Functions & Input/Output

Database and Analytics Programming

Noel Cosgrave
National College of Ireland

Introduction

This third session of the Database & Analytics module will provide an overview of:

- Basic File Access
- Functions in-depth
- Lambda functions
- Map, Reduce and Filter functions
- Internet-based data sources
- Accessing structured files:
  - CSV format
  - XML format
  - JSON format

Basic File Access

Opening a file

The *open()* function returns a file object, otherwise known as a file handle, and takes two arguments *filename* and *mode*. The latter argument specifies how the file should be opened and takes one of the following values.

- **r** - the opened file can only be read from and not written to.
- **w** - the opened file can only be written to.
- **a** - the opened file can be written to with any new data being appended to the data already in the file.
- **r+** - the opened file can be both read from and written to.
- **x** - creates the specified file if it does not already exist and returns an error if the file exists.

Basic File Access

Opening a file

With any of these modes, the opened file will be in text mode, where data are read from and written to the file as text, often with a specific encoding. If no encoding is specified, the encoding uused is dependent upon the platform.

Note that default behaviour when reading in text is to convert platform-specific line endings... \n on Unix, \r\n on Windows or \r on legacy (pre-UNIX) MacOS systems) to just \n.

When writing to a file in text mode, the process is reversed, with occurrences of \n being converted back to platform-specific line endings.

Basic File Access

Opening a file

By appending **b** to the mode, a file can be opened in binary mode. This allows you to read in binary data such as that contained in photo, audio and video files.

When working with binary files, you should **exercise extreme caution** as it is your responsibility to make sure your code loads the data into the correct data structures and preserves those structures, and in turn, the file format, when writing the file back to disk.

Basic File Access

File object methods

The *write()* writes the contents of string to the file, returning the number of characters written. Other types of objects need to be converted, either to a string (in text mode) or a bytes object (in binary mode), before writing them to the file.

```
value = ('the answer', 42)
my_file.write(str(value))
```

The *tell()* method is used to discover the current file position in a file stream. The current file position can be altered with the *seek()* method.

```
f = open("myfile.txt", "r")
print(f.tell())
print(f.readline())
print(f.tell())
f.seek(f.tell()+2)
print(f.tell())
```

Functions

- As you will have learned in the bootcamp sessions, the purpose of function is to allow the re-use of code that is called at multiple points during a program. It is the most basic example of the **DRY (Do Not Repeat Yourself)** principle in action.
- A function in Python is created using the **def** keyword, followed by **zero or more parameters** surrounded by parentheses and ending in a colon character. When values are passed to the parameters when the function is invoked, they are termed the **arguments** of the function call.
- The body of the function consists of one ore more indented statements that are executed every time the function is invoked.

Functions

The return and yield keywords

- In addition to carrying out processing on the arguments passed to them, functions can return a value for further processing.
- This is achieved by adding the **return** keyword followed by an expression to be returned. This can appear anywhere in the function body, including inside *if*... *else*... constructs.
- Once a return statement is reached, the function will exit, returning the result to the context of the caller, which can be another function or a variable.
- If there is no return statement in the function code, the function exits when the control flow reaches the end of the function body.

Functions

The return and yield keywords

In the example below, the function returns *True* if the argument is a prime number and *False* otherwise.

```
def is_prime(number) :
    if number > 1:
        for factor in range(2,number) :
            if (number % factor) == 0:
                return False
            else:
                return True
    else :
        return False
print(is_prime(33999))
```

```
True
```

Functions

The return and yield keywords

So what if we want our functions to return multiple values, for instance outputting a value for each iteration through a *for* or *while* loop? This is where the **yield** keyword comes in. The example below shows the use of **yield** in a solution to the Pascal's triangle problem from last week.

```
def pascals_triangle(numrows):
    row = (1,)
    row_count = 1
    while row_count <= numrows :
        yield row
        previous_row_pairs = zip(row, row[1:])
        new_values = (x + y for x, y in previous_row_pairs)
        row = (1, *new_values, 1)
        row_count = row_count + 1

for row in pascals_triangle(10):
    print(row)
```

Functions

Default arguments

In cases where a function can have **sensible default values** for one or more of its parameters, it is possible to supply these defaults as part of the function definition, avoiding the need to pass them each time the function is invoked.

