# Practical Programming
## *in Python*

*Inspired by 'Practical Programming' by Paul Gries, Jennifer Campbell, Jason Montojo*

## Lecture 10
## *Reading & Writing Files*

*Reading Files, Files from the Internet, Writing Files, File Formats, Parsing Files*

Kurt Rinnert, Kate Shaw

Physics Without Frontiers

The Abdus Salam International Centre for Theoretical Physics

# Abstract

*"Keep knowledge in plain text."*
*– The Pragmatic Programmer*

Computers store data in files of various types.
Arguably, the most powerful file type is plain text.
Programmers need to know how to read data from plain text files and write data to them.
Python provides powerful tools for file I/O.

# Overview

- File types and formats
- Reading data from files
- Techniques for reading files
- Files from the internet
- Writing data to files
- File structures & parsing

**File handling is an important skill for the practical programmer.**

# What Kinds of Files Are There?

- There are many kinds of files:
  - Text files
  - Music files
  - Videos
  - Word processor & presentation files
- Many contain complicated formatting information and require special programs to be useful
- *Plain text* files only contain characters and *no* formatting information
- They often have a *syntax* (for example, Python programs)

**Empty File Sizes**

| File Type | Size |
|---|---|
| Microsoft Word | 21 KB |
| OpendDocument Text | 8 KB |
| Pages Document | 29 KB |
| **Plain Text** | **0 KB** |

**Plain text files are human-readable. Everyone is free to write programs using them.**

## Data File Examples

- We have prepared several example data files for you
- You can them in the folder

```
practical_programming/data/
```

  - in your home directory
  - on the USB drive
- They are all plain text files

**The following assumes you understand the concepts explained in the Linux Tutorial.**

# Opening a File

- The function `open` opens a file (much like you open a book when you want to read it)
- `open` returns a *file object*
- The file object knows how get information from the file, how much you read and what you are about to read next
- The *file cursor* keeps track of the current position in the file (much like a bookmark)
- `'file_example.txt'` is the *file path*
- `'r'` tells Python we want to read the file

```
f = open('file_example.txt', 'r')
contents = f.read()
print(contents)
f.close()
```

## File Modes

| Flag | Operation |
|---|---|
| `'r'` | read from existing file |
| `'w'` | write new file (clobbers existing file) |
| `'a'` | append to file (creates file if needed) |

**Beware: opening an existing file with `'w'` will delete its contents!**

# Reading the Entire File

- The statement `contents = f.read()` reads the entire file into a string
- The third statement prints that string
- Newline characters in text files are treated just like any other character
- The last statement closes the file and releases all resources associated with it

```
f = open('file_example.txt', 'r')
contents = f.read()
print(contents)
f.close()
```

- Program output:

```
First line of text
Second line of text
Third line of text
```

**With this approach we have to remember to close the file.**

# The with Statement

- Python provides a **with** statement that automatically closes the file for us:

```
with open('file_example.txt', 'r') as f:
    contents = f.read()
print(contents)
```

- The general form of the **with** statement is

```
with open(filepath, mode) as variable:
    block
```

- The file is closed after the block is executed

**We always use the with statement when opening files.**

# Techniques for Reading Files

- Python provides several techniques for reading files:
  - `read`
  - `readlines`
  - `for line in file`
  - `readline`
- All techniques start reading from the current position of the file cursor
- This allows to seamlessly combine them

**It is often necessary to combine the techniques to read complicated files.**

# The Read Technique

- When no argument is given, `read` reads from the current file cursor to the end of the file
- We have seen it before:

```
with open('file_example.txt', 'r') as f:
    contents = f.read()
print(contents)
```

```
First line of text
Second line of text
Third line of text
```

- With an integer argument, `read` reads the specified number of characters
- The file cursor is moved forward accordingly

```
with open('file_example.txt', 'r') as f:
    first_ten = f.read(10)
    the_rest = f.read()
print('First ten:', first_ten)
print('The rest:', the_rest)
```

```
First ten: First line
The rest:  of text
Second line of text
Third line of text
```

**Beware of reading entire large files – memory is limited.**

## Reading at the End of a File

When the file cursor is at the end of a file the functions `read`, `readlines` and `readline` return the empty string.
If you need to read the contents of the file again you can close and reopen the file.
There are also low-level ways to move the file cursor, but we will not cover them here.

