Programming With Data (CM2015)

Course Notes

Felipe Balbi

November 25, 2020

Contents

Week 1

Key Concepts

- Set up and run Jupyter Notebook on a Windows, Mac or Linux operating system.
- Write and explain simple Python programs using variables and mathematical operators.

1.01 Welcome to the course

Python was chosen for this course due to its simplicity, ease of use and collection of freely available tools.

A print statement in python is simply:

```
1 print("Welcome to Python!")
```

We can also write our code in a file ending with the extension `.py` and run it through the Python interpreter, like so:

```
1 $ python my_file.py
```

During the course we will rely heavily on Jupyter Notebooks. This will give us a nice interface to work with.

1.06 Introduction to development environments and Python

There are many Development Environments available. Only **emacs** is worth your time. Some folks will swear that *VI* is great, however, remember that *VI VI VI* is the number of the beast (trollface).

Jokes aside, a development environment is a very personal choice. One can visit World Class Text Editor section for a small list of what's available.

1.07 Getting to grips with Python

Read the following introductory reading, which will help you get to grips with Python:

McKinney, W. Python for data analysis: data wrangling with Pandas, NumPy, and IPython. (Sebastopol, CA: O'Reilly, 2017) 2nd edition, Chapter 1 Preliminaries and Chapter 2 Python Language Basics, IPython, and Jupyter Notebooks, pp.1–46.

Available here.

1.103 Jupyter code cells

When we create a new Notebook in Jupyter, it comes with what are referred to as *cells*. Cells can be defined in terms of what they can do and we can change their types too.

Code cells can be used to write code. In our case using Python.

Markdown cells are used to write Markdown, which will serve as documentation of textual input.

Raw NBConvert probably won't be used and won't be discussed.

1.105 Jupyter Notebook basics

Click on the links to below to read about Jupyter Notebook basics and using markdown cells in Jupyter:

- Jupyter Notebook basics Using markdown cells in Jupyter

1.106 Using Python as a Calculator

Click on the link below to read about using Python as a calculator:

- Using Python as a Calculator

Week 3

Key Concepts

- Import Python, NumPy and SciPy modules, and use them to compute basic statistics.
- Use logic and iteration to fill arrays with data, sum array elements and locate array elements with certain characteristics.
- Identify and use correct syntax and explain the purpose of built-in variable types int, float and list.

2.01 Becoming familiar with Python

Python is very strict on indentation. Code blocks are separated by indentation. For example, the following piece of code:

```
def my_function():
    print("Output")

if (x > 10):
    print(x)
```

Is not the same as this one:

```
def my_function():
    print("Output")

    if (x > 10):
        print(x)
```

Python has a set of familiar operators:

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Floor division
**	Exponentiation
%	Modulus

Python also has variables and types. Variables can be virtually any name, except for a few reserved words, such as `and`, `not`, `with`, `else`, `false`, and a few others.

We initialize a variable by assigning a value to it:

```
string_a = "foo"
int_x = 100
truthy = True
falsey = False
```

Regarding types, Python is dynamically typed which means a type is inferred at the time a value is assigned to a variable. Python is unable to implicitly coerce a type mutation, which means Python is a strongly typed language.

Python has a rich collection of built-in data structures. Among them are:

Lists Mutable sequence of comma separated values

Dictionaries Unordered, unique key-value pairs

Tuples Immutable sequence, comma separated

Sets Unordered collections, unique values

Functions are designed around reuse. The basic function definition looks like the one below:

```
def function_name(param1, param2, param3):
    if param1 > 10:
        print("Large value")

    return param2 + param3
```

Parameters to functions are optional.

Python also has the concept of Modules. They are a way to package functionality into a block of code that can be imported elsewhere. Python has a rich set of open source modules ready for use in different applications.

We can import parts of a module, or the entire module. It's convention to use the `from` keyword for the current namespace. A few examples:

```
import random
from fibo import fib
import numpy as np
```

When processing data, we can pull data from a database, scrape the web, or use files in local storage. Data can be stored either in binary format or some form of text. Data representation is important.

2.03 Built-in types in Python

Read through the following literature on built-in types in Python:

- Built-in types

2.05 Mapping operators to functions

Read through the following documentation.

- Python 10.3.1 Mapping operators to functions, Python language documentation 10.3. operator – Standard operators as functions.