In the example below, a function is created with two parameters, one with a default argument. The function is then invoked passing in only one (non-default) argument and the passing in both arguments.

```
def greet_someone(person,greeting = 'Hello ') :
   return greeting + person
print(greet_someone('Hal'))
```

```
Hello Hal
```

```
print(greet_someone('Hal','Good morning '))
```

```
Good morning Hal
```

Functions

Mutable default arguments

When using default arguments, you will need to pay special attention when mutable objects are passed as defaults.

```
def add_list_to_list(new_elements, to=[]) :
    to.extend(new_elements)
    return to
```

You might expect this code to return a new list each time the *add_to_list* function is invoked. However, this is not what happens.

```
list1 = add_list_to_list([1,2,3])
print(list1)
```

```
[1, 2, 3]
```

Functions

Mutable default arguments

```
list2 = add_list_to_list([4,5,6])
print(list2)
```

```
[1, 2, 3, 4, 5, 6]
```

While it may seem somewhat counter-intuitive, instead of a new list being created each time the function is invoked, a list is created **only once** when the function is defined. This list is re-used (with its previous contents) each time the function is called.

Functions

Positional and keyword arguments

Arguments to a function can either be passed in the same order they are defined in the function header (so-called **positional arguments**) or they can be passed in any order, provided that the parameter names are supplied. These are known as **keyword arguments** .

```
import math
def distance(x1,y1,x2,y2) :
   return math.sqrt((x2-x1)**2+(y2-y1)**2)

print(distance(3,1,7,5))
```

```
5.656854249492381
```

```
print(distance(x1=3,x2=7,y1=1,y2=5))
```

```
5.656854249492381
```

Functions

Variadic functions

We have met the concept of **variadic functions** in previous lectures. To recap, variadic functions are those with an **indefinite arity**, in other words, they are functions that can take an indefinite number of arguments and are useful in situations where the number of parameters can not be determined a-priori.

So how can we create a variadic function in Python? The answer is to use an **arbitrary argument list**. By placing an asterisk in front of the last parameter, this parameter will be treated as a tuple containing all the remaining positional arguments.

```
def send_message(message, *recipients) :
    for recipient in recipients :
       print(message+" "+recipient)
```

Functions

Variadic functions

```
send_message('Hello','Hal','Frank')
```

```
Hello Hal
Hello Frank
```

It is also possible to pass arbitrary arguments using a dictionary or *key=value* format. This is achieved by putting two asterisks in front of the last parameter in the parameter list and then passing in the values as if they were keyword arguments.

```
def say_hello(**recipients) :
    for recipient,greeting in recipients.items() :
       print('{0} {1}'.format(greeting,recipient))
say_hello(Hal='Hello',Frank='Good morning')
```

```
Hello Hal
Good morning Frank
```

Functions

Variadic functions - a note of caution

Working with variadic functions, those with arbitrary argument lists, can be complex, difficult and error prone.

If all the arbitrary arguments are of the same fundamental data type, it is much better practice to supply them as a single iterable data type.

Functions

Lambda functions

- Inspired by the LISP language, a **lambda function** is an unnamed or **anonymous** function that can have any number of parameters, but is limited to one expression on a single line of code.
- A lambda function is an **expression** , and **not** a statement, and because of this it can be used in parts of your code where a full function definition would not be permitted, such as inside list comprehensions or the arguments of a function call.
- Unlike a regular function, a lambda expression will **return automatically** . Because it is limited to a single expression, it is less expressive than a full function definition. By its very nature, we can only squeeze so much logic into a lambda body.

Functions

Lambda functions

Lambda functions come into their own when used as an anonymous function inside a regular function.

