B2 Topic 4: Reading and Writing to Files

So, how do you work with files? Regardless of the language or file type, there are three key steps you need to follow.

1. Open the file.
2. Read or write the data.
3. Close the file.

Make sure you remember to do all three. Until you open the file, you won't be able to read it. But if you forget to close the file, you may leave it in a state where it can't be read by others (including yourself if you try to open it again) or where some of the data may be lost. So do not forget that last important step of closing your file when your program no longer needs to access it.

For simplicity, the plain text file, **file0.txt (https://myuni.adelaide.edu.au/courses/86136/files/12213404?wrap=1) (https://myuni.adelaide.edu.au/courses/86136/files/12213404/download?download_frd=1)** , will be used for all examples on this page. Files with the suffix or *file extension* ".txt" are text files without any specific formatting. You will learn how to work with formatted data on the next page.

Opening a File

In Python, files are opened using the built-in `open()` function. The `open()` function accepts multiple parameters for **options to open a file (https://docs.python.org/3/library/functions.html?highlight=open#open)** , but at this stage we only need the first two. The call to `open()` returns a *file object* that you can then use to further manipulate the opened file.

The first, and only mandatory, parameter for `open()` is the file name. For example, to open `file0.txt` you would use `open('file0.txt')`.

The function above works only if your Python code (or Jupyter workbook) and your file are in the same directory. If they are in different directories, you will need to provide the 'full path' to the file. For now, just be sure you have saved your file in the same folder where you are saving your Jupyter Notebook.

The second parameter is the file's *mode*. This is an optional string that specifies the mode the file is opened with. The mode controls Python's read and write permissions for opened file. If you do not set a mode, the default mode `'r'` is used to allow read-only access.

Table 1. Summary of the behaviour of the four most common modes.

Description	mode	open for reading?	open for writing?	Behaviour

read-only	`'r'`	Yes	No	Fails if the file does not exist.
write-only	`'w'`	No	Yes	If the file already exists, the file's contents is truncated (cleared), otherwise it will create an empty file.
exclusive creation	`'x'`	No	Yes	Creates an empty file; fails if the file already exists.
append-only	`'a'`	No	Yes	If the file already exists, all writes are appended to the end of it, otherwise it will create an empty file.

A `'+'` can be appended to each of the above modes to extend their permissions to both read and write. For example `open('file0.txt', 'r+')` will extend the 'read-only' functionality to 'read-and-write', which will also allow you to write new text to the file using `write()`.

Warning!

On first consideration `'r+'` and `'a+'` might seem identical, however they are not.

- For reading, `'r+'` will begin reading from the start of the file, while `'a+'` will start from the end. In both cases you can use the `seek()` method (explained later) to jump to an arbitrary byte to begin reading from there.
- For writing, `'r+'` will begin to write at the start of the file, or any arbitrary byte by using the `seek()`, while `'a+'` will only ever append text to the end of the file.

You should experiment with both of these to build an understanding of how and when to use each of them.

Let's look at some examples.

In this first example we open the file using the most straightforward approach. Here `f` is a variable that represents the file inside your code. Think of it as a handle that lets Python hold onto the actual file on your machine. If you did not use a variable to hold the opened file, you would not be able to work with it. Also, as we did not set the mode, the file will default to read-only mode.

In [1]:
```python
f = open('file0.txt')
# do something
```

```
f.close()
```

In this second example, we open the file the same way, but this time we specify the mode to write-only by using `'w'`. As a result, a blank file named file0.txt will be created, ready to be written to (if file0.txt already exists the file contents will be truncated).

In [2]:
```
f = open('file0.txt', 'w')
# do something
f.close()
```

In this final example, we used a different approach of opening the file, using **with-as**. When we open a file this way, we format the code similar to an if condition or a loop. In this example we have set the mode to read-and-write using `'r+'`. Also, notice that we did not manually close the file using close() in this example. This is because when we use with-as, the file is automatically closed when the block completes or an exception is thrown. The latter makes this a safer approach to open files.

In [3]:
```
with open('file0.txt', 'r+') as f:
    # do something
```

Remember, if you did not use the with-as approach, you need to manually close the file using `close()`.

Best Practice: Open files using "with-as"

It is good practice to use the `with` (https://docs.python.org/3/reference/compound_stmts.html#with) keyword when dealing with file objects. The advantage is that the file is properly closed after its after it is used, even if an exception is raised at some point. An exception is an error condition raised by the Python interpreter when an error occurs at runtime and execution of the code halts - potentially leaving the file in an unusable state.

Quick Check

When opening a file, the value returned from the function invocation is:

- ○ an value that indicates if the file was opened or not.
- ○ a file object.
- ○ no value is returned from open() but an exception can be raised if a problem occurs.

Check Answer

Reading from a File

Reading from a file is the process of moving data from a file into your code. Reading a file does not change its contents.

There are multiple ways to read a file in Python depending on your needs.

Firstly, you can read either the entire file or a specific number of characters using the `read()` method.

In [4]:
```python
myFile = open('file0.txt')
print(myFile.read())
myFile.close()
```

Out[4]:
```
hello world
nice to meet you
```

In [5]:
```python
myFile = open('file0.txt')
print(myFile.read(5))
myFile.close()
```

Out[5]:
```
hello
```

Secondly, you can read the file line-by-line using the `readline()` method. Like, `read()` you can optionally choose the number of characters this function reads up to. Regardless of the size is specified, the read will stop once the line ends or the number of bytes specified in the call to `read()` or `readline()` has been reached. The "**pydocs (https://docs.python.org/3.10/library/index.html)** " (information on the Python standard library) are useful to find information on parameters passed to the built-in functions. For example, you can find more about `readline()` **here (https://docs.python.org/3.10/library/io.html#io.IOBase.readline)** .

In [6]:
```python
myFile = open('file0.txt')
print(myFile.readline())
print(myFile.readline(4))
myFile.close()
```

Out[6]:
```
hello world

nice
```

You may have noticed that calling `readline()` twice, caused the second line to be read. This is because as you read a file it tracks what has been read so far, and the next time the program reads, it begins from the first character that has not yet been read. If you want to return to a place that you have already read you can use `seek()` method and provide it with the byte (or character) you want to return to. As such `seek(0)` returns to the top of the file.

Why is there a blank line between "world" and "nice" in Out[3]?

`myFile.readline()` is returning a string that contains the first line of the file. At the end of that line is a newline character to signify the line has ended. This character will be included in the string generated by `readline()`. The `print()` function also places a newline after it has run. Therefore, when `print(myFile.readline())` is run, there are two newlines being output to the terminal, resulting in the blank line being shown.

Writing to a File

Writing to a file is achieved with the `write()` function by giving it a string. All writes are appended to the end of a file. Let's look at some examples, by writing to file0.txt from earlier while the file is opened using different modes.

In [7]:

```
myFile = open('file0.txt', 'w')
myFile.write('goodbye world')
myFile.close()
```

file0.txt:

```
goodbye world
```

In [8]:

```
myFile = open('file0.txt', 'a')
myFile.write('goodbye world')
myFile.close()
```

file0.txt:

```
hello world
nice to meet yougoodbye world
```

In [9]:

```
myFile = open('file0.txt', 'x')
myFile.write('goodbye world')
myFile.close()
```

Out[9]:

```
---------------------------------------------------------------------------
FileExistsError                           Traceback (most recent call last)
/tmp/ipykernel_4278/530682198.py in
----> 1 myFile = open('file0.txt', 'x')
      2 myFile.write('goodbye world')
      3 myFile.close()

FileExistsError: [Errno 17] File exists: 'file0.txt'
```

Your turn for some practise: Reading & writing to Files

Now it's time to practice what you have covered. In exercise you will open, read, and write to a files.

1. Read the contents of `file1.txt` and copy the contents of that file into a string variable.
2. Edit the string using string methods so that the string is fully upper case. Create a new file named `file2.txt` and write the string to that file.
3. Edit the string again, this time both converting it to title case. Write to `file1.txt`, by appending the string to the end of the file.
4. Close the file.

Readings

Python documentation for `open():`

> **https://docs.python.org/3/library/functions.html#open (https://docs.python.org/3/library/functions.html#open)**

The essential Python documentation on reading and writing files:

> **https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files (https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files)**

A good overview of file handling in Python:

> Mohit, & Das, B. N. (2017). Ch. 9. ***Learn Python in 7 days*** ***(https://ebookcentral.proquest.com/lib/adelaide/reader.action?docID=4866343&ppg=189)*** . Packt Publishing, Limited.

You can also access all the readings for this course from the 'Course Readings' menu item on the left of your screen.

Next Steps

So far you have covered how to represent files in Python as well as how to read and write to them.

Next, you will learn to use structured data in your Python programs.

(h (h (h (h (h (h (h (h (h (h

B2 Topic 4: Handling Structured Data (CSV, JSON)

So far this week, we have only discussed reading and writing to TXT files. This type of text file is common and useful to many programs, but when you want to work with data, you usually want to work with formatted data (such as those seen in spreadsheets).

For the balance of the week, we will discuss how to easily work with some of the most common formatted text files: CSV, JSON, and XML.

Working with CSV files

CSV (comma-separated values) are a common form of text file. As its name implies, data is stored as values with commas separating them. The comma is called a "delimiter" and there are many cousins of CSV files that use, for example, tab characters as a delimiter. CSV files are often one of the options you can choose when saving a spreadsheet, such as Microsoft Excel or Google Sheets.

Here is an example of what a CSV file might look like, you can download it **here (https://myuni.adelaide.edu.au/courses/86136/files/12213485?wrap=1)** **(https://myuni.adelaide.edu.au/courses/86136/files/12213485/download?download_frd=1)** .

```
name,age,postcode
John,52,5002
Ye,18,3005
Siobhan,34,2356
```

This data represents the name, age, and postcode of three people. Each line has the data for a different person, with the first line being the header information. Headers are not mandatory in CSV files (and indeed there is no recognised standard that governs formatting in CSV files) but, if you have the option, you should always include a header row in your files to make them self documenting.

Reading a CSV file

You could use the `readline()` method to read the data and then use string functions to split text around the ',' characters and put the values into a list or dictionary, but there is a much easier way!

Python has a CSV module that exists to make reading from CSV files easier. The CSV module defines **readers** which allow you to read each row. There are two options:

1. `reader`: reads a row and returns the values as a *list* of strings
2. `DictReader`: reads each row and return the values as an ***ordered dictionary (https://docs.python.org/3/library/collections.html#collections.OrderedDict)*** where the first row

is used as the keys.

For now, we will be using the first option to generate lists of strings. Here is an example of how to the CSV's reader method.

In [1]:
```
import csv
dataFile =  open('customers.csv', 'r')

reader = csv.reader(dataFile)

for row in reader:
    print(row)

dataFile.close()
```

Out[1]:
```
['name', 'age', 'postcode']
['John', '52', '5002']
['Ye', '18', '3005']
['Siobhan', '34', '2356']
```

In the above code, we begin by importing the CSV module and opening the file `customers.csv` as read-only. Next we used the `reader` method to create a variable (specifically an object) of the same name. After that we iterate over the entire file using a for-loop. Each time the loop is run, the next line of the file is being assigned to the row variable (a list of strings), then the contents of that list is printed using the print function. When there are no more lines to be read, the loop ends. Afterwards, the file is closed.

Note

Sometimes when you attempt to work with a CSV or another text file, you might notice some characters you are not expecting. For example your output might have looked like this:

```
['\ufeffname', 'age', 'postcode']
['John', '52', '5002']
['Ye', '18', '3005']
['Siobhan', '34', '2356']
```

Here, `\ufeff` has appeared before the first header field. This character is a special indicator of how the file has been encoded. You can remove this by specifying the encoding when you open the file, but unfortunately there isn't any way of easily detecting how the file has been encoded. Whenever you are reading files, check that your data does not contain extra encoding information. In general, this will appear at the start of the file, so you'll pick it up if you look at the first data item in the first row. If you have some unusual values or errors in your data results, you should consider errors in reading the data in as a possible cause.

A quick online search of `\ufeff` reveals that this is the start of a 'utf-8-sig' encoded file. utf-8 is common encoding from Microsoft products. You can add the encoding parameter to the open call to read in this encoding and remove the `\ufeff` from the data like so:

`open('customers.csv', 'r', encoding='utf-8-sig')`

Writing to a CSV file

Just like with reading, the CSV module offers two options to make writing easier.

1. `writer`: writes comma delimited values to a file
2. `DictWriter`: writes the keys as comma delimited header values and the dictionary values as comma delimited row values

Again, for now, we will be using the first option.

In [2]:
```
import csv

data = [['name','age','postcode'],
    ['John', 52, 5002],
    ['Ye', 18, 3005],
    ['Siobhan', 34, 2356]]
extra_data = ['Jose', 44, 5125]

with open('output.csv', 'w') as file:
    writer = csv.writer(file)
    writer.writerows(data)
    writer.writerow(extra_data)
```

In the above program, we create a new file, output using open in write mode, then we set-up the writer. After that we used the `writerows` method to write the first set of data to the file. Then we add 1 more row using the `writerow` method to add 1 more row of data. The reason we used `writerows` (plural) the first time and `writerow` (singular) the second is because the first time we are adding a list of lists which represents multiple rows of data.

Your turn for some practise: Working with CSV files

In this exercise, you will practice using CSV files from within your Python programs.

1. Using a spreadsheet program, such as Microsoft Excel, create some data (e.g. a list of names and ages) and save your file as a CSV file with the name 'data.csv'. The first row should be your headers, all the other rows should be your data. You can do this by following: *menu* > *File* > *Save As*. In the bottom of the save as screen, you will see 'File Format'. Select Text (.csv) to save your file in CSV format. If you have any difficulty creating your own file, you can download **this file (https://myuni.adelaide.edu.au/courses/86136/files/12213486?wrap=1)** **(https://myuni.adelaide.edu.au/courses/86136/files/12213486/download?download_frd=1)** instead.
    - You'll likely get a warning that you will lose some functionality. This is because calculations, scripts, and so on can't be saved in CSV format, so you will only be saving the actual values. That is fine, it's the data you're after!
2. Open your CSV file in Python for appending.
3. Create a list that represents another row of data.
4. Write that data to data.csv.

If you do not possess a license for the Microsoft Office suite, do not stress. As a student, you can access them through the university. Firstly, you can install the suite on your own machine **following these instructions (https://www.adelaide.edu.au/technology/your-services/software/software-for-students#microsoft-office-365)** or run it in your browser or machine using **ADAPT (https://www.adelaide.edu.au/technology/your-tools/academic/adapt)**. Alternatively, you could use Google Sheets instead.

Working with JSON files

JSON (JavaScript Object Notation) is another form of structured text file. Despite its name, JSON is useful for more than just JavaScript programming as a lightweight data-interchange format. The JSON format is codified by an **ECMA standard (https://www.ecma-international.org/publications-and-standards/standards/ecma-404/)**.

The JSON format is built on two structures:

- A collection of name/value pairs. In Python, this is realized as a dictionary.
- An ordered list of values. In Python, this is realized as a list.

A JSON object is an unordered set of name/value pairs grouped by *curly braces*: {}.

Here is the same data as above, this time in JSON format. You can download it **here (https://myuni.adelaide.edu.au/courses/86136/files/12213493?wrap=1)** **(https://myuni.adelaide.edu.au/courses/86136/files/12213493/download?download_frd=1)**.

```
{
  data: [
          {name:"John", age:52, postcode:5002},
          {name:"Ye", age:18, postcode:3005},
          {name:"Siobhan", age:34, postcode:2356}
        ]
}
```

In this example, "data", "name", "age" and "postcode" are all examples of names in name/value pairs - analogous to a key in a Python dictionary.