2.104 Introduction to conditional logic

Python supports our regular `if`, `else`, and `elif` constructs. It also has the expected set of comparison operators `>`, `>=`, `<`, `<=`, `==`, and so on.

2.105 Functions and reuse

We define a function to make it easier to reuse code by changing a few parameters. Once defined, we can call the same function as many times as we want. To call a function, we input its name followed by parenthesis.

```
def my_function(eat):
    print(eat + " for dinner")

def double(x):
    return 2 * x

my_function("Carrots")
my_function("Cheese")
my_function("Bread")

print(double(2))
print(double(3))
print(double(4))
```

2.201 Conditions and logic

We combine logical statements with the connectives `or` and `and`.

```
a = 100
b = 50
c = 1000

if a > b or a > c:
    print("a is greater than b or c")
if a > b and a > c:
    print("a is greater than b and c")
```

2.202 More conditions

In Python, there are quite a few ways to deal with conditions. We might use certain conditions in cases where:

- we want to identify a time when a condition is met
- we want to iterate through a list of items
- we want to do something until something else happens, at which point we stop.

Click on the links below and read through the documentation that covers these conditional situations: Python 4.1, 4.2, 8.2 – 'if' 'for' 'while'.

- Python 4.1. if statements, Python language documentation – 4. Built-in types.
- Python 8.2. The while statement, Python language documentation – 8. Compound statements.
- Python 4.2. for statements, Python language documentation – 4. Built-in types.

Week 4

Key Concepts

- Import Python, NumPy and SciPy modules, and use them to compute basic statistics.
- Use logic and iteration to fill arrays with data, sum array elements and locate array elements with certain characteristics.
- Identify and use correct syntax and explain the purpose of built-in variable types int, float and list.

2.301 Introduction to lists

Lists are akin to *arrays* in other languages. It's a method for storing multiple values in a single variable. We can create a list by placing comma separate values inside square brackets `[]`. Like shown below:

```
1 car_emissions = [1.5, 1.26, 2.6]
2 print(car_emissions)
```

We can access individual elements of the list by indexing the variable. Note that indices start at 0, not 1. So the first element would be accessed like shown below:

```
1 print(car_emissions[0])
```

For other values, we just change the index. An interest peculiarity of python lists is that it works like a *ring*; that is to say that using negative indices allows us to walk backwards. In other words, the last element of a list is always at index `-1`:

```
1 print(car_emissions[-1])
```

We don't have to always index the list using a literal number. We can use a variable as well:

```
1 index = 1
2 print(car_emissions[index])
```

We can easily compute the length of a list using the `len()` method:

```
print(len(car_emissions))
```

Lists can also be indexed by ranges of numbers. If we want to print elements 0 through 2, we can use:

```
print(len(car_emissions[0:2]))
```

As we can see, the interval of the range is not inclusive of the final value. This means that the range `0:1` will only print the 0^{th} element.

2.302 Lists

Click on the link below to read through the documentation on lists:

- Python 4.6.4. Lists, Python language documentation – 4. Built-in types.

2.305 Loops and iteration

Iteration allows us to work through a series of objects. Python has different looping constructs to help us iterate through several objects. The first example is the `while` loop:

```
i = 1
while i <= 10:
    print(i)
    i += 1
```

We can also use a `for` loop which works rather like `while` loop above:

```
for i in range(10):
    print(i)
    i += 1
```

In Python, there are other ways to work through a list. For example:

```
teams = ["red", "blue", "green"]

for x in teams:
    print(x)
```

2.401 Libraries and dependencies

Python has a rich set of libraries available for our use. But how do we use them? We use the `import` keyword. To demonstrate, we import `NumPy` and `scipy`, two important libraries for data crunching.

After importing the library we can start using functions from the library, as shown below:

```
1 import numpy as np
2 from scipy import stats
3
4 x = np.array([1, 2, 3, 4, 5, 6])
5
6 np.mean(x)
7 stats.describe(x)
```

Week 5

Key Concepts

- Explain the difference between lists, dicts and NumPy arrays, and select appropriate data structures for particular examples of data.
- Write Python programs that can process and analyse text data in lists, dicts and NumPy arrays.
- Implement linguistic analysis algorithms that can compute word distributions, distances between distributions and a distance matrix.