For example, let's assume that you have a function definition that takes one argument, and the argument will be multiplied by an as yet unknown number.

```
def multiplier(n) :
    return lambda x : x * n
```

In true functional programming style, the function *multiplier* does not return a value, but instead returns a function.

Functions

Lambda functions

The function definition returned by the *multiplier* function can now be used, by supplying the relevant values as arguments and giving them an identifier.

```
doubler = multiplier(2)
tripler = multiplier(3)
```

The identifiers can now be used as multiplication functions.

```
print(doubler(10))
```

```
20
```

```
print(tripler(10))
```

```
30
```

Functions

The map function

The **map** function takes as its arguments the name of a function and one or more iterable object (a collection).

It executes the named function once for each item in the supplied collection(s) returning the results of those function calls as a **new collection** .

Functions

The map function

The loop-based code…

```
names = ['Mary', 'Isla', 'Sam']
for i in range(len(names)):
    names[i] = hash(names[i])
print(names)
```

```
[-8607373437925856648, 296553238086559766, -91719908828893630 56]
```

can be expressed as…

```
names = ['Mary', 'Isla', 'Sam']
print(list(map(hash,names)))
```

```
[-8607373437925856648, 296553238086559766, -91719908828893630 56]
```

Note that the returned *map* object must be cast as a list. This is because *map* uses **lazy evaluation** , where the returned values are only computed when they are accessed.

Functions

The reduce function

Like the **map** function, the **reduce** function also takes as arguments a function name and an iterable object. It returns a value, usually a scalar, which is created by combining the items in the collection.

```
from functools import reduce
print(reduce(lambda a,b : a if a > b else b,[42, 123, 42, 3, 50]))
```

```
123
```

Note that reduce has now been moved out of the main Python namespace and into the package *functools*. This is because there are better approaches to achieving the same functionality, such as list comprehensions.

The *reduce* function can also be combined with operator functions such as *add()*, *mul()* to achieve the same functionality as provided by lambda functions.

Functions

The filter function

The **filter** function also takes a function and an iterable object (such as a list) as arguments, and provides a simple and elegant way to filter elements in the iterable object.

The function passed in as an argument must return a **Boolean value** and will be applied to each element in the iterable. Those elements for which the function returns *True* are returned by the filter, those for which the function returns *False* are not.

```
import random
print(list(filter(lambda x: x % 2 == 0, range(10, 30))))
```

```
[10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

Working with CSV files

Reading CSV data

To work with files containing **Comma-Separated Variables (CSV)** we can use the standard Python module **csv**.

To read in CSV data, use the *csv.reader* object.

```
import csv
with open('data.csv', newline='') as csvfile:
    reader = csv.reader(csvfile)
    for row in reader:
        print(row)
csvfile.close()
```

Working with CSV files

Writing CSV data

To write to a CSV file, we use the *csv.writer* object. Then we either use *csvwriter.writerow()* to write a single row or *csvwriter.writerows()* to write an iterable as multiple rows.