Working with JSON in Python is made significantly easier through use of the **JSON module (https://docs.python.org/3/library/json.html?highlight=json)**. This module handles the task of writing Python data to file (called "serialization") and reading from a JSON file or string into a Python data structure (called "deserialisation").

During the process of serialisation and deserialisation, simple Python objects are translated to and from JSON according to a **fairly intuitive conversion (https://docs.python.org/3/library/json.html?highlight=json#py-to-json-table)**:

Python	JSON
dict	object

list, tuple	array
str	string
int, long, float	number
True	true
False	false
None	null

Deserialisation - Reading JSON Data

The JSON module includes a function `loads()`. To use it, `json.loads()` is given a string in JSON data format and loads the data into dictionaries (for name/value pairs) and lists (for unlabelled sequences). An example of a name/value pair is `name: "John"`.

The example below shows how the data can be converted from a JSON string into more usable dictionaries.

In [3]:
```python
import json

jstr = '''{"data":[ {"name":"John","age":52,"postcode":5002},
                    {"name":"Ye","age":18,"postcode":3005},
                    {"name":"Siobhan","age":34,"postcode":2356}
                   ]
           }'''

# Load raw data into dictionary
jdata = json.loads(jstr)
print(jdata)

print()
data = jdata["data"]
print(data)

print()
for i in range(len(data)):
    print(data[i])
```

Out[3]:
```
{'data': [{'name': 'John', 'age': 52, 'postcode': 5002}, {'name': 'Ye', 'age': 18, 'postco
de': 3005}, {'name': 'Siobhan', 'age': 34, 'postcode': 2356}]}

[{'name': 'John', 'age': 52, 'postcode': 5002}, {'name': 'Ye', 'age': 18, 'postcode': 300
5}, {'name': 'Siobhan', 'age': 34, 'postcode': 2356}]

{'name': 'John', 'age': 52, 'postcode': 5002}
{'name': 'Ye', 'age': 18, 'postcode': 3005}
{'name': 'Siobhan', 'age': 34, 'postcode': 2356}
```

That code might seem a bit overwhelming, so let's break it down.

1. We began with a JSON string (a string in JSON format). We then used `loads()` create a dictionary called `jdata` with one key called 'data'. The value of data is a list of three

dictionaries. Each of those three dictionaries has three key:value pairs. This is the first block of text that was printed.
2. Then, we took the value of 'data' from `jdata` and stored it into a new variable called `data`. Now `data` is a list of dictionaries. This is what was printed in the second block.
3. Finally, we print each dictionary in the list using a for-loop.

Reading a JSON file

To read a JSON file:

1. Open the file. The file does not need to be a JSON file, but the entire contents does need to be in JSON format.
2. Read from the file using `read()`. This will return the entire file contents as a string in JSON format.
3. Call the `json.loads()` on the string generated in Step 2.

To run the code below, you will need to JSON file from the top of this section. Download it save it in the folder with your Jupyter Notebook. Aside from loading the data from a file, the code is otherwise identical to **In [3]**.

In [4]:
```
import json

with open('customers.json', encoding='utf-8-sig') as dataFile:
    # Load data into dictionary
    jstr = dataFile.read()
    jdata=json.loads(jstr)

print(jdata)

print()
data = jdata["data"]
print(data)

print()
for i in range(len(data)):
    print(data[i])
```

Out[4]:
```
{'data': [{'name': 'John', 'age': 52, 'postcode': 5002}, {'name': 'Ye', 'age': 18, 'postco
de': 3005}, {'name': 'Siobhan', 'age': 34, 'postcode': 2356}]}

[{'name': 'John', 'age': 52, 'postcode': 5002}, {'name': 'Ye', 'age': 18, 'postcode': 300
5}, {'name': 'Siobhan', 'age': 34, 'postcode': 2356}]

{'name': 'John', 'age': 52, 'postcode': 5002}
{'name': 'Ye', 'age': 18, 'postcode': 3005}
{'name': 'Siobhan', 'age': 34, 'postcode': 2356}
```

When working with JSON, remember that name:value pairs in JSON will become dictionary elements, and comma separated values within [ ] will become part of a list.

Serialisation - Writing JSON Data

JSON is even easier to than CSV. Simply open the file for writing `'w'` or appending `'a'` and then call `json.dump(data, file)`.

In [5]:
```python
import json

data = [{'name':'John', 'age':52, 'postcode':5002},
        {'name':'Ye','age':18, 'postcode':3005},
        {'name':'Siobhan', 'age':34, 'postcode':2356}
       ]

with open('newFile.json','w') as file:
    json.dump(data,file)
```

Your turn for some practise: Working with JSON files

In this exercise, you will practice using JSON files from within your Python programs.

1. Create JSON data in a text editor and save it to a file named either data.json or data.txt.
    - If you are having difficulties, download the file data.json and use that instead.
2. Open your JSON formatted file in Python.
3. Load the data into a dictionary.
4. Choose a key (e.g. 'name'). Print the value as associated with that key for each dictionary.
    - For example, if you are working with the provided data, you might printing each dictionary's character's actor would print `John Barrowman`.

Hint #1 - Printing 'for each' entry

This is done most easily using a **for-loop**. See the for-loop used at the end of **in [4]** if you have not had much exposure to for-loops.

Hint #2 - Printing a 'specific attribute'

Dictionaries have a method called `get()`, which when provided with a key name, returns the value associated with that key.

For example, if we wanted to only list the names of people in **in [4]**, we would change the line inside the for-loop from `print(data[i])` to `print(data[i].get('name'))`.

Resources

- Python docs on the class [OrderedDict (https://docs.python.org/3/library/collections.html#collections.OrderedDict)](https://docs.python.org/3/library/collections.html#collections.OrderedDict).

B2 Topic 4: Handling Structured Data (XML)

So far this week, we have discussed reading and writing to TXT, CSV and JSON files. These file types are common in general Python programming and data science. Another common format for storing and transmitting data is the eXtensible Markup Language (XML). XML rocketed to fame at the beginning of the century and was quickly adopted by many organisations following standardisation under the aegis of the World Wide Web Consortium.

XML aims to achieve a hybrid of being both easily human-readable and machine-readable. It is a markup language with strong support for Unicode (supporting various languages) and it is useful to represent arbitrary data structures from a web service application programming interface (API) to a Microsoft Office document.

One strong advantage of XML is that an XML schema may be created to express constraints on the structure and contents of a document. Two common schema languages are the **document type definition** (https://en.wikipedia.org/wiki/Document_type_definition) (DTD) language from the XML language specification and a more expressive schema language from **RELAX NG** (https://en.wikipedia.org/wiki/RELAX_NG).

An XML document is said to be "well formed" when it is syntactically correct: start tags have corresponding end tags and the document contains no illegal characters, for example. However, armed with an XML Schema Definition (XSD), it is possible to *validate* an XML document as conformant to the schema. A validated document of arbitrary complexity can be dealt with relative confidence and predictability.

Python's built-in support for XML comes via the **xml package** (https://docs.python.org/3/library/xml.html).

Are XML files examinable?

The material herein is to add to your mastery of handling structured data with Python and forms part of the practical for this topic. This material will not be examined in the Section Test for B2.

Working with XML files

XML data is less compact than JSON but, as a markup language, its intent and approach is fairly intuitive when compared to other less *decorated* formats such as CSV files which may, or may not, include "meta-data" in the form of a descriptive first line, to indicate their content.

Below is an example of an XML file. It's readily apparent that well designed XML schemas confer human readability.

```xml
<?xml version="1.0"?>
<customers>
   <person id="person1">
      <name>John</name>
      <age>52</age>
      <postcode>5002</postcode>
   </person>
   <person id="person2">
      <name>Ye</name>
      <age>18</age>
      <postcode>3005</postcode>
   </person>
   <person id="person3">
      <name>Siobhan</name>
      <age>34</age>
      <postcode>2356</postcode>
   </person>
</customers>
```

As with JSON files, the two tasks we wish to master in Python are reading XML documents into a Python data structure and saving said data structure to file. While the task of serialisation and deserialisation of JSON files maps intuitively to two Python data types, namely dictionaries and lists, XML files have quite arbitrary schemas and the task is not at all straightforward.

While the **Zen of Python** (https://www.python.org/dev/peps/pep-0020/) claims to offer just one obvious way to achieve your goal, the innate complexity of XML documents and schema languages guarantees many routes to a goal - some harder, some easier. This is true for any other programming language dealing with XML files as well.

In this course, we will present and master one of these approaches focusing on the Python standard libraries. As an example XML file type, we will investigate Scalar Vector Graphic files.

Scalar Vector Graphics?

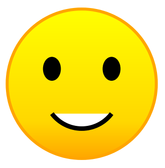
Storing image data is a tradeoff between image size and quality. A perfect representation of an image can be achieved by storing one pixel of data as three or four bytes of information in a two dimension pixel matrix. Image quality is as perfect as a sensor can provide but image size grows as the square of the image dimensions.

Alternatively, some image formats such as JPEG use an arsenal of tricks to discard information from the stored image and to compress and encode the remainder. JPEG images can dramatically reduce image size without compromising image quality disproportionately.

A third approach for created graphics is to store the image as a series of vectors - analogous to storing the strokes of a pen or brush. This results in a very small image that can be scaled up or down for accurate display: hence "Scalable Vector Graphics".

Meet an SVG File

Let's create an XML document as an SVG file for this image:



It's XML representation (available for [download here (https://myuni.adelaide.edu.au/courses/86136/files/12213416/download?wrap=1)](https://myuni.adelaide.edu.au/courses/86136/files/12213416/download?wrap=1)) is as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd" [
    <!ENTITY custom_entity "Hello">
]>
<svg xmlns="http://www.w3.org/2000/svg"
  xmlns:inkscape="http://www.inkscape.org/namespaces/inkscape"
  viewBox="-105 -100 210 270" width="210" height="270">
  <inkscape:custom x="42" inkscape:z="555">Some value</inkscape:custom>
  <defs>
    <linearGradient id="skin" x1="0" x2="0" y1="0" y2="1">
      <stop offset="0%" stop-color="yellow" stop-opacity="1.0"/>
      <stop offset="75%" stop-color="gold" stop-opacity="1.0"/>
      <stop offset="100%" stop-color="orange" stop-opacity="1"/>
    </linearGradient>
  </defs>
  <g id="smiley" inkscape:groupmode="layer" inkscape:label="Smiley">
    <!-- Head -->
    <circle cx="0" cy="0" r="50"
      fill="url(#skin)" stroke="orange" stroke-width="2"/>
    <!-- Eyes -->
    <ellipse cx="-20" cy="-10" rx="6" ry="8" fill="black" stroke="none"/>
    <ellipse cx="20" cy="-10" rx="6" ry="8" fill="black" stroke="none"/>
    <!-- Mouth -->
    <path d="M-20 20 A25 25 0 0 0 20 20"
      fill="white" stroke="black" stroke-width="3"/>
  </g>
  <text x="-40" y="75">&custom_entity; &lt;svg&gt;!</text>
  <script>
    <![CDATA[
      console.log("CDATA disables XML parsing: <svg>")
      const smiley = document.getElementById("smiley")
      const eyes = document.querySelectorAll("ellipse")
      const setRadius = r => e => eyes.forEach(x => x.setAttribute("ry", r))
      smiley.addEventListener("mouseenter", setRadius(2))
      smiley.addEventListener("mouseleave", setRadius(8))
    ]]>
  </script>
</svg>
```

While we don't need to dive into the details, observe that the document starts with an XML declaration, followed by a DTD and the `<svg>` root element. This format is somewhat akin an HTML document describing a webpage with its DOCTYPE and <html> root element. The document also contains:

- Nested elements,
- Attributes,
- Comments, and
- Character data (`CDATA`) for a behavioural script.

Reading and parsing an XML file

Reading an XML document creates a Document Object Model (DOM) from which we can access elements in the doc. This process is sharply different from reading and manipulating a JSON file in that the DOM is a more complex object representation queried using the id's and attributes of elements.

Let's look at reading some elements from the DOM describing the eyes of our smiley face:

In [1]:
```python
from xml.dom.minidom import parse

# Parse XML from a file object
with open("face.svg") as file:
     document = parse(file)

print (f"version: {document.version}, encoding: {document.encoding}.")

ellipse_elements = document.getElementsByTagName("ellipse")
for eye in ellipse_elements:
    colour = eye.getAttribute("fill")
    width =  eye.getAttribute("rx")
    height =  eye.getAttribute("ry")
    print(f"eye colour: {colour}, width: {width}, height: {height}")
```

Out[1]:
```
version: 1.0, encoding: UTF-8.
eye colour: black, width: 6, height: 8
eye colour: black, width: 6, height: 8
```

From querying the two ellipse elements used to describe the eyes, we can see they are drawn with a black fill and their height and width differ to create a distinctive, ellipsoid eye shape.

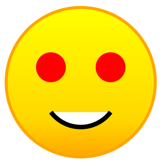
Modifying the DOM

We can also modify the DOM in memory and then write back the file as a new XML document.

In [1]:
```python
for eye in ellipse_elements:
    eye.setAttribute("fill", "red")
    eye.setAttribute("rx", "10")
    eye.setAttribute("ry", "10")
```

```
with open('red-eye.svg', 'w') as out_file:
    document.writexml(out_file, indent = "  ", addindent = "  ", newl = "\n")
```

Voila:



If we examine the XML formatted SVG file red-eye.svg, we can see that the elements that describe the eyes have been modified:

```
    <ellipse cx="-20" cy="-10" rx="10" ry="10" fill="red" stroke="none"/>
    <ellipse cx="20" cy="-10" rx="10" ry="10" fill="red" stroke="none"/>
```

Your turn for some practise: Working with XML files

In this exercise, you will practice using XML files from within a Python program.

1. Download the **face.svg (https://myuni.adelaide.edu.au/courses/86136/files/12213416/download?wrap=1)** file and open **this notebook (https://myuni.adelaide.edu.au/courses/86136/files/12213401?wrap=1) (https://myuni.adelaide.edu.au/courses/86136/files/12213401/download?download_frd=1)** .
2. Prove to yourself that you can execute the code to create your own `red-eye.svg` XML file. Remember that, unless you specify a path, you need to colocate your notebook with the SVG graphics file.
3. Open `face.svg` in a text editor - Visual Studio does a good job of highlighting XML syntax or you can ask Jupyter Notebook to open the SVG file as text.
4. Locate the "path" element that uses the "fill" attribute to specify the fill colour "white" for the teeth.
5. Modify the DOM in code to change the "fill" attribute for the mouth to "red".
6. Find the "circle" element that defines the face in the SVG document and locate the "stroke-width" attribute.
7. Set the "stroke-width" to "8".
8. Save your XML document as `red-mouth.svg` by modifying the file name when writing the XML document to file.

All being well, you should be able to open `red-mouth.svg`. with a non-text application (for example, a web browser) and see your modified XML document rendered as an image:

Hint #1 - Help! I get a NodeList exception!!!

The function document.getElementsByTagName("...") returns a list of DOM elements. If you wish to find the first element in the list, you need to index the first element in the array of elements like:

```
paths = document.getElementsByTagName("path")
mouth = paths[0]
mouth.setAttribute("fill", "red")
```

Next steps

That completes content for Topic 4 of Section B2. In the Workshop Activities we will look in depth at JSON and CSV files and practice working with these structured file formats.

B2 Topic 5: Introduction to NumPy

NumPy is an open source library that provides a number of high performance features for scientific computing with Python. These features include:

- powerful N-dimensional arrays,
- comprehensive mathematical functions for linear algebra,
- sophisticated random number generation,
- Fourier transforms, and
- much, much more! 😂

A key to NumPy's success is that it is highly performant being implemented in well-optimised C code "under the hood" with a straightforward Python interface.

Let's illustrate the need for specialised arrays with an example. Consider we have 2 variables with tabular data read from a file. For simplicity, we will call them `tbl_1` and `tbl_2`. We need to perform some basic arithmetic operations like `tbl_1 + tbl_2`.

If we proceed to use lists, we need to ensure the following conditions are met before we can perform the operations:

1. data in the lists are compatible with the operation being performed,
2. we must keep track of dimensions (shape) of the table,
3. the size (shape) of the lists match, and
4. we must also recognise and handle missing values.

From the above requirements, it is clear that handling data for computation in Pythonic collections like lists is tedious. We would have to iterate through all elements to ensure all the conditions are met. Only then can we proceed with the computation. This is where NumPy comes in handy.

NumPy Arrays

NumPy is a library for Python that provides another structure for storing and manipulating data called **arrays**. Arrays are similar to lists in that they store a group of values. However, arrays allow faster operations on vectors and matrices and may be used for calculating statistical information about data. Arrays are ideal for manipulating and storing dense data (i.e.: data where you have values for all or most attributes you're interested in) and also handle sparse and missing data elegantly.

NumPy arrays are contiguous and homogeneous and have dimensionality (shape). NumPy arrays define a new way to recognise and handle missing data. NumPy also includes linear algebra operations, such as solving systems of linear equations and computation of Eigenvectors and Eigenvalues; both of which are key operations in data science, but are not part of the standard Python math module.

If you have installed Anaconda, then you already have NumPy installed and are ready to go!

The first step to working with NumPy is to import the library:

In [1]:

```
# When you add "as" you can alias your library
# import [library] as [prefered_name]

import numpy as np
```

The core of NumPy is the **array** object class. Unlike a Python List, **all the elements of a NumPy array must be of the same type;** commonly, a numeric type like an integer (int) or decimal (float). In NumPy, vectors (one dimension) and matrices (two or more dimensions) are both called *arrays*.

NumPy arrays can contain a wide variety of data like:

Type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C size_t; normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64
float16	Half precision float:sign bit, 5-bit exponent, 10-bit mantissa
float32	Single precision float:sign bit, 8-bit exponent, 23-bit mantissa
float64	Double precision float:sign bit, 11-bit exponent, 52-bit mantissa
complex_	Inherited from Python's complex data type
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128	Complex number, represented by two 64-bit floats (real and imaginary components)

Remember: All the elements in the array must have the same data type.

Vectors (1-D arrays)

The 1-dimensional vector is the most commonly used array. 1D arrays store values at indexes, similar to lists.

array value:	8 6 4 3 6
index:	0 1 2 3 4

2D arrays

2D arrays don't just have a number of elements; they have rows and columns. For example, if you have 20 elements, those elements could be arranged as 4 rows; each with 5 columns, or 10 rows; each with 2 columns, or 5 rows; each with 4 columns. All of these different shapes store 20 elements. The *shape* of an array is the number of rows and columns the array has. The term *shape* may seem to be an odd choice of name for the number of elements. That is because the concept of shape is related to having rows and columns. In 1D arrays, you really only have columns; similar to a single row in a spreadsheet with multiple values in different columns. Often, you have multiple rows, with each row containing data related to each column.

N-dimensional arrays (ndarrays)

NumPy supports higher dimensional arrays denoted as *ndarrays*. All NumPy arrays have the property of shape. It denotes the dimensionality of the array. The shape property returns a tuple with the number of elements in the tuple corresponding to the number of dimensions of the array. For example, a 3D array of shape (2, 3, 3) has 2 rows, 3 columns with 3 elements in each column.

Note

Remember not to use the `len()` function with multi-dimensional arrays as this will only give you the number of rows, not the total number of elements!

In [2]:

```
import numpy as np

#The shape vs size of an array
a2d = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

print('The total number of elements in the array is:',a2d.size)

rows, columns = a2d.shape
print('The number of rows is:',rows,'the number of columns is:',columns)
```

```
The total number of elements in the array is: 12
The number of rows is: 3 the number of columns is: 4
```

Vectorisation

NumPy provides an easy and flexible interface to optimise computation with arrays of data. For this reason, it is ideal for data science. In NumPy, the computation can be very fast, or not, depending on the type of function you choose. If you want fast computation, you should use vectorised operations implemented by one of NumPy’s universal functions (ufuncs). Vectorisation describes the use of optimised, pre-compiled code written in a low-level language (like the language C) to perform mathematical operations over a sequence of data. This is done in place of an explicit iteration in Python - for example, instead of using a for-loop statement.

Vectorised arithmetic operators (ufunc)

NumPy's universal functions (ufuncs) provide vectorised implementations of arithmetic functions. Use these whenever you need to do operations over large data sets in arrays, instead of using a for loop.

Operator	unfunc	Description
`+`	`np.add`	Addition
`-`	`np.subtract`	Subtraction
`-`	`np.negative`	Unary negation (e.g $-5$)
`*`	`np.multiply`	Multiplication
`/`	`np.divide`	Division
`**`	`np.power`	Exponentiation
`//`	`np.floor_divide`	Integer division
`%`	`np.mod`	Modulo (division remainder)

Example of arithmetic operations

In [3]:

```
import numpy as np

# Addition

a1 = np.arange(9).reshape(3,3)
a2 = np.arange(3)
print('a1=\n', a1)
print('a2=\n', a2)

print('Addition a1 + 5 =\n', a1+5)
print ('np.add a1 + 5 =\n', np.add(a1,5))

# Addition two arrays
print ('np.add a1 + a2 =\n', np.add(a1,a2))
```

```
a1=
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
a2=
 [0 1 2]
Addition a1 + 5 =
 [[ 5  6  7]
 [ 8  9 10]
 [11 12 13]]
np.add a1 + 5 =
 [[ 5  6  7]
 [ 8  9 10]
 [11 12 13]]
np.add a1 + a2 =
 [[ 0  2  4]
 [ 3  5  7]
 [ 6  8 10]]
```

In [4]:

```
# Subtraction

print('a1=\n', a1)

print('Subtraction a1 - 5 =\n', a1-5)
print ('np.subtract a1 - 5 =\n', np.subtract(a1,5))
```

```
a1=
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
Subtraction a1 - 5 =
 [[-5 -4 -3]
 [-2 -1  0]
 [ 1  2  3]]
np.subtract a1 - 5 =
 [[-5 -4 -3]
 [-2 -1  0]
 [ 1  2  3]]
```

In [5]:

```
# Negation

print('a1=\n', a1)

print('Negative -a1 =\n', -a1)
print ('np.negative a1  =\n', np.negative(a1))
```

```
a1=
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
Negative -a1 =
 [[ 0 -1 -2]
 [-3 -4 -5]
 [-6 -7 -8]]
np.negative a1  =
 [[ 0 -1 -2]
 [-3 -4 -5]
 [-6 -7 -8]]
```

In [6]:

```
# Multiplication

a1 = np.arange(9).reshape(3,3)
a2 = np.arange(3)
print('a1=\n', a1)
print('a2=\n', a2)

print('Multiplication a1*a2 =\n', a1*a2)
print ('np.multiply (a1,a2)  =\n', np.multiply(a1,a2))
```

```
a1=
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
a2=
 [0 1 2]
Multiplication a1*a2 =
 [[ 0  1  4]
 [ 0  4 10]
 [ 0  7 16]]
np.multiply (a1,a2)  =
 [[ 0  1  4]
 [ 0  4 10]
 [ 0  7 16]]
```

In [7]:

```
# Division

a1 = np.arange(9).reshape(3,3)
a2 = np.arange(3)
print('a1=\n', a1)

print('Division a1/2 =\n', a1/2)
print ('np.divide (a1,2)  =\n', np.divide(a1,2))
print('a2=\n', a2)
print('np.divide (a1,a2) =\n', np.divide(a1,a2))
#NOTE: Be careful with the division by zero! Anaconda give you a warning and the result of dividin
g
# a number by zero will be infinity or nan 0/0
```

```
a1=
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
Division a1/2 =
 [[0.  0.5 1. ]
 [1.5 2.  2.5]
 [3.  3.5 4. ]]
np.divide (a1,2)  =
 [[0.  0.5 1. ]
 [1.5 2.  2.5]
 [3.  3.5 4. ]]
a2=
 [0 1 2]
np.divide (a1,a2) =
 [[nan 1.  1. ]
 [inf 4.  2.5]
 [inf 7.  4. ]]
```

```
~/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:8: RuntimeWarning: divide by zero en
countered in true_divide

~/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:8: RuntimeWarning: invalid value enc
ountered in true_divide
```

In [8]:

```
# Integer Division

a1 = np.arange(9).reshape(3,3)
a2 = np.arange(3)
print('a1=\n', a1)

print('Integer division a1//2 =\n', a1//2)
```

```
print ('np.divide (a1,2)  =\n', np.floor_divide(a1,2))
print('a2=\n', a2)
print('np.floor_divide (a1,a2)=\n', np.floor_divide(a1,a2))
```

```
a1=
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
Integer division a1//2 =
 [[0 0 1]
 [1 2 2]
 [3 3 4]]
np.divide (a1,2)  =
 [[0 0 1]
 [1 2 2]
 [3 3 4]]
a2=
 [0 1 2]
np.floor_divide (a1,a2)=
 [[0 1 1]
 [0 4 2]
 [0 7 4]]
```

```
~/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:8: RuntimeWarning: divide by zero en
countered in floor_divide
```

In [9]:

```
#Exponentiation

a1 = np.arange(9).reshape(3,3)
print('a1=\n', a1)

print('Exponentiation a1**=\n', a1**2)
print ('np.power (a1,2)=\n', np.power(a1,2))
```

```
a1=
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
Exponentiation a1**=
 [[ 0  1  4]
 [ 9 16 25]
 [36 49 64]]
np.power (a1,2)=
 [[ 0  1  4]
 [ 9 16 25]
 [36 49 64]]
```

In [10]:

```
#Modulo (division remainder)

a1 = np.arange(9).reshape(3,3)
print('a1=\n', a1)

print('Modulo a1/3=\n', a1%3)
print ('np.mod (a1,3)=\n', np.mod(a1,3))
```

```
a1=
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
Modulo a1/3=
 [[0 1 2]
 [0 1 2]
 [0 1 2]]
np.mod (a1,3)=
 [[0 1 2]
 [0 1 2]
 [0 1 2]]
```

Other ufunc Functions

NumPy has many other functions. Some of these are the NaN-safe version. This means that NumPy will ignore missing values when applying the functions. The following table lists other functions in ufuncs.

Function	Nan-safe Version	Description
`np.all()`	Not available	Evaluate whether all elements are true
`np.any()`	Not available	Evaluate whether any elements are true
`np.argmax()`	`np.nanargmax()`	Find index of maximum value
`np.argmin()`	`np.nanargmin()`	Find index of minimum value
`np.max()`	`np.nanmax()`	Find maximum value
`np.mean()`	`np.nanmean()`	Compute mean of elements
`np.median()`	`np.nanmedian()`	Compute median of elements
`np.min()`	`np.nanmin()`	Find minimum value
`np.percentile()`	`np.nanpercentile()`	Compute rank-based statistics of elements
`np.prod()`	`np.nanprod()`	Compute product of elements
`np.std()`	`np.nanstd`	Compute standard deviation
`np.sort()`	Not available	return a sorted copy of an array
`np.sum()`	`np.nansum()`	Compute sum of elements
`np.transpose()`	Not available	Permute the dimensions of an array
`np.var()`	`np.nanvar()`	Compute variance

Some common-use examples follow, but you can find all of the function details in the **NumPy Reference (https://docs.scipy.org/doc/numpy-1.14.0/reference/index.html)** . You should look over the functions to see what is available should you need it.

You can also get a list of all the methods that NumPy (`np`) has. type `np.` or `NumPy.` in a Jupyter Notebook code cell and press the tab key (note you need to type the '.'). You will see a menu with all the methods which you can navigate with arrow keys.

Your Turn for some Practice

Try this now in your Jupyter Notebook

In [11]:

```
import numpy as np

# Example: Find the minimum and maximum number in an array

a1=np.random.uniform(-1000, 1000, size=(3, 4)) # generate a random number
print("a1= \n",a1)
print()

# If you don't specify in the function max or min if you want the max/min
# of each row or column it will find the max or min in the entire array

print("Minimum number in all a1: ", np.min(a1))
print("Maximum number in all a1: ", np.max(a1))
print()

# looking per column axis=0

print("Minimum number per column in a1: ", np.min(a1, axis=0))
print("Maximum number per column in a1: ", np.max(a1, axis=0))
print()

# looking per row axis=1

print("Minimum number per row in a1: ", np.min(a1, axis=1))
print("Maximum number per row in a1: ", np.max(a1, axis=1))
```

Out [11]:

```
a1=
 [[-747.77810106 -743.16212313   25.82845025   10.86801941]
 [-838.46590937 -514.73367799 -104.20638905 -380.15258846]
 [-556.77717196  861.82194049  226.90126443  -13.26003961]]

Minimum number in all a1:  -838.4659093669034
Maximum number in all a1:  861.8219404875499

Minimum number per column in a1:  [-838.46590937 -743.16212313 -104.20638905 -380.15258846]
Maximum number per column in a1:  [-556.77717196  861.82194049  226.90126443   10.86801941]

Minimum number per row in a1:  [-747.77810106 -838.46590937 -556.77717196]
Maximum number per row in a1:  [  25.82845025 -104.20638905  861.82194049]
```

In [12]:

```
# Transpose of an array
a2= np.random.randint(low=-10, high=10, size=(2, 3))
print('a2 =\n', a2)

print('Transpose of a2 =\n', np.transpose(a2) )
```

Out [12]:

```
a2 =
 [[ -2   1  -8]
 [  7 -10  -4]]
Transpose of a2 =
 [[ -2   7]
 [  1 -10]
 [ -8  -4]]
```

Readings

- The **NumPy Reference** (https://numpy.org/doc/stable/) at numpy.org.
- The **NumPy User Guide** (https://numpy.org/doc/stable/user/index.html) at numpy.org particularly the **Quickstart overview** (https://numpy.org/doc/stable/user/quickstart.html) that provides additional examples of basic operations.
- Nelli, F. (2018). **The NumPy Library** (https://rdcu.be/cPYZa) . In: *Python Data Analytics*. Apress, Berkeley, CA.

(h (h (h (h (h (h (h (h (h (h (h

B2 Topic 5: Creating and Manipulating Arrays

Creating a 1D array

There are several ways you can create arrays with NumPy:

- Using the `arange()` function.
- Providing a list.
- Using the `zeros()` function.

Creating a 1D array with a range

Use NumPy `arange()` when the values you want in your array are between a minimum and maximum value with a consistent interval (i.e. amount of space) between them. For example, if you want the even numbers between 2 and 20, the minimum is 2, the maximum is 20 and the interval is 2 — you want every second number added to your array.

The format of the `arange()` function is:

`arange([start,] stop[, step,], dtype=None)`

- **start** of the interval is a numeric value included in the interval and it is optional. The default start value is 0.
- **stop** is the end of the interval. It is a numeric value not included in the interval.
- **step** is the space between the values. it is a numeric and optional value. The default step size is 1.
- **dtype** indicates the type of the array. If dtype is not given, infer the data type from the other input arguments.

View examples

In [1]:

```
import  numpy as np

# Using Numpy arange with just stop argument.
# start and step default to 0 and 1 and the type is defined by the type of the stop argument (int)
a1=np.arange(5)
a1
```

Out[1]:

```
array([0, 1, 2, 3, 4])
```

In [2]:

```
# the type of an array
type(a1)
```

Out[2]:

```
numpy.ndarray
```

In [3]:

```
# Creating an array of floats
np.arange(5.0)
```

Out[3]:

```
array([0., 1., 2., 3., 4.])
```

In [4]:

```
# Specifying start and stop - step defaults to 1 and dtype defaults to the arguments (int)
# NOTE: The interval does not include 12
np.arange(5,12)
```

Out[4]:

```
array([ 5,  6,  7,  8,  9, 10, 11])
```

In [5]:

```
# Specifying a step
np.arange(5,12,2)
```

Out[5]:

```
array([ 5,  7,  9, 11])
```

Creating a 1D array from lists or tuples

If you have your data in list form (for example, after reading it from a CSV file), you can convert it into an array using the function `array()`. The main arguments of the function are the *list* to be converted into the array and the *type*.

View examples

In [6]:

```
# Create an array using the array( ) function
a1 = np.array([0, 1, 2, 3, 4],float)
a1
```

Out[6]:

```
array([0., 1., 2., 3., 4.])
```

In [7]:

```
# If your list has floats and you define the type as int, the function array() will
# truncate the digits after the decimal point
a2 = np.array([0, 1.0, 2.5, 3.4, 4],int)
a2
```

Out[7]:

```
array([0, 1, 2, 3, 4])
```

In [8]:

```
# Creating an array of strings
a2 = np.array([0, 1.0, 2.5, 3.4, 4],str)
a2
# NOTE: the numbers are converted to U3 (this is unicode a common representation for characters)
```

Out[8]:

```
array(['0', '1.0', '2.5', '3.4', '4'], dtype='<U3')
```

Creating a 1D array with zeros ()

The `zeros()` function will create an array filled with zeros for you. This is useful when some values may be missing in your data — you can represent the missing information with zeros.

The format of the `zeros()` function is:

`zeros(shape, dtype , order)`

- **shape** int or tuple of ints.
- **dtype** indicates the type of the array and it is optional or you can use any of the types define in NumPy. For example `np.int64` or `np.float64`.
- **order** defines how to store the multi-dimensional data row-major (programming language C-style) or column-major (programming language Fortran-style). The argument order is optional and the default is 'C'. It is unlikely you will need to use Fortran order, and you won't see any difference when working with the array in Python (it will look the same to you), but be aware Fortran ordering exists should you need it.

View examples

In [9]:

```
# Create an array using zeros()
# only shape is specified: 3 elements
# dtype defaults to float
# order defaults to C-style.
a3 = np.zeros(3)
a3
```

Out[9]:

```
array([0., 0., 0.])
```

In [10]:

```
# specifying data type
a3 = np.zeros((3), dtype=int)
a3
```

Out[10]:

```
array([0, 0, 0])
```

Creating 2D Arrays

The shape property can be used to create zero-filled arrays with the `zeros()` function by specifying the number of rows and columns in the array as a tuple. Note the different shapes created in these examples:

In [11]:

```
# Creating a multi-dimensional array of zeros
a4 = np.zeros((3,4))
a4
```

Out[11]:

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

In [12]:

```
# Creating a multi-dimensional array of zeros
a4 = np.zeros((4,3))
a4
```

Out[12]:

```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

You can also create 2D arrays from a list of lists.

In [13]:

```
# Create a 2D array from nested lists
a4 = np.array([[1,2,3], [4,5,6], [7,9,9]])
a4
```

Out[13]:

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 9, 9]])
```

Slicing Sections of an Array

To access an individual array element, you use square brackets, just like lists. You can also access sub arrays through *slicing*. Slicing arrays can be useful when you work with large data sets. You can access and process pieces of these data sets without the need to manage all of the data set. This reduces the time needed to do your calculations.

You take a slice of an array with the slice notation, marked by the colon (:) character.

`array1[start: stop: step]`

start, stop and step have the same behaviour as in the function `arange()`.

Remember if any of these parameters are unspecified, they default to the values start=0, stop=last index, step=1.

Let's start with **1-dimensional array** examples.

View examples

In [14]:

```
# Examples of Slicing in one-dimensional array
a1 = np.arange(20) #create a one-dimensional array in a range with 20 elements starting by zero
print(a1)

#Slice out the first seven elements: stop at index 7.  start and step have default values
a1[:7]
```

Out[14]:

```
[ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19]
array([0, 1, 2, 3, 4, 5, 6])
```

In [15]:

```
# Take a slice from the middle of the array
# starting with element at index 5 stopping at index 15
a1[5:15]
```

Out[15]:

```
array([ 5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

In [16]:

```
# Get a slice with every second element from the array
a1[::2]
```

Out[16]:

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

In [17]:

```
# Take a slice with every third element starting at 4
a1[4::3]
```

Out[17]:

```
array([ 4,  7, 10, 13, 16, 19])
```

Having a negative step can be confusing, but the effect is that the values for start and stop are swapped. This can convenient when you want to reverse an array.

View examples

In [18]:

```
# Reversing all the elements
a1[::-1]
```

Out[18]:

```
array([19, 18, 17, 16, 15, 14, 13, 12, 11, 10,  9,  8,  7,  6,  5,  4,  3,
        2,  1,  0])
```

In [19]:

```
# Reversing a part of the array, starting with the element at index 10
a1[10::-2]
```

Out[19]:

```
array([10,  8,  6,  4,  2,  0])
```

Multidimensional slices work in the same way as accessing sections of an array, with multiple slices separated by commas. Try these examples out in your Jupyter notebook.

In [20]:

```
# Create a multidimensional array
a2 = np.array([[3, 12, 5, 65],[13, 90, 2, 49], [35, 79, 1, 8]])
a2
```

Out[20]:

```
array([[ 3, 12,  5, 65],
       [13, 90,  2, 49],
       [35, 79,  1,  8]])
```

In [21]:

```
# Slice the first two rows and the first three columns
a2[:2, :3]
```

Out[21]:

```
array([[ 3, 12,  5],
       [13, 90,  2]])
```

In [22]:

```
# Slice all the rows and the first column
a2[:,:1]
```

Out[22]:

```
array([[ 3],
       [13],
       [35]])
```

In [23]:

```
# Slice all the rows and every other column
a2[:3, ::2]
```

Out[23]:

```
array([[ 3,  5],
       [13,  2],
       [35,  1]])
```

In [24]:

```
# Use slices to invert the rows and columns
# Slice the entire array, but step -1 for both rows and columns
print('original a2 \n', a2)
print('inverted a2 \n', a2[::-1, ::-1])
```

Out [24]:

```
original a2
 [[ 3 12  5 65]
 [13 90  2 49]
 [35 79  1  8]]
inverted a2
 [[ 8  1 79 35]
 [49  2 90 13]
 [65  5 12  3]]
```

Reshaping arrays

Sometimes you may need to change the shape of an array you have created. For example, if you add a new column of data. You can do this using the `reshape()` function. You can also use `reshape()` to create a 2-dimensional array from 1D data.

The `reshape()` function takes the number of rows and columns to use for the 2D array.

In [25]:

```
# Using re-shape function and arrange to create a two-dimensional array from 1D data

#Create a 1D array
a1d = np.arange(0,12)
print ('a1d before reshape ', a1d)

# Reshape data to a 3x4 array
```

```
a1d=a1d.reshape(3,4)
print ('a1d after reshape \n', a1d)
```

Out [25]:

```
a1d before reshape  [ 0  1  2  3  4  5  6  7  8  9 10 11]
a1d after reshape
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

In [26]:

```
# You can also combine the calls into one line
# the arange( ) is called first, and then reshape( ) is called on the array returned by arange( )
a2d = np.arange(0,15).reshape(3,5)
a2d
```

Out[26]:

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

In [27]:

```
# What happens if your one-dimensional array can not fit in your new shape
a1d = np.arange(0,17).reshape(3,5)

# You can't reshape arrays with different sizes. In this case we have a range of 17 elements
# and the reshape size is 3X5 = 15 elements
```

Out [27]:

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-55-413732923524> in <module>
      1 #What happens if your one-dimensional array can not fit in your new shape
----> 2 a1d = np.arange(0,17).reshape(3,5)
      3 
      4 #You can't reshape arrays with different sizes. In this case we have a range of 17 elements
      5 #and the reshape size is 3X5 = 15 elements

ValueError: cannot reshape array of size 17 into shape (3,5)
```

Concatenate arrays

We can combine multiple arrays joining them into one array using:

- `np.concatenate()` Join a sequence of arrays along an existing axis (rows or columns). Parameters:
  - A sequence of arrays to concatenate. The arrays must have the same shape, except in the dimension corresponding to axis (rows by default). This means that there can be different numbers of rows, but all of the arrays must have the same number of columns.
  - axis : int, optional. The axis along which the arrays will be joined. Default is 0 (rows). 1 selects columns.

View examples

In [28]:

```
# Example of concatenation one-dimensional arrays
a1 = np.array([-1, -2, -3])
a2 = np.array([3, 2, 1])
a3 = np.array([4, 5, 6])

# Remember, you can concatenate more than two arrays
np.concatenate([a1, a2, a3])
```

Out[28]:

```
array([-1, -2, -3,  3,  2,  1,  4,  5,  6])
```

In [29]:

```
# Example of concatenation two-dimensional arrays
matrix = np.array([[1, 2, 3],[4, 5, 6]])
print('matrix = \n', matrix)
np.concatenate([matrix, matrix])#by default in axis 0 (rows)
```

Out [29]:

```
matrix = [[1 2 3] [4 5 6]]

array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
```

In [30]:

```
# Example of concatenate in axis 1 (columns )
matrix = np.array([[1, 2, 3],[4, 5, 6]])
print('matrix = \n', matrix)
np.concatenate([matrix, matrix],axis = 1)
```

Out[30]:

```
matrix = [[1 2 3] [4 5 6]]

array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

In [31]:

```
# Dimensions MUST match other than the dimension being joined
# v1 only has 2 columns, can't join on rows!
v1 =np.array([[1, 2],[3,4]])
matrix = np.array([[4, 5, 6],[7, 8, 9]])
np.concatenate([matrix, v1])
```

Out [31]:

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-108-39705cdff0a6> in <module>
      3 v1 =np.array([[1, 2],[3,4]])
      4 matrix = np.array([[4, 5, 6],[7, 8, 9]])
----> 5 np.concatenate([matrix, v1])

ValueError: all the input array dimensions except for the concatenation axis must match exactly
```

In [32]:

```
# But you could concatenate on columns (both v1 and matrix have two rows)
np.concatenate([matrix, v1],axis=1)
```

Out[32]:

```
array([[4, 5, 6, 1, 2],
       [7, 8, 9, 3, 4]])
```

- `np.vstack()` Stack arrays in sequence vertically (row wise). Parameters:
  - Arrays to concatenate. The arrays must have the same shape along all but the first axis (rows). 1D arrays must have the same length.

View example

In [33]:

```
# Example of np.vstack (vertical or row)
v1 =np.array([1, 2, 3])
matrix = np.array([[4, 5, 6],[7, 8, 9]])
np.vstack([v1, matrix])
```

Out[33]:

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

- `np.hstack()` Stack arrays in sequence horizontally (column wise). Parameters
  - Arrays to concatenate. The arrays must have the same shape along all but the second axis (columns), except 1D arrays which can be any length.

View example

In [34]:

```
# Example of np.hstack (horizontal or columns)
matrix = np.array([[4, 5, 6],[7, 8, 9]])
print('matrix = \n', matrix)
h1 =np.array([[-1],[-2]])
print('h1 = \n', h1)
np.hstack([h1,matrix])
```

Out[34]:

```
matrix = [[4 5 6] [7 8 9]]

h1 = [[-1] [-2]]

array([[-1,  4,  5,  6],
       [-2,  7,  8,  9]])
```

Split arrays

You can also split an array into several arrays using:

- `np.split()`: Split an array into multiple sub-arrays. Parameters:
    - Array to be divided into sub-arrays
    - Indices or sections: int or one-dimensional array.
        - If indices_or_sections is an integer, N, the array will be divided into N equal arrays along axis.
        - If indices_or_sections is an array, each number in the array indicates where the next section ends.
        - If such a split is not possible, an error is raised.

View examples

In [35]:

```
# Example of np.split of one-dimensional array into 4 equal sized arrays
vec = np.arange(4)
print('vec = ', vec)

v1,v2,v3,v4 = np.split(vec,4)
print('v1 = ',v1)
print('v2 = ',v2)
print('v3 = ',v3)
print('v4 = ',v4)

# Change the split from 4 to 3 in your notebook at run it to see what happens.
# v1,v2,v3 = np.split(vec,3)
```

Out [35]:

```
vec =  [0 1 2 3]
v1 =  [0]
v2 =  [1]
v3 =  [2]
v4 =  [3]
```

In [36]:

```
# Example of np.split of one-dimensional array into 3 parts  Rows 0-2, rows 3-4 and rows 5-8
# note [3,5] 3 indicates that the first split ends at index 3.  So this first split includes index
es 0-2,
# second split starts at next index, 3.
# 5 indicates second split ends at index 5.  So the second split will include indexes 3-4
# when there are no more indexes, whatever is left in the array will end up in a remainder split
vec = np.arange(8)
print('vec = ', vec)

v1,v2,v3 = np.split(vec,[3,5])
print('v1 = ',v1)
print('v2 = ',v2)
print('v3 = ',v3)
```

Out [36]:

```
vec =  [0 1 2 3 4 5 6 7]
v1 =  [0 1 2]
```

```
v2 =  [3 4]
v3 =  [5 6 7]
```

In [37]:

```
# Example of np.split in one-dimensional array into 5 parts
vec = np.arange(10.0)
print(vec)

(print(np.split(vec, [3, 5, 6, 10])))
# note that the last array is empty as there are no remaining elements after index 9
```

Out [37]:

```
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
[array([0., 1., 2.]), array([3., 4.]), array([5.]), array([6., 7., 8., 9.]), array([], dtype=float
64)]
```

In [38]:

```
# Example with two-dimensional array

#Create a two-dimensional array
m1 = np.arange(24).reshape((6, 4))
print("m1 =\n", m1)

#Split into two equal arrays
mS1,mS2 = np.split(m1,2);#[2]index for rows, the two first
print("mS1 =\n", mS1)
print("mS2 =\n", mS2)
```

Out [38]:

```
m1 =
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
mS1 =
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
mS2 =
 [[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

In [39]:

```
# Example with two-dimensional array

# Create a two-dimensional array
m1 = np.arange(24).reshape((6, 4))
print("m1 =\n", m1)

#Split the first two rows from the rest
mS1,mS2 = np.split(m1,[2]);
print("mS1 =\n", mS1)
print("mS2 =\n", mS2)
```

```
m1 =
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
mS1 =
 [[0 1 2 3]
 [4 5 6 7]]
mS2 =
 [[ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

In [40]:

```
# You can also use a negative index to indicate the offset from the end of the array rather than t
he beginning
# split the array, stop at the second row from the end
mS1, mS2 = np.split(m1,[-2]);
print("mS1 =\n", mS1)
print("mS2 =\n", mS2)
```

```
mS1 =
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
mS2 =
 [[16 17 18 19]
 [20 21 22 23]]
```

- `np.vsplit()`
    - Array to be divided into sub-arrays
    - Indexes of sections

View examples

In [41]:

```
# Example of vsplit ()
# Note the behaviour of vsplit is the same as split( ).  split optionally takes another argument i
ndicating
# the axis (row or column) to use.  The default is row.  vsplit always splits on rows.
print("m1 =\n", m1)

# split the array into 2 equal sections of rows
above,  below = np.vsplit(m1, 2)
print("above =\n", above)
print("below =\n", below)
```

```
m1 =
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
above =
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
below =
 [[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

In [42]:

```
# Example of vsplit ()
print("m1 =\n", m1)

# split the array into three parts: up to row 2, row 2, rest of array
a,b,c= np.vsplit(m1, np.array([2,3]))
print('a = \n',a)
print('b = \n',b)
print('c = \n',c)
```

```
m1 =
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
a =
 [[0 1 2 3]
 [4 5 6 7]]
b =
 [[ 8  9 10 11]]
c =
 [[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

- `np.hsplit()`
  - Array to be divided into sub-arrays
  - Indexes of sections

View examples

In [43]:

```
# Example of hsplit ()
# hsplit() always splits on columns
print("m1 =\n", m1)

# split array into 2 even sections of columns
left,right= np.hsplit(m1, 2)
print('left = \n',left)
print('right = \n',right)
```

```
m1 =
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
left =
 [[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]
 [16 17]
 [20 21]]
right =
 [[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]
 [18 19]
 [22 23]]
```

In [44]:

```
# Example of hsplit ()
print("m1 =\n", m1)

#split array into three parts: up to column 2, column 2, rest of array
a,b, c= np.hsplit(m1,np.array([2,3]))
print('a = \n',a)
print('b = \n',b)
print('c = \n',c)
```

```
m1 =
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
a =
 [[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]
 [16 17]
 [20 21]]
b =
 [[ 2]
 [ 6]
 [10]
 [14]
 [18]
 [22]]
c =
 [[ 3]
 [ 7]
 [11]
 [15]
 [19]
 [23]]
```

Copy arrays

Arrays present the same problem as lists when you assign an array to another array. If you modify either of the arrays the other array is modified as well. This behaviour occurs as NumPy arrays are mutable objects. This can be avoided by using the `copy()` method or `np.array()`, which will give you a new array that has the same values.

