ENGSCI 233 – Computational Techniques and Computer Systems

Week 2: Data and Objects PY04 – File Input-Output

Andreas W. Kempa-Liehr & Colin Simpson

THE UNIVERSITY OF AUCKLAND NEW ZEALAND | ENGINEERING

Department of Engineering Science

10.03.2022 – slides˙data˙2022.pdf, Rev. ed82f93

Working with Data

This module focuses on working with data that is stored in files located somewhere on a computer system.

This module will develop your understanding of fundamental concepts necessary for working with data:

- File input/output (I/O).
- File system manipulation.
- Representation of data in a data-type.

Working with Data

There are some key operations related to working with data:

- Locating files and directories in the file system.
- Copy, move and delete files in the file system.
- Create and delete directories in the file system.
- Opening and closing files.
- Reading data and/or metadata from a file.
- Writing data and/or metadata to a file.

File System

There are two types of object in a file system:

- File: can contain metadata and data.
- Directory: can contain files and/or directories.

Starting from the root directory, we can follow the path of directories to any file or directory located on the file system.

Root Directory

You are very likely to be running one of the following operating systems on your computer: Windows, Mac OS, or Linux.

Mac OS and Linux are often referred to as Unix-like.

The root directory is defined differently for each:

- Windows: indicated by a drive letter, typically `C:\`
- Unix-like: the root directory is simply `/`

Note that the directory separator is a back slash in Windows, and a forward slash in Unix-like.

Change Directory

On a command line, we can issue change directory commands, with `cd`, to navigate the file system.

While the `cd` command is available on any command line, its usage is very different on Windows and Unix-like.

Fortunately, there should be plenty of online documentation for whatever type of command line you are using.

Absolute and Relative Paths

There are two common methods used to identify the path to a specific location in the file system:

- Absolute path: the path starting from the root directory.
- Relative path: the path starting from the current/working directory e.g. where your Python script is run from.

The absolute path can often be very long, but will never fail. Relative paths are often used as they are shorter, but will fail if starting from the wrong directory.

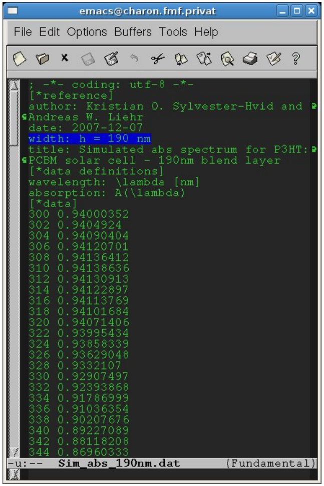
Metadata and Data

A file will commonly contain both metadata and data. They are structured as follows:

- Metadata, which describes the data, is often stored in *header line(s)*. Similar to column headings in an excel spreadsheet.
- The data typically follows the metadata.

We typically read in the metadata and data separately.

File with metatdata and data



```
; -*- coding: utf-8 -*-
[*reference]
author: Kristian O. Sylvester-Hvid and
Andreas W. Liehr
date: 2007-12-07
width: h = 190 nm
title: Simulated abs spectrum for P3HT:
PCBM solar cell - 190nm blend layer
[*data definitions]
wavelength: \lambda [nm]
absorption: A(\lambda)
[*data]
300 0.94000352
302 0.9404924
304 0.94090404
306 0.94120701
308 0.94136412
310 0.94138636
312 0.94130913
314 0.94122897
316 0.94113769
318 0.94101684
320 0.94071406
322 0.93995434
324 0.93858339
326 0.93629048
328 0.9332107
330 0.92907497
332 0.92393868
334 0.91786999
336 0.91036354
338 0.90207676
340 0.89227089
342 0.88118208
344 0.86960333
-u:--  Sim_abs_190nm.dat        (Fundamental)
```

Riede, . . ., Liehr (2010): *Computer Physics Communication* 181, 651–662

Opening/Closing a File in Python

Opening a file in Python requires the path to that file. Python will assign it a file pointer, allowing multiple open files simultaneously.

There are a few considerations when opening a file in Python:

- Existing files should be opened in read-only mode. If opened in write mode, existing data may be eliminated!
- You can create a new file in which to write data.
- Close the file when I/O operations are completed.

Metadata and Data Structure

It is helpful, when reading data from a file, to have pre-existing knowledge about how the metadata and data are structured:

- Number of header lines.
- Number of columns of data.
- Number of rows of data.
- The character(s) used to separate data in the columns i.e. delimiter.

For example, a comma-separated values (CSV) file typically has:

- A header line describing the data in each column.
- A comma delimiter between columns.

Reading Data in Python

Typically, we read in the metadata and data separately.

An example of how we could read in data from a file is given in the Jupyter notebook. A brief summary of that code:

- Open the file.
- Read in the metadata.
- Iteratively read in the data line by line. Split each line of data into components and store these into array-like variables.
- Close the file once all lines have been read in.

Writing Data to Files

Each programming language has its own way of formatting when writing data to screen or a file. Within Python itself, there are multiple different formatting methods (see *this page*).

The Jupyter notebook uses the `str.format()` method, and includes some examples to peruse.

A few key considerations in writing format statements:

- Different format codes for data types (e.g. `d` for integer.)
- Rounding to specific level of accuracy.
- Fixed character width for improved human readability.

File System Manipulation

It is most common to use a command line (batch on Windows, bash on Unix-like) to manipulate the file system.

Python offers some basic file system commands:

- Cope, move or delete files.
- Create and delete directories.
- Wildcard notation for selecting files and directories.

See the Jupyter notebook for some examples. Be careful with these commands, or you might delete the wrong files forever!