```
import csv
csvData = [['Person', 'Age'], ['Peter', '22'],
['Jasmine', '21'], ['Sam', '24']]
with open('data.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerows(csvData)
csvfile.close()
```

Working with semi-structured data

The advent of RESTful Web Services (Fielding and Taylor 2000), the exchange of data using standard HTTP protocols increased.

While it is possible to represent internet-based resources in any manner, there are two defacto standard representations in common use: XML and JSON.

We will look at these two data formats in more detail in the following slides.

XML

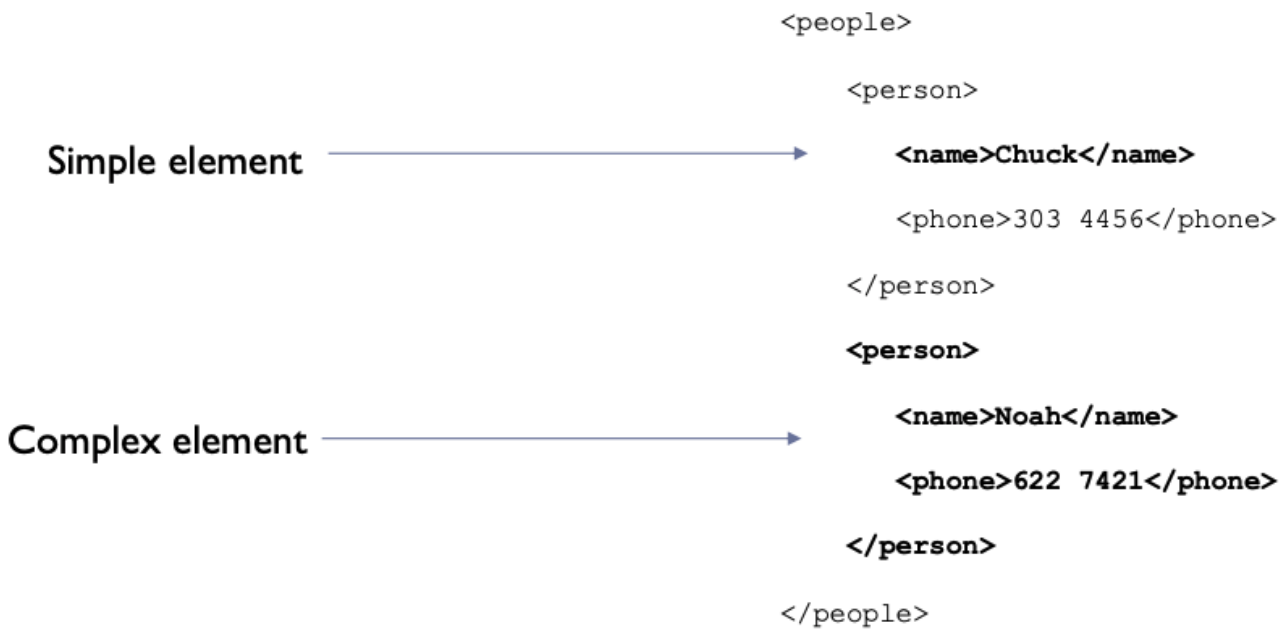
XML schemata

- A schema is a formal description an XML document.
- It express the constraints on the structure and content of the document.
- It is often used to specify a **contract** between two communicating systems, i.e. a particular application will only accept XML that conforms to the specified schema.
- If a document complies with the specification outlined in the schema, it is said to be “valid”.