In [45]:

```
# ERROR - Copy an array in another array WITHOUT copy()
a2 =np.array([[3, 12, 5, 65],[13, 90, 2, 49], [35, 79, 1, 8]])
print('a2 = \n',a2)

a2Slice= a2[:2,:2]
print('\n a2Slice = \n',a2Slice)

# What happens if we modified a position of the array?
a2Slice[0, 0] = -10
print('\n a2Slice after modified= \n',a2Slice)
print('\n a2 is modified too! =\n', a2)
```

```
a2 =
 [[ 3 12  5 65]
 [13 90  2 49]
 [35 79  1  8]]

 a2Slice =
 [[ 3 12]
 [13 90]]

 a2Slice after modified=
 [[-10  12]
 [ 13  90]]

 a2 is modified too! =
 [[-10  12   5  65]
 [ 13  90   2  49]
 [ 35  79   1   8]]
```

In [46]:

```
# Copy an array in another array with copy()

# If you want an independent copy use copy() or np.array
a2 =np.array([[3, 12, 5, 65],[13, 90, 2, 49], [35, 79, 1, 8]])
print('a2 = \n',a2)

a2Slice= a2[:2,:2].copy()
print('\n a2Slice with copy() = \n',a2Slice)

# What happens if we modified a position of the array?
a2Slice[0, 0] = -10
print('\n a2Slice after modified= \n',a2Slice)
print('\n a2 is NOT modified =\n', a2)

a2Slice = np.array(a2[:2,:2])
print('\n a2Slice with np.array() = \n',a2Slice)

# What happens if we modified a position of the array?
a2Slice[0, 0] = -10
print('\n a2Slice after modified= \n',a2Slice)
print('\n a2 is NOT modified =\n', a2)
```

```
a2 =
 [[ 3 12  5 65]
 [13 90  2 49]
 [35 79  1  8]]

 a2Slice with copy() =
 [[ 3 12]
 [13 90]]

 a2Slice after modified=
 [[-10  12]
 [ 13  90]]
```

```
a2 is NOT modified =
[[ 3 12  5 65]
[13 90  2 49]
[35 79  1  8]]

a2Slice with np.array() =
[[ 3 12]
[13 90]]

a2Slice after modified=
[[-10  12]
[ 13  90]]

a2 is NOT modified =
[[ 3 12  5 65]
[13 90  2 49]
[35 79  1  8]]
```

Readings

- Nelli, F. (2018). **The NumPy Library (https://rdcu.be/cPYZa)** . In: *Python Data Analytics*. Apress, Berkeley, CA.

Next Steps

In this week's Workshop, we will practice working through examples with NumPy.

B2 Topic 5: Introduction to pandas

In the first topic this week, we explored the NumPy library. Despite its usefulness, NumPy has limitations - namely it inability to handle mixed type datasets and labelled data. Numpy can't store a collection of data with different data types.

A typical example might be a student attendance sheet. In such cases, each datapoint (a student record) has multiple types (for example: string for the student's name, an integer for marks) and a label (for example: a student ID). NumPy arrays can't store such mixed data in a single array or collection and can't index labels. Pandas comes in handy in such cases.

Pandas Library

The pandas library is a perfect tool for anyone who wants to perform data analysis using Python as a programming language. pandas is an open source library for highly specialized data analysis. It is widely used for the statistical purposes of analysis and decision making.

pandas introduces two new data types: `Series` and `DataFrame`.

A pandas Series is a one-dimensional data structure that comprises of a key-value pair. It is similar to a Python dictionary, except it provides more freedom to manipulate and edit the data. In contrast, a pandas DataFrame is a two-dimensional data-structure that can be thought of as akin to a *spreadsheet*. A DataFrame can also be thought of as a *combination of two or more series*.

These types allow you to index by numbers or labels and DataFrames allow you to mix different types within your data. In addition to gaining these benefits, pandas is built on top of NumPy and provides the performance of NumPy arrays.

The first step to working with pandas is to import the pandas and NumPy library.

In [1]:

```
import pandas as pd
import numpy as np
```

Data Structures in Pandas

The core of Pandas are the following data structures:

- **Series** — an object used to represent a 1-dimensional array of indexed data.
- **Dataframe** — consists of an ordered collection of columns (similar to a spreadsheet), each column can contain a value of a different type like string, date, numeric, etc.

Series

You can create a series from a list or array. In the figure, you can see the structure of a series.

Series

Index	Value
0	-5
1	10
2	-3
3	-25
4	45

This looks a lot like NumPy arrays, but pandas gives you more ways to index the data. Sometimes you want to create a series using significant labels other than numbers to distinguish and identify each item regardless of the order in which they were inserted into the series.

The main difference between a pandas series and NumPy array is the index. While the NumPy array only has an **implicitly** defined integer *index used to access the values*, the Pandas series can have an **explicitly** defined *index used to access the values*. With series, the indexes can be numeric (default) or any immutable type.

Constructing Series Objects

Let's construct a Series:

`pd.Series(data, index= index)`

- `data` can be an array-like, iterable, dict, or scalar (ie: a single value, like int, string, etc.) value.
- `index` is an optional parameter, by default it is an integer sequence starting from zero.

View examples

In [2]:

```
# Construct a series with a default index

pd.Series([5,10, 15, 20])
```

Out[2]:

```
0     5
1    10
```

```
2    15
3    20
dtype: int64
```

In [3]:

```
# Construct a series with label indexes

pd.Series(22,['a','b','c','d','e'])
```

Out[3]:

```
a    22
b    22
c    22
d    22
e    22
dtype: int64
```

In [4]:

```
# Construct a series from a dictionary

pd.Series({2:'orange', 3:'apple', 1:'nectarine', 4:'strawberry'})
```

Out[4]:

```
2        orange
3         apple
1     nectarine
4    strawberry
dtype: object
```

In [5]:

```
# Construct a series from specific indexes from a dictionary

pd.Series({2:'orange', 3:'apple', 1:'nectarine', 4:'strawberry'}, index=[2,4])
```

Out[5]:

```
2        orange
4    strawberry
dtype: object
```

In [6]:

```
# Creating a Series from an array
# create an array one-dimensional randomly between -50 to 50

a1 = np.random.randint(low=-50, high =50, size=10)
data = pd.Series(a1)
data

#NOTE: the Series contains both a sequence of values and a sequence of indices.
```

Out[6]:

```
0   -47
1    38
2   -13
3   -10
4   -13
5     3
6   -45
7   -18
8   -48
9    33
dtype: int64
```

In [7]:

```
# Create the same Series with specific indexes

data = pd.Series(a1, index=['a','b', 'c', 'd', 'e', 'f', 'h', 'i', 'j', 'k'])
data
```

Out[7]:

```
a   -47
b    38
c   -13
d   -10
e   -13
f     3
h   -45
i   -18
j   -48
k    33
dtype: int64
```

In [8]:

```
# visualize the values

data.values
```

Out[8]:

```
array([-47,  38, -13, -10, -13,   3, -45, -18, -48,  33])
```

In [9]:

```
# visualize the indexes

data.index
```

Out[9]:

```
Index(['a', 'b', 'c', 'd', 'e', 'f', 'h', 'i', 'j', 'k'], dtype='object')
```

Accessing the data in a series is similar to accessing NumPy arrays: using an index value inside of square brackets. Remember, if you are using the implicit (default) numeric index, the index starts at zero.

Also, you can access a slice of the data as you did with arrays in NumPy.

`<series_name> [start, stop, step]`

View example

In [10]:

```
# getting the data in the position 6
# The 6 value is the index label of the single item in the Series whose value is -13.

data[6]
```

Out[10]:

```
-45
```

In [11]:

```
# Slice the series

data[3:8]
```

Out[11]:

```
d   -10
e   -13
f     3
h   -45
i   -18
dtype: int64
```

In [12]:

```
# Reverse the series

data[::-1]
```

Out[12]:

```
k    33
j   -48
i   -18
h   -45
f     3
e   -13
d   -10
c   -13
b    38
a   -47
dtype: int64
```

In [13]:

```
# Getting the data at every second position

data[::2]
```

Out[13]:

```
a   -47
c   -13
e   -13
h   -45
j   -48
dtype: int64
```

Series are mutable objects. Recall that if you assign mutable objects, they refer to the same object. Be sure to use the Series `copy()` method if you want a copy.

View example

In [14]:

```
# Note: When you assign a mutable object to a series changes to the series or mutable
# object will change both!

# create an array one-dimensional with random integers between -50 to 50
a1 = np.random.randint(low=-50, high =50, size=10)

# create a data series fro this array
data = pd.Series(a1, index=['a','b', 'c', 'd', 'e', 'f', 'h', 'i', 'j', 'k'])
```

```
print('data =')
print(data)
print()

print('Before modify a1[0], data[0]= ',data[0])

# modifying the array, changes the data series!
a1[0]=100
print('a1[0] = ', a1[0])
print('After modify a1[0], data[0]= ',data[0])
print()

# To avoid, you need to copy the arrays or series.
print('To avoid this problem, we create the series with a copy of a1 array')
data = pd.Series(np.copy(a1), index=['a','b', 'c', 'd', 'e', 'f', 'h', 'i', 'j', 'k'])
print()
print('Before modifying a1[0], data[0]= ',data[0])
a1[0]=-187
print('a1[0] = ', a1[0])
print('After modifying a1[0], data[0]= ',data[0])
```

```
data =
a     -7
b    -10
c    -25
d    -13
e     17
f      0
h    -32
i    -31
j    -46
k    -23
dtype: int64

Before modify a1[0], data[0]=  -7
a1[0] =  100
After modify a1[0], data[0]=  100

To avoid this problem, we create the series with a copy of a1 array

Before modifying a1[0], data[0]=  100
a1[0] =  -187
After modifying a1[0], data[0]=  100
```

DataFrame

The pandas Series is a 1-dimensional labelled structure. DataFrame is a 2-dimensional labelled structure. As with a Series, it is built on top of NumPy arrays to take advantage of faster processing (compared to Python Lists and Dictionaries). DataFrames are 2-D arrays with attached row and columns labels, and generally with different data types across columns and/or missing data. A DataFrame is very similar to a spreadsheet.

In the table, you can see the structure of a Data Frame.

Data Frame

	Columns			
index	**city**	**state**	**pop_density**	**unemployed_rate**
0	Sydney	New South Wales	4627345	4.3%
1	Melbourne	Victoria	4246375	4.9%
2	Brisbane	Queensland	2189878	NaN
3	Adelaide	South Australia	1225235	7.3%
4	Canberra	Australian Capital Territory	367752	3.5%
5	Perth	Western Australia	1896548	NaN
6	Hobart	Tasmania	NaN	6.4%
7	Darwin	Northern Territory	NaN	6.1%

Constructing Data Frame Objects

You can create a DataFrame from lists, dictionaries, arrays, Series or combinations of these. The most common way to create a new Data Frame is using a `DataFrame()` constructor and passing a dictionary as a parameter, but we'll look at other options below. As a DataFrame is an object, it has attributes and methods.

View examples

Let's create a DataFrame using arrays and a list of Series.

In [15]:

```
# Create a DataFrame from two-dimensional array

df1 = pd.DataFrame(np.array([[6.5, 90.3], [3.6, 3.2]]))
df1
```

Out[15]:

	0	**1**
0	6.5	90.3
1	3.6	3.2

In [16]:

```
# Creating a DataFrame from a List of Series objects

df2 = pd.DataFrame([pd.Series(np.arange(1,6)),#First row (series 1 to 5) with 5 elements
                    pd.Series(np.arange(6,11)),#Second row (series 6 to 10) with 5 elements
                    pd.Series(np.arange(11,16))])# Third row (series 11 to 15) with 5 elements
df2
```

Out[16]:

	0	1	2	3	4
0	1	2	3	4	5
1	6	7	8	9	10
2	11	12	13	14	15

Note: The rows and columns are labelled with numbers. When you don't specify labels for the rows and/or columns, by default Pandas creates a range of the integers starting from 0 as the labels.

As with NumPy arrays, you can use the shape attribute to get the dimensions of a DataFrame.

In [17]:

```
# Getting the shape of df2

df2.shape
```

Out[17]:

```
(3, 5) # => 3 rows and 5 columns
```

A DataFrame can be created from a dictionary, Series and even the combination of these two. The dictionary keys provide the column labels for the DataFrame.

In [18]:

```
# Create a DataFrame using 3 Series and a Dictionary

s1 = pd.Series(np.arange(1,6))
s2 = pd.Series(np.arange(6,11))
s3 = pd.Series(np.arange(11,16))
df2 = pd.DataFrame({'C1':s1, 'C2': s2, 'C3':s3})
df2

# Note how the Dictionary keys become the column labels
```

Out[18]:

	C1	C2	C3
0	1	6	11
1	2	7	12
2	3	8	13
3	4	9	14
4	5	10	15

In [19]:

```
# Create a DataFrame from a Dictionary with List values

data_dict = {'product' : ['ball','pen','pencil','paper','mug'],
            'color' : ['blue','green','yellow','red','white'],
            'price' : [1.2,1.0,0.6,0.9,1.7],
            'price_discount': [1.1,0.8, 0.5,0.8,1.5]}
df3 = pd.DataFrame(data_dict)
df3

#Again note how the keys become the column labels and the values become the data
```

Out[19]:

	product	color	price	price_discount
0	ball	blue	1.2	1.1
1	pen	green	1.0	0.8
2	pencil	yellow	0.6	0.5
3	paper	red	0.9	0.8
4	mug	white	1.7	1.5

If the labels for the index (row) are not explicitly specified in your DataFrame, pandas, by default, assigns a numeric sequence starting from 0. As you saw earlier, if you want to assign labels to the indexes (rows) you can include the index parameter when you create your DataFrame object to assign the labels you want.

In [20]:

```
# Assign labels to the index (rows)

df3 = pd.DataFrame(data_dict, index=['one','two','three','four','five'])
df3
```

Out[20]:

	product	color	price	price_discount
one	ball	blue	1.2	1.1
two	pen	green	1.0	0.8
three	pencil	yellow	0.6	0.5
four	paper	red	0.9	0.8
five	mug	white	1.7	1.5

A DataFrame performs automatic alignment of the data for each Series in the dictionary. This means that each value will appear in the correct column. To achieve this, the Series or arrays you are using to create the frame should have the same size.

What happens if your Series or arrays don't have the same size? No worries! Pandas will automatically fill these spaces with not a number (NaN) values.