Introduction to Data Types

Once data is available in a computer program (e.g. read in from file, or generated during a simulation), it is often convenient to store it as a particular data type. Some already familiar types:

- Integer
- Float
- NumPy Array
- Python List
- String

Here is a handy reference list of some common data types.

Custom Data Types

Sometimes, an existing data type will not be the most effective tool for storing our data.

Later, we will define some of own custom data types:

- Linked List
- Network

This will be achieved through Object-Oriented Programming (OOP). Check out the supplementary Jupyter notebook.

Object-Oriented Programming

MATLAB introduced you to procedural, structured programming i.e. code with statements, loops, control blocks and functions.

Python is multi-paradigm i.e. it can support many different programming styles (including procedural programming).

Python is particularly well suited to object-oriented programming. We will introduce the idea of objects, with attributes and methods.

Defining a Class

```
class Animal(object):
    def __init__(self):
        self.species = 'unknown animal'
        self.name = 'unnamed'
        self.age = 0
```

This defines a class called `Animal`. It currently has:

- One method: `__init__(self)`. Think of this as a function specific to the class.
- Three attributes: `species`, `name` and `age`. Think of these as variables specific to the class.

Creating an Object

We can create any number of objects belonging to a defined class:

```
animal1 = Animal()
animal2 = Animal()
```

Both objects are instances of the same `Animal` class. When creating a new object, the `__init__(...)` method is run i.e. think of it as a special method used for initialisation.

What are the current attributes of `animal1` and `animal2`?

Modifying Object Attributes

We can use or modify an object attribute similarly to a variable:

```
animal1.species = 'human'
animal1.name = 'Andreas Kempa-Liehr'
animal1.age = 49

animal2.species = 'dog'
animal2.name = 'Angus'
animal2.age = 12

print(animal1.species, animal2.species)
print(animal1.age > animal2.age)
```

Defining a Node Class

```
class Node(object):
    def __init__(self, value, pointer):
        self.value = value
        self.pointer = pointer
```

To create an object of this class, we need to provide two arguments for `value` and `pointer`. For example:

```
nd3 = Node(3, None)
```

Why don't we provide an argument for `self`?

Understanding `self`

When defining a method for a class, it will always expect `self` as its first argument.

The object for which we are using the method will become the `self` argument. This allows a method to interact with the object.

Chaining Attributes

The `pointer` attribute of `class Node` is intended to be set as another object of that class. For example:

```
nd3 = Node(3, None)
nd2 = Node(2, nd3)
nd1 = Node(1, nd2)
nd0 = Node(0, nd1)
```

Therefore, `nd0.pointer` should be equal to the object `nd1`.

What will `nd0.pointer.pointer` return?

What will `nd1.pointer.pointer.value` return?

Applications of Object-Oriented Programming

Using the Python `Class` will allow us to define/create some of our own advanced data-types:

- linked list: an extension of the basic list/array that specifically connects adjacent items together.
- network: a generalised form of the linked list, which allows multiple connections between different items/nodes.

These dedicated data-types have many useful and powerful applications, as we will later see.

Linked List: Introduction

A linked list is a sequence of node objects. Each node has two attributes:

- `value`: the value that would normally be stored in an array.
- `pointer`: the next node in the sequence.

To access a specific node object, we start at the head node and follow the sequence of pointers to the desired node.

Linked List vs Array

- Memory: linked lists must store both values and pointers. An array stores just the values.
- Ease of access: linked lists have *sequential access*, which can be time-consuming for a long sequence. Arrays have *random access*, which requires only the pointer to the array start and an array index.
- Manipulation: inserting/deleting items in an array can potentially be very time-consuming. A linked list only requires re-assigning pointer of previous node, and assigning pointer of new node.

Defining a Linked List Class

Defining a basic linked list class:

```
class LinkedList(object):
    def __init__(self):
        self.head = None
```

This single attribute, `head`, will be set to the first node object in the linked list sequence. Each subsequent item of the linked list is another node.

Recall that each node object has a value and pointer to another node object (except the final node, which has a pointer of `None`).

Methods of Linked List Class

A summary of the methods defined for `class LinkedList`:

- `append(...)`: insert a new node at end of linked list.
- `insert(...)`: insert a new node at a specific location in linked list.
- `get_length(...)`: returns the number of nodes of the linked list.
- `get_node(...)`: returns the node object at a specific location in the linked list.
- `get_value(...)`: returns the node value at a specific location in the linked list.

Networks: Introduction

A network can be thought of as a set of nodes connected to each other by pointers (or arcs in this case).

A linked list, where nodes have a single outward connection to one other node, could be considered a special case of a network.

Consider a railway network as an example:

- An individual train station can be represented as a node, with its name as an attribute.
- Stations are connected by train lines, which can be represented by an arc.

Defining a new Node Class

```
class Arc(object):
    def __init__(self):
        self.name = None
        self.value = None
        self.arcs_in = []
        self.arcs_out = []
```

Each node will have a name (i.e. string), a value, and a list of arc objects that either start at this node (`arcs_out`), or end at this node (`arcs_in`).

Note that this is a little different to our Node class for the linked list data-type.

Defining an Arc Class

```
class Arc(object):
    def __init__(self):
        self.weight = None
        self.from_node = None
        self.to_node = None
```

Each arc will have a weight value, a node object as its origin, and a node object as a destination.

Defining a Network Class

```
class Network(object):
    def __init__(self):
        self.nodes = []
        self.arcs = []
```

The only attributes required for the Network class is a list of all node and arc objects.

These node and arc objects themselves contain all the information about how the network is structured.

The focus of the lab is to implement methods for a network.