XML

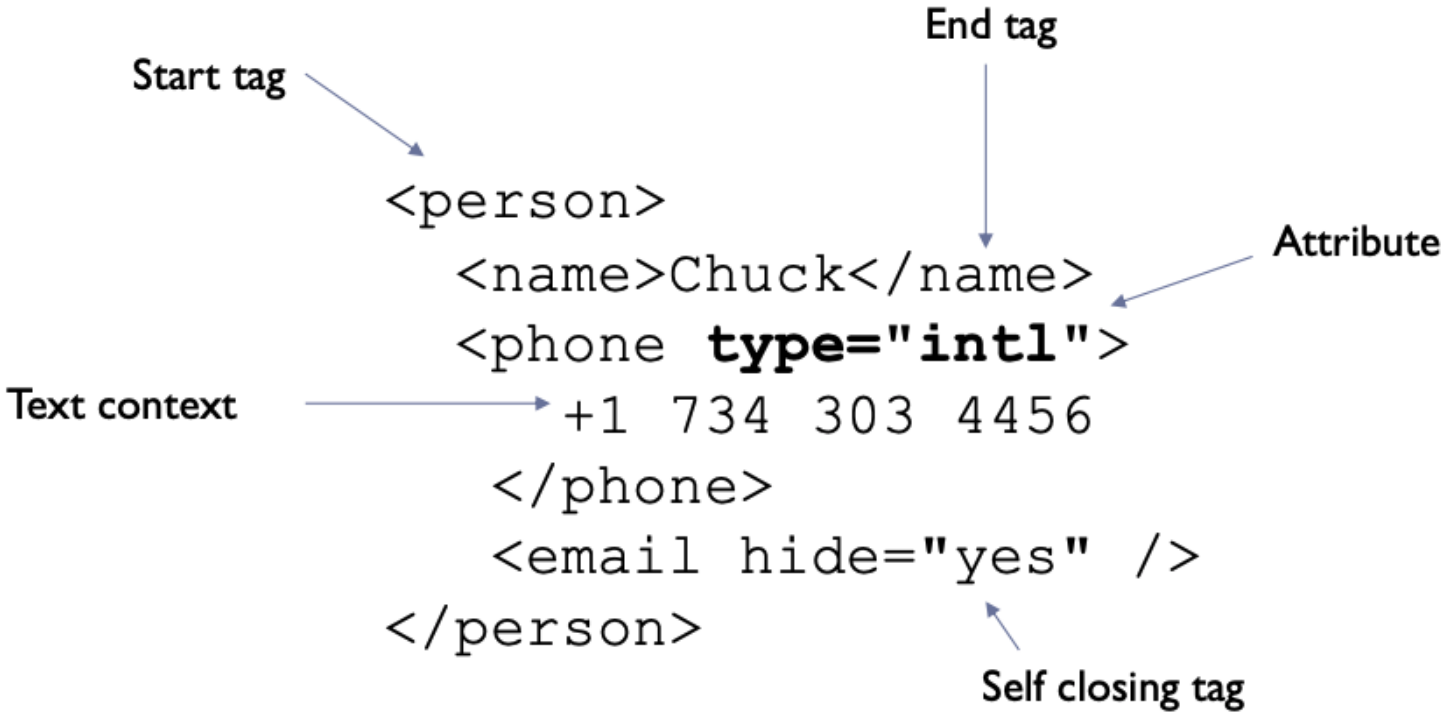
Document structure

Tags are used to denote the beginning and end of elements. Tags must be closed in the reverse of the order they were opened.



XML

Document structure



Attributes are key/value pairs in the opening tag. Line endings do not matter. White space is generally discarded on text elements. Indentation is only used to improve readability.

XML

Serialisation and Deserialisation

XML, along with JSON, is often used as **serialisation format** .

Serialisation is the process of creating a file or stream-based representation of an object in a program so that it can be transported to another system.

De-serialisation is the process of converting the file or stream-based representation back into an object within a program.

Serialisation/Deserialisation (frequently shortened to **SerDes**) allows for interaction between programs in a programming **language-independent** manner.

XML

Creating an XML tree in Python