In [21]:

```
# Create a DataFrame using 4 Series with different sizes and a dictionary for column labels

s1 = pd.Series(np.arange(1,6)) # 5 elements
s2 = pd.Series(np.arange(6,9)) # 3 elements
s3 = pd.Series(np.arange(11,16)) # 5 elements
s4 = pd.Series((np.arange(16,20)))# 4 elements

df2 = pd.DataFrame({'C1':s1, 'C2': s2, 'C3':s3, 'C4':s4})
df2
```

Out[21]:

	C1	C2	C3	C4
0	1	6.0	11	16.0
1	2	7.0	12	17.0
2	3	8.0	13	18.0
3	4	NaN	14	19.0
4	5	NaN	15	NaN

Did You Know?

Pandas stands for "Python Data Analysis Library". According to the Wikipedia page on pandas, "the name is derived from the term "panel data (https://en.wikipedia.org/wiki/Panel_data)", an econometrics (https://en.wikipedia.org/wiki/Econometrics) term for multidimensional structured data sets."

pandas

Readings

1. Getting started guide (https://pandas.pydata.org/docs/getting_started/index.html#getting-started) at pandas.pydata.org.
2. Pandas User Guide (https://pandas.pydata.org/docs/user_guide/index.html) at pandas.pydata.org.
3. Nelli, F. (2018). The pandas Library—An Introduction (https://rdcu.be/cP3E5) and pandas: Reading and Writing Data (https://rdcu.be/cP3FJ) in: Python Data Analytics. Apress, Berkeley, CA

Next Steps

Next up, we will look at operations on DataFrames.

B2 Topic 5: DataFrame Operations

Learning a Python Library

Within the scope of this course, we wish to introduce you to pandas and communicate the basic usage of Series and DataFrames. The practical and workshop aim to develop your skills beyond the necessarily brief presentation of material here relying on a "secret" weapon required to use any library of moderate complexity: the Application Programming Interface (API).

The API, **presented in this pandas API reference (https://pandas.pydata.org/docs/reference/index.html)**, is essential to using the library for all tasks beyond simply creating basic Series or DataFrame objects. You must take the time to at least briefly familiarise yourself with the description of:

- **Series (https://pandas.pydata.org/docs/reference/series.html)** including the attributes of a Series object and the methods that can be invoked on an object of this class, and
- **DataFrame (https://pandas.pydata.org/docs/reference/frame.html)** including constructor, attributes and methods.

It is **absolutely not necessary that you memorise** all of these aspects of these classes! However, when asked to calculate the median of a series for example, you should first turn to the API to provide the necessary details to solve the problem. You **will** be provided access to the pandas (and NumPy) API references during the Module Test and the practicals should provide many opportunities to search for solutions in this documentation.

Per the suggested readings on the previous page, you should also take the opportunity to look at the pandas **Getting Started (https://pandas.pydata.org/docs/getting_started/index.html)** and **User Guide (https://pandas.pydata.org/docs/user_guide/index.html)** to tackle the practical assignment.

Using library APIs is a foundational "meta" skill of working to solve non-trivial problems with Python.

Missing Values

You can define a specific index in a Series, but what happens if you have more indexes (rows) than you have data? For example, with the weather data, perhaps wind speed failed to be recorded during an observation. You can't just fill these with 0s as the wind speed could actually be 0 so recording a non-observed wind speed as 0 would affect calculations you did about wind speed (such as mean wind speeds).

Pandas uses the `NaN` value to fill these spaces.

Note

Pandas allows you to explicitly define Not a Number (NaN) values and add them to a Series or a DataFrame.

Tip: Try to be consistent for all the data that you define as missing data. We suggest you use `np.nan` in all the cases.

In [36]:

```
# Creating an index and NaN values

data  =  {'apple': 1.5, 'blueberry': 3.5, 'banana': 2.99, 'orange': 4.3, 'mandarin':np.nan}

# Create a series with the dictionary

series1 = pd.Series(data)
print('series1 =')
print(series1)

# Create another series with more indexes than series1

fruits = ['apple','blueberry','banana', 'orange', 'mandarine', 'rockmelon', 'strawberry']

# Create a series using the series1 data and the additional indexes
# series2 has more indexes than the data provided.  There are missing values for some indexes

series2 = pd.Series(series1, index=fruits)
print('series2 =')
print(series2)
```

```
series1 =
apple        1.50
blueberry    3.50
banana       2.99
orange       4.30
mandarin      NaN
dtype: float64

series2 =
apple         1.50
blueberry     3.50
banana        2.99
orange        4.30
mandarin       NaN
rockmelon      NaN
strawberry     NaN
dtype: float64
```

Note

Note that all the missing values have automatically been filled in with `NaN`.

You can find out how many NaNs you have in your Series by checking if the values are `null` with the `isnull( )` and `notnull( )` functions. Let's see what happen with NaNs.

In [37]:

```
# isnull() function returns true when is a null value or NaN
series1.isnull()

# mandarin is null because it has NaN for value
```

Out[37]:

```
apple        False
blueberry    False
banana       False
orange       False
mandarin      True
dtype: bool
```

In [38]:

```
# notnull() function return true when the value is NoT a null value or NaN
series2.notnull()

# apple, blueberry, banana and orange are True because they all have a value.
# However, mandarin, rockmelon and strawberry are False because they have NaN values
```

Out[38]:

```
apple          True
blueberry      True
banana         True
orange         True
mandarin      False
rockmelon     False
strawberry    False
dtype: bool
```

Unique Values, Counts and NaNs

Sometimes in a Series, you have some repeated values, so you may want to know the unique values in your Series. To get the unique values in a Series, you use the `unique()` function.

You may also want to know how many times a value is repeated or its occurrences. If so, you use `value_counts()` function.

Finally, with the `isin()` function you can determine if values are contained in the Series.

In [30]:

```
# Getting the unique values

colors = pd.Series(['yellow', 'white', 'black', 'black', 'white', 'red', 'yellow', 'red', 'purple','b
lack', 'black', 'red'])
print('colours =')
print(colors)

print('unique values in colours \n', colors.unique())
```

```
colours =
0     yellow
1      white
2      black
3      black
4      white
5        red
6     yellow
7        red
8     purple
9      black
10     black
11       red
dtype: object

unique values in colours
 ['yellow' 'white' 'black' 'red' 'purple']
```

In [31]:

```
# Counting the occurrences in a series

colors.value_counts()
```

Out[31]:

```
black     4
red       3
yellow    2
white     2
purple    1
dtype: int64
```

In [32]:

```
# Evaluating the membership of the values

colors.isin(['white','green'])

# Note this function returns a a series with True for the elements
# that match or False for those that don't match
```

Out[32]:

```
0     False
1      True
2     False
3     False
4      True
5     False
6     False
7     False
8     False
9     False
10    False
11    False
dtype: bool
```

In [33]:

```
# Filtering the data using the boolean values that the isin() function returns

colors[colors.isin(['white','red'])]
```

Out[33]:

```
1     white
4     white
5       red
```

```
7       red
11      red
dtype: object
```

You can count how many data values are valid or missing by combining `isnull()` or `notnull()` with `sum()`.

`<seriesName>.isnull().sum()`

If you want to filter the data to only see valid data or only missing data, you can use `isnull()` or `notnull()` as conditions on the index.

`<seriesName>[<seriesName>.notnull()]` (you'll get the `notnull` (i.e. valid) data).

In [39]:

```
# Counting notnull and null data

print('Number of null data values in series2 is',series2.isnull().sum())
print('Number of valid data values in series2 is',series2.notnull().sum())
```

```
Number of null data values in series2 is 3
Number of valid data values in series2 is 4
```

In [40]:

```
# Filtering NOT null data

print('The valid data in series2 are')
print(series2[series2.notnull()])
print('Missing data values in series2 are')
print(series2[series2.isnull()])
```

```
The valid data is series2 are
apple        1.50
blueberry    3.50
banana       2.99
orange       4.30
dtype: float64
Missing data values in series2 are
mandarin     NaN
rockmelon    NaN
strawberry   NaN
dtype: float64
```

Setting Column Labels

You can specify the labels of the columns when you construct the DataFrame object by adding the arguments: `columns = [<array_of_names>]`.

You can get the names of the columns in a DataFrame using the columns attribute.

You can also change the names of the columns by assigning values to the columns attribute.

Remember that the column names are a pandas index, so you can access the label for a specific column by using its position.

In [45]:

```
# Setting names to the columns

df1 = pd.DataFrame(np.array([[6.5, 90.3], [3.6, 3.2]]), columns = ['col1','col2'])
df1
```

Out[45]:

	col1	col2
0	6.5	90.3
1	3.6	3.2

In [46]:

```
# Getting the names of the columns using the DataFrame columns attribute

print('name of the columns ',df1.columns)
```

```
name of the columns  Index(['col1', 'col2'], dtype='object')
```

In [47]:

```
# Getting the names of a specific columns using the index

print('column 1 label:', df1.columns[0])
print('column 2 label:', df1.columns[1])
```

```
column 1 label: col1
column 2 label: col2
```

In [48]:

```
# Setting new names for the columns using the DataFrame columns attribute

df1.columns = ['c1','c2']
print('column 1 label:', df1.columns[0])
print('column 2 label:', df1.columns[1])
df1
```

```
column 1 label: c1
column 2 label: c2
```

Out[48]:

	c1	c2
0	6.5	90.3
1	3.6	3.2

Setting Row (Index) Labels

Index (row) labels work similarly to columns. Just remember that pandas DataFrames call the row, *index*.

You can assign index abels when you are constructing your DataFrame object by adding the arguments: `index = [ < array_of_names > ]`.

You can access the names of the indexes (rows) with the Data Frame index attribute.

In [49]:

```
# Create a DataFrame with labels in the indexes(rows)

df1 = pd.DataFrame(np.array([[6.5, 90.3], [3.6, 3.2]]), columns = ['c1','c2'], index=['row1','row2'])
df1
```

Out[49]:

	c1	c2
row1	6.5	90.3
row2	3.6	3.2

In [50]:

```
# Getting the labels of the index using the index attribute

df1.index
```

Out[50]:

```
Index(['row1', 'row2'], dtype='object')
```

In [51]:

```
# Getting the label of a specific index (row) by the position

print('row 1 label:', df1.index[0])
print('row 2 label:' ,df1.index[1])
```

```
row 1 label: row1
row 2 label: row2
```

In [52]:

```
# Rename the index(rows) using the index attribute

df1.index = ['r1','r2']
print('row 1 label:', df1.index[0])
print('row 2 label:' ,df1.index[1])
df1
```

```
row 1 label: r1
row 2 label: r2
```

Out[52]:

	c1	c2
r1	6.5	90.3
r2	3.6	3.2

Setting a Column as an Index

You may have found while working with the data that it would have been easier to have the city names as the indexes (row labels) rather than numbers. If this suits the way you are working with your data, you can do this using the `set_index()` function.

Keep in mind that if you are going to do this, the index values must all be unique. This wouldn't work if you have two Sydney indexes!

View example

In [81]:

```
# Change the index labels
# set_index takes the column(s) that you want to use for the indexes

df_new_pop = df_pop.set_index('cities')
print(df_pop)
print(df_new_pop)
```

```
        cities      area  density                         state  \
0       Sydney  12368.00  4627345               New South Wales   
1    Melbourne   9990.00  4246375                      Victoria   
2     Brisbane  15826.00  2189878                    Queensland   
3        Perth   6418.00  1896548             Western Australia   
4     Adelaide   3258.00  1225235               South Australia   
5   Gold Coast    414.13   591473                    Queensland   
6     Canberra    814.20   367752  Australian Capital Territory   
7    Newcastle    261.80   308308               New South Wales   
8   Wollongong    714.00   292190               New South Wales   
9   Logan City    957.00   282673                    Queensland   
10      Hobart   1696.00   218000                      Tasmania   

    unemployed_rate  min_Temp  max_Temp  
0               4.3      7.50     38.64  
1               4.9      9.80     20.53  
2               NaN     10.93     44.11  
3               NaN     16.81     22.95  
4               7.3      4.34     26.37  
5               6.4      4.47     24.97  
6               3.5      4.24     27.83  
7               4.3      4.68     41.36  
8               4.3      2.05     25.57  
9               6.4     18.74     23.62  
10              6.2     12.04     29.19  
                area  density                         state  unemployed_rate  \
cities                                                                         
Sydney      12368.00  4627345               New South Wales              4.3   
Melbourne    9990.00  4246375                      Victoria              4.9   
Brisbane    15826.00  2189878                    Queensland              NaN   
Perth        6418.00  1896548             Western Australia              NaN   
Adelaide     3258.00  1225235               South Australia              7.3   
Gold Coast    414.13   591473                    Queensland              6.4   
```

```
Canberra      814.20   367752  Australian Capital Territory             3.5
Newcastle     261.80   308308               New South Wales             4.3
Wollongong    714.00   292190               New South Wales             4.3
Logan City    957.00   282673                    Queensland             6.4
Hobart       1696.00   218000                      Tasmania             6.2

            min_Temp  max_Temp
cities
Sydney          7.50     38.64
Melbourne       9.80     20.53
Brisbane       10.93     44.11
Perth          16.81     22.95
Adelaide        4.34     26.37
Gold Coast      4.47     24.97
Canberra        4.24     27.83
Newcastle       4.68     41.36
Wollongong      2.05     25.57
Logan City     18.74     23.62
Hobart         12.04     29.19
```

Selecting Data

Just like Python (Lists, Dictionaries) and NumPy (1-dimensional array, 2-dimensional array) you can select the data you want from your DataFrame by using attributes such as values (for all of the data) or by using indexes (labels) to select specific data.

Note how `np.nan` has been used to fill missing values.

In [58]:

```python
# Create a DataFrame containing population data to use in selecting data examples

# Create DataFrame using Series of population density, state, unemployment rate and city names.
s_city = pd.Series(['Sydney','Melbourne','Brisbane','Perth','Adelaide','Gold Coast','Canberra','Newcastle','Wollongong','Logan City'])
s_density = pd.Series([4627345, 4246375, 2189878, 1896548,1225235, 591473, 367752, 308308, 292190, 282673])
s_state = pd.Series(['New South Wales', 'Victoria','Queensland', 'Western Australia','South Australia',
                     'Queensland','Australian Capital Territory','New South Wales','New South Wales','Queensland'])
s_unemployed_rate = pd.Series([4.3, 4.9, np.nan, np.nan, 7.3, 6.4, 3.5, 4.3, 4.3, 6.4])

df_pop = pd.DataFrame({'cities':s_city, 'density':s_density, 'state':s_state, 'unemployed_rate':s_unemployed_rate})
df_pop
```

Out[58]:

	cities	density	state	unemployed_rate
0	Sydney	4627345	New South Wales	4.3
1	Melbourne	4246375	Victoria	4.9
2	Brisbane	2189878	Queensland	NaN
3	Perth	1896548	Western Australia	NaN
4	Adelaide	1225235	South Australia	7.3

	cities	density	state	unemployed_rate
5	Gold Coast	591473	Queensland	6.4
6	Canberra	367752	Australian Capital Territory	3.5
7	Newcastle	308308	New South Wales	4.3
8	Wollongong	292190	New South Wales	4.3
9	Logan City	282673	Queensland	6.4

Now, if you want to select *all* the data as a 2-dimensional array, use the values attribute.