3.01 Introduction to data

Python has a rich set of data types available for our use. To name a few:

String a sequence of characters

Boolean `True` or `False`

Float Decimal numbers

Integer Whole numbers

Complex Complex numbers

Python also has a rich set of built-in data structures which, typically, can hold different types of data::

Tuples Immutable arrays

Lists Mutable arrays

Dicts Hash tables

When it comes to encoding, Python supports many character encodings, including:

- ASCII (7-bits)
- Extended ASCII (8-bits)
- UTF-8 (8-bits)

- UTF-32 (32-bits)

We can use Python to work with many different file formats. Some formats are easier to work with than others. A few examples of file formats are:

- Flat File Formats

  CSV / TSV Tabular data, usually delimited by a special character or a *TAB*

- Non-flat File Formats

  Markup HTML, XML, Markdown, etc.

  Data Exchange Formats JSON

3.02 Data structures and data types

- McKinney, W. Python for data analysis: data wrangling with Pandas, NumPy, and IPython. (Sebastopol, CA: O'Reilly, 2017) 2nd edition, Chapter 3 Built-in Data Structures, Functions, and Files, pp.51–85.
- Natural Language Toolkit

3.04 Data handling

To open a file with python we use:

```
1 f = open("myfile.txt", "r")
```

There are a few different flags that can be passed to `open()`, here's a summary:

`r` Read

`a` Append (to the end of the file)

`w` Write. If the file doesn't exist, it will also create the file

`x` Create a file. If the file already exists, it produces an error

`t` Tells `open()` we're dealing with a text file

`b` Tells `open()` we're dealing with binary file

3.05 Working with text-based data

Let's assume we have a file `myfile.txt` with the content:

```
The quick brown fox jumped over the lazy dog
```

If we want to manipulate this data, we have to open the file for reading as before, then read the data into python, finally we close the file because it's not needed anymore:

```
f = open("myfile.txt", "r")
data = f.read()
f.close()
```

If we want to count the number of words in the file, we need to break our file contents into words and store the result as a list, like so:

```
words = data.split(" ")
```

To count the number of words, all we have to do now is compute the length of the returned list:

```
print(len(words))
```

3.06 counting words the verbose way

Continuing from last section, let's count the frequency of words in a given input:

```
counter = {}

for i in words:
    if i in counter:
        counter[i] += 1
    else:
        counter[i] = 1

print(counter)
```

3.07 Programming activity: reading files

Write a program that reads a text file into memory, then tokenises it on the space character.

- `https://docs.python.org/3/library/stdtypes.html#str.split`

The following NLTK documentation might be helpful here:

- nltk.tokenize package

Week 6

Key Concepts

- Write Python programs that can process and analyse text data in lists, dicts and NumPy arrays.
- Explain the difference between lists, dicts and NumPy arrays, and select appropriate data structures for particular examples of data.
- Implement linguistic analysis algorithms that can compute word distributions, distances between distributions and a distance matrix.

3.201 The dictionary data type

A dictionary is a kind of hash table, or associative array.

```
mydict = {
    "team": "red",
    "goals": "4",
    "conceded": "2"
}

x = mydict.get("goals")
```

Week 7

Key Concepts

- Write Python programs that can read and write files in CSV and JSON formats.
- Describe different types of data files and evaluate their appropriateness for storing different types of data.
- Process data for purpose.

4.01 Introduction

Open Data describes data that is publicly available and free from restrictions. We can use these data files to experiment and solve specific issues.

Some sources of Open Data are:

- `https://data.gov.uk`
- `https://europeandataportal.eu`
- `https://data.london.gov.uk`
- `https://undatacatalog.org/`
- `https://data.gov`
- `https://data.gov.in`
- `https://www.opendata.fi/en`
- `https://www.kaggle.com/datasets`

When using these data sources, it's common that we have to clean the data, i.e. remove superfluous, irrelevant, or incorrect data.

4.03 Exploring the CSV data format

CSV stands for *Comma-Separated Values*. This means there are specific characters that define a "field" in the dataset. Usually this character is a comma, hence the name, but it could be anything.

Most (all?) spreadsheet programs can import CSV files and display the data in a tabular format in rows and columns of the spreadsheet. This makes it easy to visualize the data in a consistent manner.

We can also use a regular text editor (or a great text editor, like Emacs!! **insert trollface here**) to explore the data. It allows us to understand conventions about the data format and how it's usually laid out.