```
import xml.etree.ElementTree as ET
data = '''<person>
<name>Chuck</name>
<phone type="intl">
+1 734 303 4456
</phone>
<email hide="yes"/>
</person>'''
tree = ET.fromstring(data)
print('Name:',tree.find('name').text)
```

```
Name: Chuck
```

```
print('Attr:',tree.find('email').get('hide'))
```

```
Attr: yes
```

XML

Navigating an XML tree in Python

```
import xml.etree.ElementTree as ET
input = '''<data><users>
<user x="2"><id>001</id><name>Chuck</name></user>
<user x="7"><id>009</id><name>Brett</name></user>
</users></data>'''
xmldata = ET.fromstring(input)
userlist = xmldata.findall('users/user')
for user in userlist:
    print('Name', user.find('name').text)
    print('Id', user.find('id').text)
    print('Attribute', user.get("x"))
```

```
Name Chuck
Id 001
Attribute 2
Name Brett
Id 009
Attribute 7
```

XML

Well-known XML Schemata

- Keyhole Markup Language (KML) is used to annotate geolocation data, and is used on sites such as Google Maps.
- Mathematical Markup Language (MathML) is a way of including mathematical equations in HTML pages and other documents. It produces similar results to the equation environment in LaTeX.
- RSS - Really Simple Syndication and Atom are both used to provide real-time news feeds.
- The RDF - Resource Description Framework is used to describe and model the information available from web resources.

XML

Well-known XML Schemata

- Scalable Vector Graphics (SVG) is used to define portable vector-based images and animations that can be directly embedded in web pages.
- Closer to our own particular field, the Data Documentation Initiative (DDI) is an standard for the production of machine-readable descriptions of statistical data files, surveys and questionnaires, as well as any processing/transformation carried out on these files.