If you want to the values of one column, use:

`<DataFrame_name>.<name_column>` or
`<DataFrame_name>['<name_column>']`

In [59]:

```
# values property gives all data as an array

df_pop.values
```

Out[59]:

```
array([['Sydney', 4627345, 'New South Wales', 4.3],
       ['Melbourne', 4246375, 'Victoria', 4.9],
       ['Brisbane', 2189878, 'Queensland', nan],
       ['Perth', 1896548, 'Western Australia', nan],
       ['Adelaide', 1225235, 'South Australia', 7.3],
       ['Gold Coast', 591473, 'Queensland', 6.4],
       ['Canberra', 367752, 'Australian Capital Territory', 3.5],
       ['Newcastle', 308308, 'New South Wales', 4.3],
       ['Wollongong', 292190, 'New South Wales', 4.3],
       ['Logan City', 282673, 'Queensland', 6.4]], dtype=object)
```

In [60]:

```
# Selecting only cities

df_pop.cities
```

Out[60]:

```
0        Sydney
1     Melbourne
2      Brisbane
3         Perth
4      Adelaide
5    Gold Coast
6      Canberra
7     Newcastle
8    Wollongong
9    Logan City
Name: cities, dtype: object
```

In [61]:

```
# Selecting only unemployment rate

df_pop['unemployed_rate']
```

Out[61]:

```
0    4.3
1    4.9
2    NaN
3    NaN
4    7.3
5    6.4
6    3.5
7    4.3
8    4.3
9    6.4
Name: unemployed_rate, dtype: float64
```

In [62]:

```
# With <DataFrame_Name>. and tab key you can see the properties and methods of the series
# Also the name of the columns as well.  Try this in your Jupyter notebook.
df_pop.density
```

Out[62]:

```
0    4627345
1    4246375
2    2189878
3    1896548
4    1225235
5     591473
6     367752
7     308308
8     292190
9     282673
Name: density, dtype: int64
```

Slicing a DataFrame

Slicing with [] is similar to slicing a 2-dimensional NumPy array by rows (**C | Week 1 : 2D Arrays (https://myuni.adelaide.edu.au/courses/86136/pages/c-%7C-week-n+1-accessing-sections-of-an-array?module_item_id=2610789)**).

`<DataFrame_name>[start: stop: step]`

- **start** is a numeric value included where the selection will start and it is optional. The default start value is 0.
- **stop** is the end of the section. It is a numeric value not included in the interval.
- **step** is the step size between the values. it is a numeric and optional value. The default step size is 1.

View examples

In [63]:

```
# Getting the first two rows

df_pop[:2]
```

Out[63]:

	cities	density	state	unemployed_rate
0	Sydney	4627345	New South Wales	4.3
1	Melbourne	4246375	Victoria	4.9

In [64]:

```
# Invert the rows

df_pop[::-1]
```

Out[64]:

	cities	density	state	unemployed_rate
9	Logan City	282673	Queensland	6.4
8	Wollongong	292190	New South Wales	4.3
7	Newcastle	308308	New South Wales	4.3
6	Canberra	367752	Australian Capital Territory	3.5
5	Gold Coast	591473	Queensland	6.4
4	Adelaide	1225235	South Australia	7.3
3	Perth	1896548	Western Australia	NaN
2	Brisbane	2189878	Queensland	NaN
1	Melbourne	4246375	Victoria	4.9
0	Sydney	4627345	New South Wales	4.3

In [65]:

```
# Get the rows in the indexes 3 to 4

df_pop[3:5]
```

Out[65]:

	cities	density	state	unemployed_rate
3	Perth	1896548	Western Australia	NaN
4	Adelaide	1225235	South Australia	7.3

Slicing with Conditions

Another way you can slice is by using a condition inside of loc:

`<DataFrame>.loc[<condition>, [<col1,..., coln>]]`

You can also use more than one condition by joining conditions with:

Note

When using conditions in pandas and NumPy, "**&**" is always used for "and" and "**|**" for "or".

In [82]:

```
# Example of Slicing with a condition
# Getting the cities and population density columns for population densities bigger than 2 millions

df_pop.loc[df_pop.density > 2000000,['cities','density']]
```

Out[82]:

	cities	density
0	Sydney	4627345
1	Melbourne	4246375
2	Brisbane	2189878

In [83]:

```
# Example of Slicing with combined conditions
# Get all of the information for rows where the unemployment rate is between 4.0 and 4.5
df_pop.loc[(df_pop.unemployed_rate > 4.0) & (df_pop.unemployed_rate > 4.5)]
```

Out[83]:

	cities	area	density	state	unemployed_rate	min_Temp	max_Temp
0	Sydney	12368.0	4627345	New South Wales	4.3	7.50	38.64
7	Newcastle	261.8	308308	New South Wales	4.3	4.68	41.36
8	Wollongong	714.0	292190	New South Wales	4.3	2.05	25.57

Readings

B2 Topic 5: Introduction to Visualisation Using Pandas

Reminder

The following information is provided for your interest and is not an examinable element of this Section.

Note

NOTE: The Pandas library does *not* draw the plots; instead, pandas tells Matplotlib how to draw the plots using Pandas data, selecting the Series for plots, labelling axis, creating legends, and determining appropriate visual elements, such as choosing colours and markers. This allows you to write a small amount of code to create amazing data visualisations.

Open a new Jupyter notebook so you're ready to start plotting from Series and Data Frames!

As usual, start by importing the libraries you need. We'll need NumPy to generate our data, pandas to store the data as a Series or DataFrame, as well as plotting the data and pyplot to set display aspects of our plots, such as the axes ranges and plot styles.

In [2]:

```
#libraries
import numpy as np #manipulation of arrays
import pandas as pd #manipulation of data as DataFrame or Series
import matplotlib.pyplot as plt # setting style and plot attributes such as the range for x and y axes

# this is a 'magic function' (Python's official word for it!)
# it makes your plots appear in your notebook
%matplotlib inline

#We'll use the white grid style for our plots
plt.style.use('seaborn-whitegrid')
```

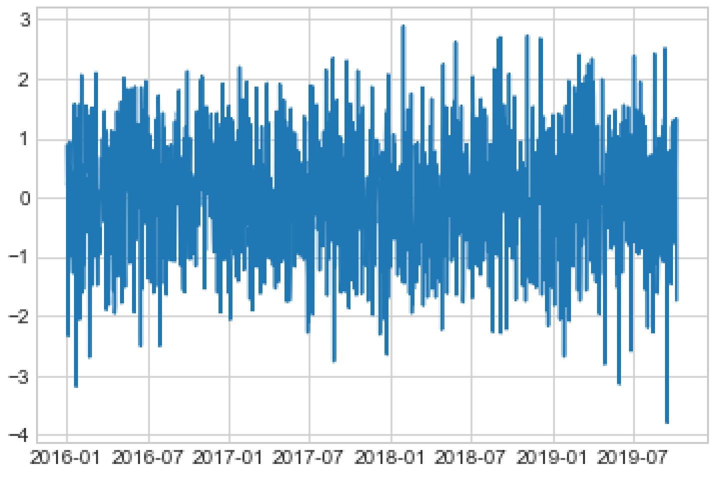
Plotting a Pandas Series

In [4]:

```
#Same plot but using directly Matplotlib plt.plot()
plt.plot(timeSeries)
#The generated plot is slightly different in the x-axes legend
#Pandas will generally work out more visually appealing settings
```

Out[4]:

```
[<matplotlib.lines.Line2D at 0x1192f4320>]
```



Plotting a Pandas DataFrame

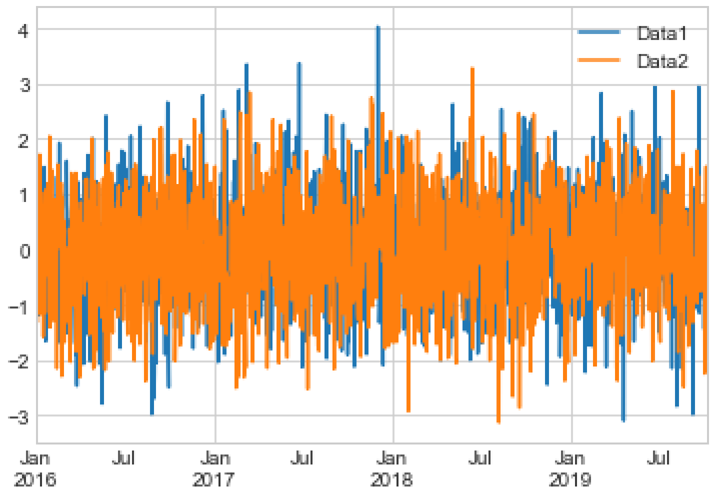
You can use the same `plot()` function to plot the data in a Data Frame. Of course, a Data Frame is generally used when we have more than one column of data, so let's start by creating a Data Frame with two columns of data.

In [6]:

```
# plot it!
df.plot()
```

Out[6]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x106e6f908>
```



Labelling and changing appearance of plots

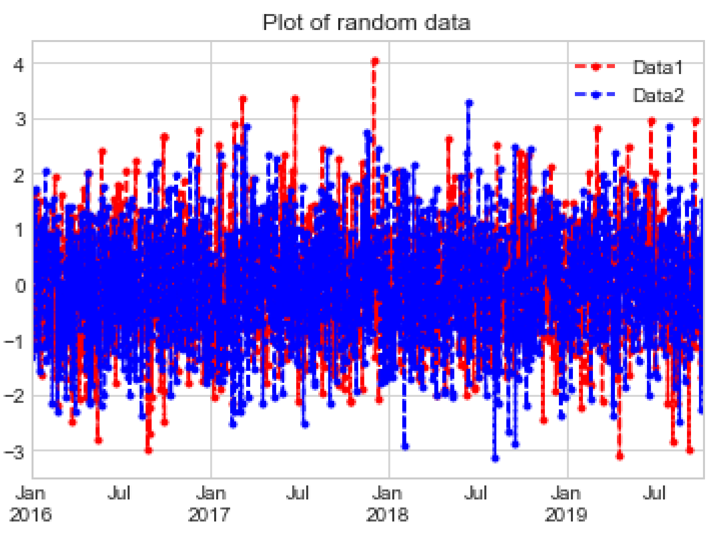
You can add a title, x-label, y-label and legend to your Pandas plots and change the colour and marker of the line, just like you did in Matplotlib. By setting the values for the attributes when you call plot:

- color
- marker
- linestyle
- title

You can find a full list of the options in the notes section at [Matplotlib (https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.plot.html)](https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.plot.html) .

In [7]:

```
# set style attributes in plot()
randomPlot = df.plot(color=['r','b'], marker='.', linestyle='--', title='Plot of random data')
```



You can set x and y labels and the legends using:

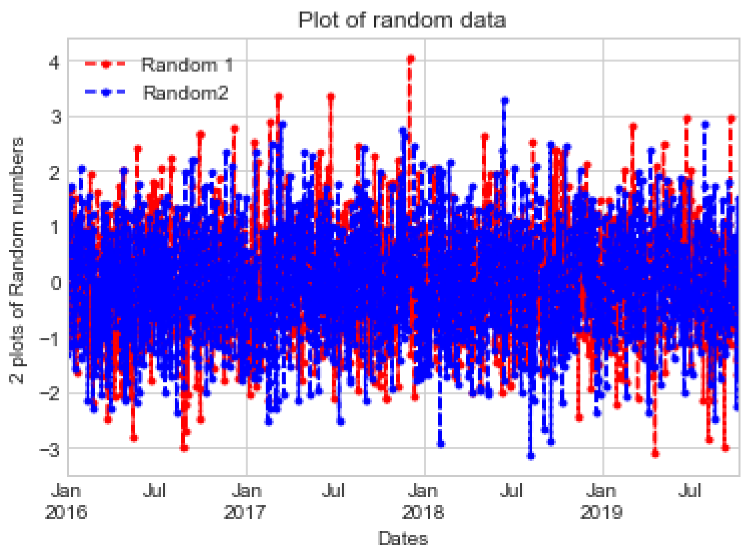
- set_xlabel
- set_ylabel
- legend

In [8]:

```
# The x and y labels are part of the axes of the plot
# the axes are returned by the plot() function
axes= df.plot(color=['r','b'], marker='.', linestyle='--', title='Plot of random data')

# set the label attributes of the axes
axes.set_xlabel('Dates')
axes.set_ylabel('2 plots of Random numbers')

# change the legend labels (default set to the column names: Data1 and Data2)
# move the legend to the top left
axes.legend(['Random 1', 'Random2'], loc='upper left')
```

Out[8]:

```
<matplotlib.legend.Legend at 0x118d52e48>
```



Summary

Pandas plots share a lot of similarities with Matplotlib.

Plots have default attributes that can be set when creating the plot including:

- line colour
- line style
- marker style
- titles

You can set x and y labels and the legends using:

- `set_xlabel`
- `set_ylabel`
- `legend`

Like Matplotlib, plots are displayed in figures. The figures can hold one plot or multiple subplots.

The main difference to Matplotlib is you are plotting data in Series and Data Frames, instead of NumPy arrays and Pandas uses column names to label your data automatically.

The pandas library has its own methods for visualising data. Just as pandas is built on top of NumPy to provide you with more flexibility in how you store your data (allowing different types to be mixed and allowing you to index using labels rather than just positions), so too are pandas plotting functions built on top of Matplotlib, but are specifically designed to work with the pandas classes: Series and Data Frames. You will want to use pandas plotting directly most of the time. However, it is still often useful to go directly to Matplotlib to set up your data for the final plot output.

To plot a Series or DataFrame, the same method `plot()` is used that you previously used to plot a NumPy array. The `plot()` method that is used will depend on the object containing your data. If your object is a numpy array you will get the behaviour you saw with **Matplotlib ($CANVAS_COURSE_REFERENCE$/modules/items/g23ac4ff966c32f24abbb5228de39de2b)**. If your object is a Series or a DataFrame, you will get the `plot( )` behaviour covered in this section. Make sure your data is stored in a Series or DataFrame to get the Pandas plot behaviour!

As an example Series, we're going to use a Time Series. Of course, you could use other Series' that have an index other than time; the plotting will happen the same way. But Time Series is a common type of Series data.

For our Time Series data, we will use random numbers between 0 and 1. This is just an example after all!

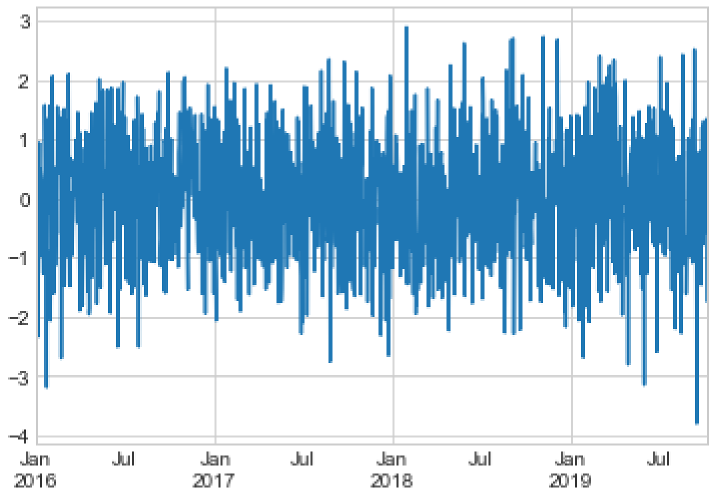
You may find it helpful to have your pandas Jupyter notebook open so that you can refer back to your Time Series notes while going through this example.

In [3]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

%matplotlib inline

# create random data for our Time Series
# the seed to generate the same random numbers as in this example
seedValue =20
np.random.seed(seedValue)

# Create a range of dates that will become the index
dateRange = pd.date_range('2016-01-01', '2019-10-07')

# Generate the random data values (1 for each of the dates in the date range)
numberGen = len(dateRange)#1376 numbers
data = np.random.randn(numberGen)

#Create the Time Series (a Series with a DateTimeIndex)
timeSeries = pd.Series(data, index = dateRange)
timeSeries.plot()
```

Out[3]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x118d655f8>
```



In [5]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

%matplotlib inline

#Data Frame with two columns of random numbers
indexValues = pd.date_range('2016-01-01', '2019-10-07')
numberGen = len(indexValues)

data1 = pd.Series(np.random.randn(numberGen), index=indexValues)
data2 = pd.Series(np.random.randn(numberGen), index=indexValues)

#Create the data frame using dictionary format (label becomes column labels, data becomes rows)
df = pd.DataFrame({'Data1':data1, 'Data2':data2})

# use head(n) to only see first n rows.  Otherwise it will display all of the data
df.head(10)
```

Out[5]:

	Data1	Data2
2016-01-01	0.415551	-0.068035
2016-01-02	-0.614132	-1.316433
2016-01-03	-0.172212	-0.660858
2016-01-04	-0.936464	1.355537
2016-01-05	-1.183777	0.966524
2016-01-06	0.852587	0.541251
2016-01-07	1.643641	0.122672
2016-01-08	0.731546	1.718812
2016-01-09	0.969932	1.477371
2016-01-10	-0.935829	-1.163298

Next Steps

Let's move on to our Workshop on pandas.