4.04 Online documentation

- Python 14.1. csv – CSV file reading and writing, Python language documentation – 14. File formats
- Python pandas.read_csv, pandas 0.22.0 documentation – API reference
- Python pandas.DataFrame.to_csv, pandas 0.22.0 documentation – API reference – pandas.DataFrame
- Python API reference – Reindexing / Selection/ Label manipulation, pandas 0.22.0 documentation
- Python pandas.read_json, pandas 0.22.0 documentation – API reference

4.05 Getting started with Pandas and handling Data

- McKinney, W. Python for data analysis: data wrangling with Pandas, NumPy, and IPython. (Sebastopol, CA: O'Reilly, 2017) 2nd edition, **Chapter 5 Getting Started with pandas, pp.125–167** and **Chapter 6 Data Loading, Storage, and File Formats, pp.169–181**.

Week 8

Key Concepts

- Write Python programs that can read and write files in CSV and JSON formats.
- Describe different types of data files and evaluate their appropriateness for storing different types of data.
- Process data for purpose.

4.101 Working with CSV in Python

To work with CSV data in python, we must open the file and read its contents into python. Afterwards, we can parse the data into something we can work with.

```
import csv

list = []

with open("data.csv") as csvfile:
    reader = csv.reader(csvfile, delimiter = ",")
    for row in reader:
        list.append(float(row[6]))

list = list[1:]
print(sum(list) / len(list))
```

4.104 Data processing libraries used to improve efficiency

There are a plethora of python libraries to aid data processing, some of which provide very efficient data processing primitives for our use. Converting our previous example to use some of these libraries, we get:

```
import pandas as pd

df = pd.read_csv("data.csv")

# Basic statistics
```

```
print(df.describe())

# Sorting
print(df.sort_values(by='Value', ascending=False, na_position='first'))

# Filtering
print(df.filter(like='50', axis=0))
```

4.105 Introduction to the JSON format

The *JavaScript Object Notation*, or JSON, format is open standard data format that uses human-readable text to store and transfer data in key-value pair form.

A JSON file represents data in a Tree structure, which can make it slightly more complex to deal with. An example of JSON format follows:

```
{
  "staff":
  {
    "name": "Sean",
    "age": 31,
    "city": "London"
  }
}
```

4.106 Handling JSON files in Python

To manipulate JSON data in python, we need the `json` package.

```
import json

mestring = '{ "staff": { "name": "Sean", "age": 31, "city": "London" } }'

# medict = {
#     "staff":
#     {
#         "name": "Sean",
#         "surname": "McGrath",
#         "age": 31,
#         "city": "London"
#     }
# }

with open('me.json', 'r') as handle:
```

```
    medict = json.load(handle)
    print(medict)
    print(medict['name'])
```

We can also use `pandas`:

```
import json
import pandas as pd

with open('me.json', 'r') as handle:
    medict = json.load(handle)

    df = pd.DataFrame.from_dict(medict, columns=['name', 'surname'])
```

Week 9

Key Concepts

- Explain what HTTP is and how the client–server model makes it possible to access data on the internet.
- Implement an HTTP client in Python and use it to retrieve data from an HTTP server in HTML and JSON format.
- Read data from RESTFul web APIs.

5.01 Introduction to retrieving data from the web

HTTP is the protocol used for serving web applications. It's based on a Request-Response architecture in that the client sends a request and the server (maybe) offers a response.

The *lifetime* of a request can be simplified as below:

1. User types URL in browser
2. Check cache (Y/N)
3. DNS lookup of IP address
4. Browser initiates TCP connection
5. Browser sends HTTP request
6. Server handles the request
7. Browser receives HTTP response
8. Browser renders content

The protocol dictates this sort of handshake or conversation between the involved parties. This incurs extra overhead. In order to produce HTTP requests and parse HTTP responses we need to know how they are created. We need to know:

1. HTTP Method
2. Path
3. HTTP Version

4. Request Headers
5. Request Body

If we're making requests for web pages, those are written in HTML (HyperText Markup Language). It's a tag-based format that defines the layout of a webpage.

A browser will build a structure referred to as the DOM (Document Object Model). This is essentially a tree representation of the web page. Most elements in elements are composed of opening and closing tags such as `<body>` and `</body>`, but some are self-closing tags, such as `<br/>`.