# The Readlines Technique

- We use this when we want to get a *list of strings* containing the lines from a file
- The function `readlines` works similar to `read`
- As with `read`, the file cursor is moved to the end of the file
- Here is an example:

```
with open('file_example.txt', 'r' as example_file:
    lines = example_file.readlines()

print(lines)
```

```
['First line of text.\n', 'Second line of text.\n', 'Third line of text.\n']
```

**Note that the newline characters were *not* removed. The last line might not have one.**

# The Readlines Technique

- Assume `planets.txt` contains the following:

```
Mercury
Venus
Earth
Mars
```

- We get a list of the lines like this:

```
>>> with open('planets.txt', 'r') as pf:
...     planets = pf.readlines(
...
>>> planets
['Mercury\n', 'Venus\n', 'Earth\n', 'Mars']
```

- We can print the lines in reverse order:

```
>>> for planet in reversed(planets):
...     print(planet.strip(
...
Mars
Earth
Venus
Mercury
```

- Or we can sort the lines alphabetically:

```
>>> for planet in sorted(planets):
...     print(planet.strip(
...
Earth
Mars
Mercury
Venus
```

**The warning for large files also applies here.**

# Processing Files Line by Line

- For example, let's print the line lengths:

```
>>> with open('planets.txt', 'r') as data_file:
...     for line in data_file:
...         print(len(line))
...
8
6
6
4
```

- The same with the newlines stripped:

```
>>> with open('planets.txt', 'r') as data_file:
...     for line in data_file:
...         print(len(line.strip()))
...
7
5
5
4
```

**This is safe, even for very large files.**

# The Readline Technique

- This technique reads one line at a time
- We use it when we only want to read part of a file
- This is often necessary to skip *headers* or *comments*
- For example, the file `hopedale.dat` (a TSDL file) describes the number of colored fox fur pelts produced in Hopedale, Labrador, in the years 1834 – 1925
- Here are the first few lines, until the year 1842:

```
Coloured fox fur production, HOPEDALE, Labrador,, 1834-1925
#Source: C. Elton (1942) "Voles, Mice and Lemmings", Oxford Univ. Press
#Table 17, p.265--266
      22
      29
       2
      16
      12
      35
       8
      83
     166
```

**Note the header and the comments in the first few lines.**

# The Readline Technique

- We combine `readline` and line-by-line processing to compute the total number of pelts:

```
with open('hopedale.dat', 'r') as hopedale_file:
    hopedale_file.readline()

    data = hopedale_file.readline().strip()
    while data.startswith('#'):
        data = hopedale_file.readline().strip()

    total_pelts = int(data)

    for data in hopedale_file:
        total_pelts = total_pelts + int(data.strip())

print("Total number of pelts:", total_pelts)
```

- This is the output:

```
Total number of pelts: 4382
```

**Both techniques are safe to use with large files.**

# Files over the Internet

- The file containing the data we want could be on a computer half around the Earth
- If the file is accessible in the internet, we can still access it very much like a local file
- For example, the Hopedale data:

  http://robjhyndman.com/tsdldata/ecology1/hopedale.dat

- There is an important difference: there are many types of files (images, music, text…)
- Python can't assume files from the internet are plain text
- For this reason, we don't read characters from these files but a different type: `bytes`

**The bytes need to be *decoded* using the correct *encoding*. This quickly gets complicated.**

## What is a Byte?

To a computer, information is nothing but *bits*, which we think of as ones and zeros.
All data – for example, characters, sounds, and pixels – are represented as sequences of bits.
Most modern computers organize these bits into groups of eight. Each such group of bits is called a *byte*.
Programming languages interpret sequences bytes for us and let us think of them as integers, strings, functions, and documents.

# Files over the Internet

- Many encodings are documented in the Python online documentation:

  http://docs.python.org/3/library/codecs.html#standard-encodings

- The Hopedale data is encoded using UTF-8
- The module `urllib.urlrequest` provides functions for reading files from the internet:

```
import urllib.request
url = 'http://robjhyndman.com/tsdldata/ecology1/hopedale.dat'
with urllib.request.urlopen(url) as webpage:
for line in webpage:
    line = line.decode('utf-8')
    line = line.strip()
    print(line)
```

**In doubt, use UTF-8 for text-like data. Be polite, avoid unintended DOS attacks.**

# Writing Files

- This programs writes the words `Computer Science` to a file called `topics.txt`:

```
with open('topics.txt', 'w') as output_file:
    num_characters = output_file.write('Computer Science')
print(num_characters)
```

```
16
```

- The `'w'` mode creates a new file or overwrites an existing file
- We can also append to a file using the `'a'` mode:

```
with open('topics.txt', 'a') as output_file:
    output_file.write('Software Engineering')
```

**What will the resulting file look like?**

# Writing Files

- As opposed to `print`, the `write` method does not append a newline
- When writing to files you have to specify *all* characters you want to write
- This includes newline characters:

```
with open('topics.txt', 'w') as output_file:
    output_file.write('Computer Science\n')

with open('topics.txt', 'a') as output_file:
    output_file.write('Software Engineering\n')
```

- This yields the file with the following contents:

```
Computer Science
Software Engineering
```

**When we write to a file we write a *stream* of characters.**

# Missing Values

- It is quite common that data values are missing
- For example, the `hebron.dat` file on the right

```
...
93
55
262
-
102
...
```

```
for line in input_file:
    n = int(line.strip())
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./read_smallest.py", line 19, in smallest_value
  n = int(line.strip())
ValueError: invalid literal for int( with base 10: '-')
```

**This is a very common problem.**

# Missing Values

- We could deal with that in this way:

```
for line in input_file:
    line = line.strip()
    if line != '-':
        n = int(line)
```

```
...
93
55
262
-
102
...
```