- The Statistical Data and Metadata eXchange (SDMX) is an initiative to achieve standardisation of the exchange of international and supra-national bodies.

JavaScript Object Notation

JavaScript Object Notation(JSON) is a lightweight format for data interchange.

- It is easy for humans to read and write.
- It is easy for machines to parse and generate.
- It is the data format used by the JavaScript Programming Language

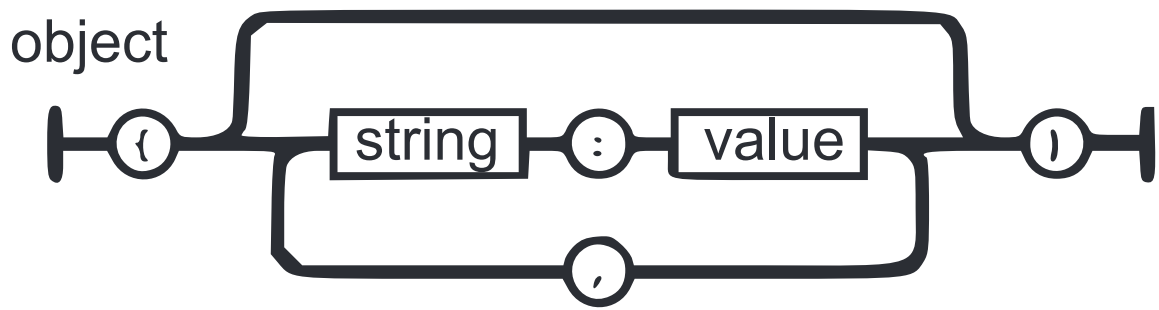
JSON is composed of two fundamental structures:

- A **collection of name/value pairs** . In various languages, this is realised as an object, a record, a struct, a hash table, a keyed list or an associative array (map).
- An **ordered list of values** . In most programming languages, this is realised as an array, vector, list or sequence.

JavaScript Object Notation

Objects

An **object** is an unordered set of name/value pairs, enclosed in curly braces. Each name and value are separated by a colon character and the name/value pairs are separated by a comma. Keys must be presented as strings, while values can be a valid JSON data type.



This structure is similar to that of dictionaries in Python.

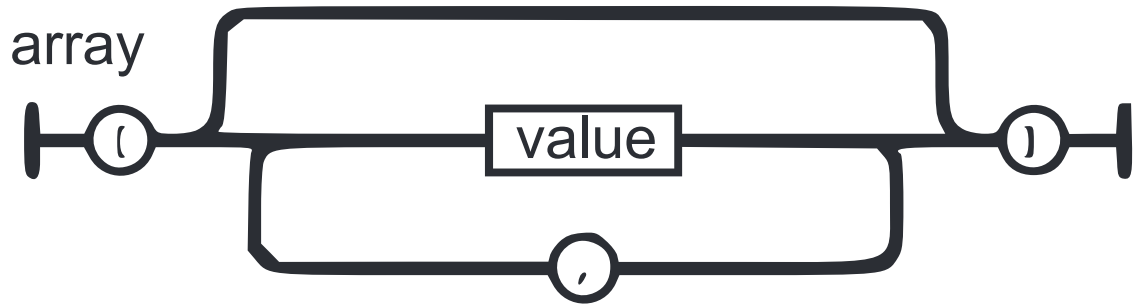
```
{ 'first' : 'John', 'last' : 'Donne', 'Phone number' : 7773452 }
```

JavaScript Object Notation

Arrays

An array is an ordered list of values. Each value is separated by a comma and the entire list is enclosed in square brackets.

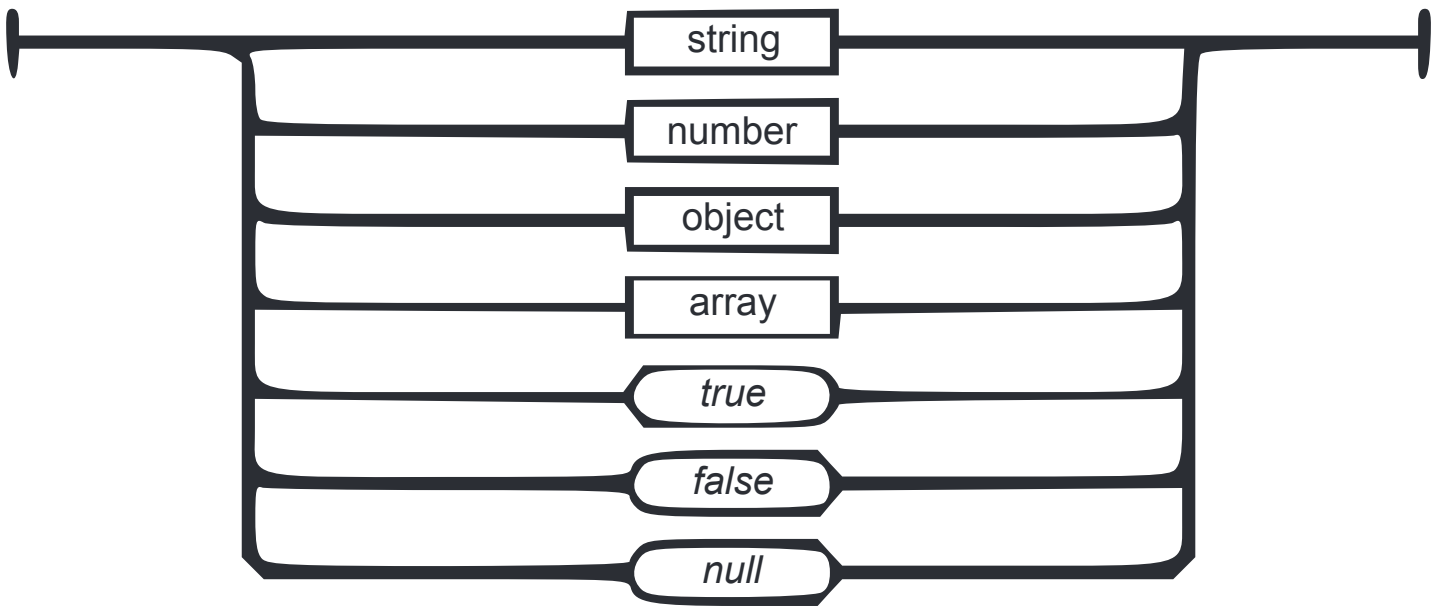
Array values must be of one of the valid JSON data types (see the next slide on JSON values).



JavaScript Object Notation

Values

JSON values can be one of the following data types: a string; a number; an object; an array; a boolean or null.



Consuming REST APIs

The code below uses *urllib* to fetch a list of public APIs from an open API. The JSON response is parsed to extract just the API names.