Tags can also have attributes, which are name-value pair added within the tag.

5.02 Handling data on the web

- Python 'HTTP protocol client', Internet protocols and support
- Python 'Simple HTML and XHTML parser'
- Pyquery 'A jquery-like library for Python'
- Requests: HTTP for Humans

5.04 HTTP and transferring data via the web

A request will start with an HTTP verb. Usually a *POST* or a *GET*. That will follow with a URL. The server will send back a response containing a status code.

Something like so:

```
GET /~smcgr004/test.html HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: doc.gold.ac.uk
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

A response may look like this:

```
HTTP/1.1 200 OK
Date: Mon, 10 Nov 2020 12:00:00 GMT
Server: Apache/2.2.15 (CentOS)
Last-Modified: Wed, 10 Nov 2020 11:00:00 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed
```

5.05 Introduction to web scraping

We can issue HTTP programmatically with Python. Like shown in the example below:

```
from bs4 import BeatifulSoup
import requests
import json

def get_soup(URL, jar=None):
    request_headers = {
        "update_insecure_request": "1",
        "user-agent": "Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:47.0) Gecko/2010 0101 F
        "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/
        "accept-enconding": "gzip, deflate, br",
        "accept-language": "en-US,en;q=0.8"
    }

    if jar:
        r = requests.get(URL, cookies=jar, headers=request_headers)
    else:
        r = requests.get(URL, headers=request_headers)
        jar = requests.cookies.RequestsCookiesJar()

    print(r.url)
    data = r.text
    soup = BeatifulSoup(data, "html.parser")

    return soup, jar
```

We can use this sample method like shown below:

```
soup, jar = get_soup('http://doc.gold.ac.uk/~smcgr004/test.html')
print(jar)
```

Week 10

Key Concepts

- Explain what HTTP is and how the client–server model makes it possible to access data on the internet.
- Implement an HTTP client in Python and use it to retrieve data from an HTTP server in HTML and JSON format.
- Read data from RESTFul web APIs.

5.104 Scraping, APIs and libraries

- PyGitHub
- GitHub API
- McKinney, W. Python for data analysis: data wrangling with Pandas, NumPy, and IPython. (Sebastopol, CA: O'Reilly, 2017) 2nd edition, Chapter 6 Data Loading, Storage, and File Formats pp.181–185.

5.106 Considering alternative ways to parse text

- Python Regular Expressions

Week 11

Key Concepts

- Describe the structural elements of a relational database such as tables, columns and relations.
- Write simple SQL queries to read and write data from a relational database into Python using an SQL library.
- Select and use appropriate data structures to store data obtained from relational databases.

6.01 Introduction to databases

Relational Databases are based on the Relational Model proposed by E. F. Dodd in 1970. We use the Structured Query Language (SQL) to communicate with the database.

Relational Model is based on concepts from First-Order Predicate Logic and Set Theory.

Statements in First-Order Logic are composed of a Subject followed by a predicate. The predicate tells us something about the subject, e.g. *Sean is a lecturer*.

Relational Databases embed similar logic when it concerns **objects** and **properties**. Any given database can be reduced to a set of logical statements about their relationships.

We can think of tables in a database as Sets, therefore each **record** (or row) within a table is an **object** (or element) within that set.

The Relational Model organizes data into tables of rows and columns:

Name	Qty Drinks	Qty Food	Qty Snack	Delivery	Cost
Barry	0	2	1	0	7.50
Ameer	1	0	1	1	8.25
Lucy	0	1	0	0	2.25

Rows are records and columns are fields.

In general, we rely on the CRUD paradigm when dealing with a database.

Create Add a new record to a table

Read Read a record or a set of records from a table

Update Modify an existing record within a table

Delete Remove a record from a table

6.05 Some variants of SQL and general guidance for this topic

- PyGreSQL ‘connect - open a PostgreSQL connection’, Module functions and constants (2020).
- PyGreSQL ‘execute – execute a database operation’, Cursor – The cursor object (2020).
- PostgreSQL ‘7.1 Overview’, Documentation: PostgreSQL 10 – Chapter 7 (2020).
- PostgreSQL ‘7.2.2 The WHERE Clause’, Documentation: PostgreSQL 10 Chapter 7 (2020).