data[key] = datetime.datetime.strptime(value, "%Y-%m-%d").date()
        elif key == "pilot_percent_hours_on_type":
            data[key] = float(value)     # value is text, needs to be converted
        elif key == "pilot_total_hours":
            data[key] = int(value)       # value is text, needs to be converted
        elif key == "midair":
            data[key] = bool(int(value)) # value is text, needs to be converted
        else:
            data[key] = value
```

- If we are not in a narrative and are not reading a new report ID, there are only three more possibilities:
  - we are reading `key=value` items
  - we are at a narrative start marker

    ```
    elif line == ".NARRATIVE_START.":
        narrative = ""                   # narrative becomes the empty string
    ```

  - something went wrong

    ```
    else:
        raise KeyError("parsing error on line {0}".format(lino))
    ```

Parsing Text

```
            ...
            return True
        except (OSError, ValueError, KeyError,
                IncidentError) as err:
            print("{0}: import error: {1}".format(
                  os.path.basename(sys.argv[0]), err))
            return False
        finally:
            if fh is not None:
                fh.close()
```

- After reading all the lines we return `True` to the caller
- If an exception occurs, the `except` block catches the exception, prints an error message, and returns `False`
- And no matter what, if the file was opened, it is closed at the end

Outline

Usage Example

- You can now use `Incidents` and `IncidentCollections`
  - With data read/stored from/to files
- In file `incidents.ait` we release a collection of incidents
  - in textual format, you can actually open and inspect it
- In module `Incidents.py` we release a module holding both classes
  - you can use it, e.g., in the interpreter:

```
>>> import Incidents
>>> incidents = Incidents.IncidentCollection()
>>> incidents.import_text_manual("incidents.ait")
True
>>> next(incidents.values()).report_id
'20070102000049C'
>>> next(incidents.values()).date
datetime.date(2007, 1, 2)
>>> next(incidents.values()).airport
'WILLIAM P HOBBY'
```

Usage Example

- You can also define programs that use `Incidents` and `IncidentCollections`
- This was done in file `convert_incidents.py`, in which we release a format converter
  - you can use it, e.g., in the terminal to convert the textual to a pickle representation:

    ```
    > python3 convert-incidents.py incidents.ait incidents.aip
    ```

  - you can now change the representation back:

    ```
    > python3 convert-incidents.py incidents.aip read_me.ait
    ```

  - and check that `incidents.ait` and `read_me.ait` have the same content