```
import urllib.request
import json
import ssl
apis = []
ssl._create_default_https_context = ssl._create_unverified_context
url = 'http://api.publicapis.org/entries'
with urllib.request.urlopen(url) as response :
  parsed_json = json.loads(response.read())
  for item in parsed_json['entries'] :
     apis.append(item['API'])
print(apis[30:34])
```

```
['Harvard Art Museums', 'Iconfinder', 'Icons8', 'Noun Project']
```

Note that this API does not require authentication. Any API that requires authentication will need additional code to handle that aspect of the communication.

Loading web resources into Python

The urllib module

The Python module **urllib** can be used to interact with resources accessible through the internet.

```
import urllib.request
response = urllib.request.urlopen('http://python.org/')
html = response.read()
html = str(html)
```

Polite crawlers should identify themselves using the *User-Agent* http header.

```
request = urllib.request.Request('http://python.org/')
request.add_header("User-Agent", "My Python Crawler")
opener = urllib.request.build_opener()
response = opener.open(request) # make the http request
```

Loading web resources into Python

The urllib module

We can retrieve the status for the request and various metadata for the page, such as the name of the server software and the content type of the returned resource.

```
import urllib.request
with urllib.request.urlopen('http://python.org/') as response:
    # print(response.headers)
    print('The response code is {}'.format(response.status))
    print('The page is served by {}'.format(response.headers['Server']))
    print('The has the content type {}'.format(response.headers['Content-Type']))
```

```
The response code is 200
The page is served by nginx
The has the content type text/html; charset=utf-8
```

Web scraping

Web scraping is a term for when a program acts as a browser and retrieves web pages, parses them, extracts the relevant parts and then follows the links in the pages, once again retrieving, parsing and extracting information.

Search engines are programs that do this to build up an index of the contents of the web. This is termed “spidering the web” or “web crawling”.

So why would we want to scrape data?

- One legitimate use for this would be to recover your data from a system that does not provide an export function.
- To obtain social media data - such as “who is connected to who?”

Web scraping

Legalities

The legalities of web scraping are somewhat vague. Very few websites explicitly permit web scraping - Wikipedia is an example. Some sites explicitly state that it is not permitted. Frequently, you will find that websites don't make any statement on the matter one way or the other.

Before attempting to scrape a website, you should check the site's terms and conditions page to see if there is any prohibition of web scraping. Note that the activity of web scraping may be described in other ways.

If there are prohibitions, you should respect them. If there are not, then you need to exercise your own judgment.

Web scraping

Legalities

You should always be aware that the activity of web scraping consumes server resources on the web host.

If you are automatically fetching a single page on one occasion one page once, this is probably not going to cause you problems.

However, if you are scraping tens of thousands of pages every hour, your activities will be noticed.

To avoid placing undue burden on a site you are scraping, you should limit the number of URLs you fetch and you should cache any content you retrieve to avoid unnecessarily re-fetching it.

Web scraping

Example using Beautiful Soup to extract data from Wikipedia

```
import urllib.request
from bs4 import BeautifulSoup
import pandas as pd
url = 'https://en.wikipedia.org/wiki/World_Happiness_Report'
web_page = urllib.request.urlopen(url)
parsed_data = BeautifulSoup(web_page, "html.parser")
table = parsed_data.find_all('table')[4] # the fifth table
rank, country, score = ([],[],[])
for row in table.findChildren('tr'):
    cells=row.findChildren('td')
    if len(cells)==9: # check there are 9 columns
        rank.append(cells[0].text.strip())
        country.append(cells[1].text.strip())
        score.append(cells[2].text.strip())
country_df=pd.DataFrame(rank,columns=['Rank'])
country_df['Country']=country
country_df['Score']=score
print(country_df)
```

Web scraping

Example using Beautiful Soup to extract data from Wikipedia

```
    Rank                    Country  Score
0      1                    Finland  7.769
1      2                    Denmark  7.600
2      3                     Norway  7.554
3      4                    Iceland  7.494
4      5                Netherlands  7.488
..   ...                        ...    ...
151  152                     Rwanda  3.334
152  153                   Tanzania  3.231
153  154                Afghanistan  3.203
154  155  Central African Republic  3.083
155  156                South Sudan  2.853

[156 rows x 3 columns]
```

Summary

In this session of the Database & Analytics module we looked at:

- Basic File Access
- Functions in-depth
- Lambda functions
- Map, Reduce and Filter functions
- Internet-based data sources
- Accessing structured files:
  - CSV format
  - XML format
  - JSON format

Bibliography

Fielding, Roy Thomas, and Richard N. Taylor. 2000. “Architectural Styles and the Design of Network-Based Software Architectures.” PhD thesis, University of California, Irvine.