**Why is this not very satisfying?**

# Asking for Forgiveness

- `ValueError` is an *Exception*
- Exceptions are types related to errors
- We use exceptions to ask for fogiveness:

```
for line in input_file:
    try:
        n = int(line.strip())
    except ValueError:
        pass # or print a warning message
```

```
...
93
55
262
-
102
...
```

**It is better to ask for forgiveness than permission.**

# Whitespace-Delimited Data

- The file linked below contains information about lynx pelts in the years 1821-1934

  http://robjhyndman.com/tsdldata/ecology1/lynx.dat

- You can also find the file in `practical_programming/data`
- All values are integers
- Each line contains many values separated by whitespace
- We use the string method `split` to create a list of the values
- To get integer values we also get rid of the trailing dots

```
for line in lynx_file:
    tokens = line.split()
    for token in tokens:
        value = int(token[:-1])
```

**The `split` method is is your friend when reading whitespace delimited data.**

# Multi-line Records

- The file on the right describes the arrangement of atoms in a molecule
- The first line contains the name
- Then follow the atoms and their positions, followed by and *end marker*

```
COMPND      AMMONIA
ATOM      1  N  0.257  -0.363   0.000
ATOM      2  H  0.257   0.727   0.000
ATOM      3  H  0.771  -0.727   0.890
ATOM      4  H  0.771  -0.727  -0.890
END
```

**So far no problem. We know how to read this file.**

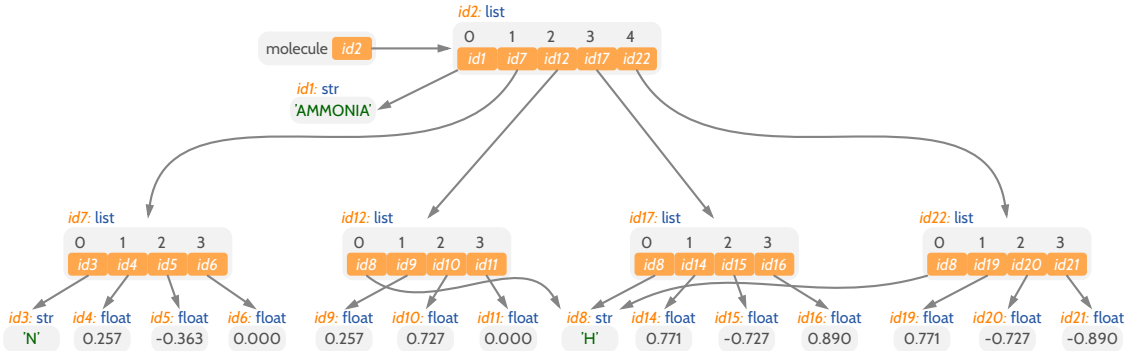
# Multi-line Records: Reading One Molecule

```python
def read_molecule(reader):
    line = reader.readline()
    if not line:
        return None

    key, name = line.split()

    molecule = [name]
    line = reader.readline()

    # Parse all the atoms in the molecule.
    while not line.startswith('END'):
        key, num, atom_type, x, y, z = line.split()
        molecule.append([atom_type, x, y, z])
        line = reader.readline()

    return molecule
```

```
COMPND      AMMONIA
ATOM      1  N  0.257  -0.363   0.000
ATOM      2  H  0.257   0.727   0.000
ATOM      3  H  0.771  -0.727   0.890
ATOM      4  H  0.771  -0.727  -0.890
END
COMPND      METHANOL
ATOM      1  C  -0.748  -0.015   0.024
ATOM      2  O  0.558   0.420  -0.278
ATOM      3  H  -1.293  -0.202  -0.901
ATOM      4  H  -1.263   0.754   0.600
ATOM      5  H  -0.699  -0.934   0.609
ATOM      6  H  0.716   1.404   0.137
END
```

**Several multi-line records are a common file syntax.**

# The Molecule Memory Model




**The memory model reflects the file structure.**

# Multi-line Records: Reading All Molecules

```
def read_all(reader):
    result = []
    reading = True

    while reading:
        molecule = read_molecule(reader)
        if molecule:
            result.append(molecule)
        else:
            reading = False

    return result

if __name__ == '__main__':
    molecule_file = open('molecules.pdb', 'r')
    molecules = read_all(molecule_file)
    print(molecules)
```

```
COMPND      AMMONIA
ATOM      1  N  0.257  -0.363   0.000
ATOM      2  H  0.257   0.727   0.000
ATOM      3  H  0.771  -0.727   0.890
ATOM      4  H  0.771  -0.727  -0.890
END
COMPND      METHANOL
ATOM      1  C  -0.748  -0.015   0.024
ATOM      2  O  0.558   0.420  -0.278
ATOM      3  H  -1.293  -0.202  -0.901
ATOM      4  H  -1.263   0.754   0.600
ATOM      5  H  -0.699  -0.934   0.609
ATOM      6  H  0.716   1.404   0.137
END
```

**Several multi-line records are a common file syntax.**

# Exercises Lecture 10