B2 Topic 5: Workshop Activities (pandas)

Reflective Learning

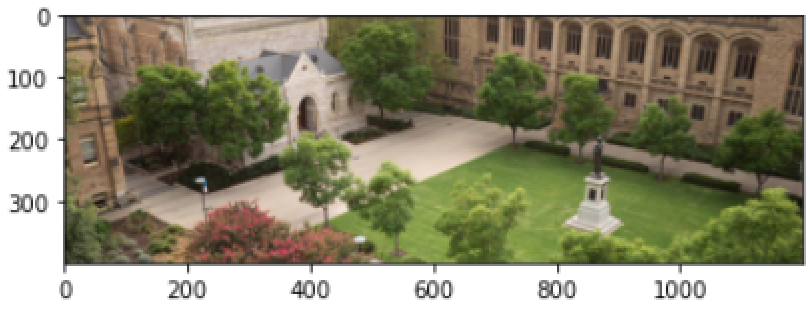
Before the workshop, take some time to reflect on our work with NumPy earlier this week. Begin by installing the library **scikit-image** in your development environment. This should be as simple as:

```
pip install scikit-image
```

Open a Jupyter Notebook and add this code fragment:

```
from skimage import io
import matplotlib.pyplot as plot
import requests
import numpy as np

URL = "https://www.adelaide.edu.au/campuses/sites/default/files/styles/ua_image_full_width/public/medi
a/images/2020-03/nth-tce-campus-08063_uoa-1440x500.jpg?h=57013b68&itok=v2e3CqAR"
response = requests.get(URL)
open("uofa-banner.png", "wb").write(response.content)
photo = io.imread('uofa-banner.png')

plot.imshow(photo)
type(photo)
```

This code uses the `request` library to download an image from the University website (and saves it locally). As you'll see, the photo is a NumPy n-dimensional array.



Use `photo.shape` to examine the dimensions of the array. The third dimension is for 3 bytes of Red, Green, Blue data. Now that we have our data, we can manipulate the array and see the results. Take time to:

- flip the image vertically,
- flip the image horizontally,
- slice the image in half horizontally and show just the left or right half, and
- show just the statue - note that you can use the axis labels to estimate the pixel ranges on the full size image.

All of these actions should be of the form:

```
plot.imshow(photo[##slice##])
```

If you are struggling, you can download **this notebook (https://myuni.adelaide.edu.au/courses/86136/files/12213406?wrap=1)** **(https://myuni.adelaide.edu.au/courses/86136/files/12213406/download?download_frd=1)** with solutions.

"Phone a Friend": API References

Both NumPy and pandas are Python libraries of sophistication and power. While the course content presented this week serves to introduce some of the key classes for arrays, series and data manipulation across both the libraries, it is essential that you familiarise yourself with the Application Programming Interface references that define constructors, attributes and methods of the key classes in these two libraries:

- **NumPy API reference (https://numpy.org/doc/stable/reference/)**, and
- **pandas API reference (https://pandas.pydata.org/docs/reference/index.html)**.

As noted **here (https://myuni.adelaide.edu.au/courses/86136/pages/c-%7C-week-2-dataframe-operations)**, you **do not need to memorise the detail** of these APIs, but you should be sufficiently familiar with them to understand that, when asked (for example) to find the *sum* of a series or the *average* value, that you would use the API reference to help solve the problem.

You **will** be allowed to reference these API's during the module test and you **will** find them invaluable to complete the practicals this week.

Let's Try pandas

1. In your Jupyter Notebook, write code to create four DataFrames that include the following features.
   - Create a DataFrame using a two-dimensional array 5x4 (5 rows and 4 columns)
   - Create a DataFrame using 3 Series with the same size.
   - Create a DataFrame using 5 Series with different sizes
   - Create a DataFrame with labelled rows (index) and columns using a dictionary.

View solution

```
import pandas as pd
import numpy as np

# 5x4 array
# could also use lists rather than arange, but arange, saves typing :)

df1 = pd.DataFrame(np.array([np.arange(1,5),np.arange(5,9), np.arange(9,13),np.arange(13,1
7),np.arange(17,21)]))
print('Data Frame from 5x4 array')
print(df1)
print()

# 3 series same size
s1 = pd.Series(np.arange(1,5))
s2 = pd.Series(np.arange(5,9))
s3 = pd.Series(np.arange(9,13))
df2 = pd.DataFrame([s1,s2,s3])
print('Data Frame from 3 equal size Series')
print(df2)
print()

# 5 series different sizes
s1 = pd.Series(np.arange(1,5))
s2 = pd.Series(np.arange(5,6))
s3 = pd.Series(np.arange(9,13))
s4 = pd.Series(np.arange(8,13))
s5 = pd.Series(np.arange(1,12))
df3 = pd.DataFrame([s1,s2,s3,s4,s5])
print('Data Frame from 5 different sized Series')
print(df3)
print()

# dictionary for column names
df4 = pd.DataFrame({'col1':[1,2,3,4],'col2':[5,6,7,8], 'col3':[9,10,11,12]}, index=['row
1','row2','row3','row4'])
print('Data Frame from Dictionary with index (row) and column labels')
print(df4)
```

```
DataFrame from 5x4 array
    0   1   2   3
0   1   2   3   4
1   5   6   7   8
2   9  10  11  12
3  13  14  15  16
4  17  18  19  20

DataFrame from 3 equal size Series
   0   1   2   3
0  1   2   3   4
1  5   6   7   8
2  9  10  11  12

DataFrame from 5 different sized Series
    0     1     2     3     4    5    6    7    8     9     10
0  1.0   2.0   3.0   4.0   NaN  NaN  NaN  NaN  NaN   NaN   NaN
1  5.0   NaN   NaN   NaN   NaN  NaN  NaN  NaN  NaN   NaN   NaN
2  9.0  10.0  11.0  12.0   NaN  NaN  NaN  NaN  NaN   NaN   NaN
3  8.0   9.0  10.0  11.0  12.0  NaN  NaN  NaN  NaN   NaN   NaN
4  1.0   2.0   3.0   4.0   5.0  6.0  7.0  8.0  9.0  10.0  11.0

DataFrame from Dictionary with index (row) and column labels
      col1  col2  col3
row1     1     5     9
row2     2     6    10
row3     3     7    11
row4     4     8    12
```

2. In your Jupyter Notebook write code to create a DataFrame from a Series, about any topic (sales, students, etc.). Your DataFrame should have at least six columns and 10 rows. Write

code using `loc()` *and* `iloc()` to modify and slice your data do extract each of the following:

- Invert your DataFrame (the last row should now be the first row)
- Get the third row with all the data in the columns
- Get the third column with all the data in the rows
- Get the last two columns with all the data in the rows
- Get the first two columns with the first five rows
- Get the fourth and fifth columns with the last two rows
- Get every second column for all the rows

View solution

In [2]:

```
import pandas as pd
import numpy as np

# First we create a DataFrame with 6 columns and 11 rows
# Temperature
s_maxTemp = pd.Series(np.round(np.random.uniform(20, 45,11),2))
s_minTemp = pd.Series(np.round(np.random.uniform(-5, 19,11),2))
s_area = pd.Series([12368, 9990, 15826, 6418, 3258, 414.13, 814.2, 261.8, 714, 957, 1696])
s_cities = pd.Series(['Sydney','Melbourne','Brisbane','Perth','Adelaide','Gold Coast','Canbe
rra','Newcastle',
        'Wollongong','Logan City','Hobart'])
s_density = pd.Series([4627345, 4246375, 2189878, 1896548,1225235, 591473, 367752, 308308, 2
92190, 282673, 218000])
s_state = pd.Series(['New South Wales', 'Victoria','Queensland', 'Western Australia','South
Australia',
                     'Queensland','Australian Capital Territory','New South Wales','New Sout
h Wales',
                     'Queensland', 'Tasmania'])

s_unemployed_rate = pd.Series([4.3, 4.9, np.nan, np.nan, 7.3, 6.4, 3.5, 4.3, 4.3, 6.4, 6.2])
df_pop = pd.DataFrame({'cities':s_cities,'area': s_area, 'density': s_density, 'state':s_sta
te,
                       'unemployed_rate':s_unemployed_rate, 'min_Temp':s_minTemp, 'max_Tem
p':s_maxTemp })
df_pop
```

Out[2]:

	cities	area	density	state	unemployed_rate	min_Temp	max_Tem
0	Sydney	12368.00	4627345	New South Wales	4.3	7.50	38.64
1	Melbourne	9990.00	4246375	Victoria	4.9	9.80	20.53
2	Brisbane	15826.00	2189878	Queensland	NaN	10.93	44.11
3	Perth	6418.00	1896548	Western Australia	NaN	16.81	22.95
4	Adelaide	3258.00	1225235	South Australia	7.3	4.34	26.37
5	Gold Coast	414.13	591473	Queensland	6.4	4.47	24.97
6	Canberra	814.20	367752	Australian Capital Territory	3.5	4.24	27.83

	cities	area	density	state	unemployed_rate	min_Temp	max_Temp
7	Newcastle	261.80	308308	New South Wales	4.3	4.68	41.36
8	Wollongong	714.00	292190	New South Wales	4.3	2.05	25.57
9	Logan City	957.00	282673	Queensland	6.4	18.74	23.62
10	Hobart	1696.00	218000	Tasmania	6.2	12.04	29.19

In [3]:

```
# Inverse your DataFrame (the last rows should now be the first rows )

print(df_pop.iloc[::-1])
print(df_pop.loc[::-1])
```

```
        cities      area  density                         state  \
10      Hobart   1696.00   218000                      Tasmania
9   Logan City    957.00   282673                    Queensland
8   Wollongong    714.00   292190               New South Wales
7    Newcastle    261.80   308308               New South Wales
6     Canberra    814.20   367752  Australian Capital Territory
5   Gold Coast    414.13   591473                    Queensland
4     Adelaide   3258.00  1225235               South Australia
3        Perth   6418.00  1896548             Western Australia
2     Brisbane  15826.00  2189878                    Queensland
1    Melbourne   9990.00  4246375                      Victoria
0       Sydney  12368.00  4627345               New South Wales

    unemployed_rate  min_Temp  max_Temp
10              6.2     12.04     29.19
9               6.4     18.74     23.62
8               4.3      2.05     25.57
7               4.3      4.68     41.36
6               3.5      4.24     27.83
5               6.4      4.47     24.97
4               7.3      4.34     26.37
3               NaN     16.81     22.95
2               NaN     10.93     44.11
1               4.9      9.80     20.53
0               4.3      7.50     38.64
        cities      area  density                         state  \
10      Hobart   1696.00   218000                      Tasmania
9   Logan City    957.00   282673                    Queensland
8   Wollongong    714.00   292190               New South Wales
7    Newcastle    261.80   308308               New South Wales
6     Canberra    814.20   367752  Australian Capital Territory
5   Gold Coast    414.13   591473                    Queensland
4     Adelaide   3258.00  1225235               South Australia
3        Perth   6418.00  1896548             Western Australia
2     Brisbane  15826.00  2189878                    Queensland
1    Melbourne   9990.00  4246375                      Victoria
0       Sydney  12368.00  4627345               New South Wales

    unemployed_rate  min_Temp  max_Temp
10              6.2     12.04     29.19
9               6.4     18.74     23.62
8               4.3      2.05     25.57
7               4.3      4.68     41.36
6               3.5      4.24     27.83
5               6.4      4.47     24.97
4               7.3      4.34     26.37
3               NaN     16.81     22.95
2               NaN     10.93     44.11
1               4.9      9.80     20.53
0               4.3      7.50     38.64
```

In [4]:

```
# Get the third row with all the data in the columns

print(df_pop.iloc[2:3])
print(df_pop.loc[2:2])
```

```
     cities     area  density       state  unemployed_rate  min_Temp  max_Temp
2  Brisbane  15826.0  2189878  Queensland              NaN     10.93     44.11
     cities     area  density       state  unemployed_rate  min_Temp  max_Temp
2  Brisbane  15826.0  2189878  Queensland              NaN     10.93     44.11
```

In [5]:

```
# Get the third column with all the data in the rows

print(df_pop.iloc[:,2:3])
print(df_pop.loc[:,'density':'density'])
```

```
    density
0   4627345
1   4246375
2   2189878
3   1896548
4   1225235
5    591473
6    367752
7    308308
8    292190
9    282673
10   218000
    density
0   4627345
1   4246375
2   2189878
3   1896548
4   1225235
5    591473
6    367752
7    308308
8    292190
9    282673
10   218000
```

In [6]:

```
# Get the last two rows with all the data in the columns

print(df_pop.iloc[9:11])
print(df_pop.loc[9:10])
```

```
       cities    area  density       state  unemployed_rate  min_Temp  \
9   Logan City   957.0   282673  Queensland              6.4     18.74
10      Hobart  1696.0   218000    Tasmania              6.2     12.04

    max_Temp
9      23.62
10     29.19
       cities    area  density       state  unemployed_rate  min_Temp  \
9   Logan City   957.0   282673  Queensland              6.4     18.74
10      Hobart  1696.0   218000    Tasmania              6.2     12.04

    max_Temp
```

```
9      23.62
10     29.19
```

In [7]:

```
# Get the first two columns with the first 5 rows

print(df_pop.iloc[0:5,:2])
print(df_pop.loc[0:4,:'area'])
```

```
      cities     area
0     Sydney  12368.0
1  Melbourne   9990.0
2   Brisbane  15826.0
3      Perth   6418.0
4   Adelaide   3258.0
      cities     area
0     Sydney  12368.0
1  Melbourne   9990.0
2   Brisbane  15826.0
3      Perth   6418.0
4   Adelaide   3258.0
```

In [8]:

```
# Get the fourth and fifth columns with the last two rows

print(df_pop.iloc[-2::,3:5])
print(df_pop.loc[9:10:,'state':'unemployed_rate'])
```

```
         state  unemployed_rate
9   Queensland              6.4
10    Tasmania              6.2
         state  unemployed_rate
9   Queensland              6.4
10    Tasmania              6.2
```

In [9]:

```
# Get every two columns with all the rows
print(df_pop.iloc[:,::2])
print(df_pop.loc[:,::2])
```

```
        cities  density  unemployed_rate  max_Temp
0       Sydney  4627345              4.3     38.64
1    Melbourne  4246375              4.9     20.53
2     Brisbane  2189878              NaN     44.11
3        Perth  1896548              NaN     22.95
4     Adelaide  1225235              7.3     26.37
5   Gold Coast   591473              6.4     24.97
6     Canberra   367752              3.5     27.83
7    Newcastle   308308              4.3     41.36
8   Wollongong   292190              4.3     25.57
9   Logan City   282673              6.4     23.62
10      Hobart   218000              6.2     29.19
        cities  density  unemployed_rate  max_Temp
0       Sydney  4627345              4.3     38.64
1    Melbourne  4246375              4.9     20.53
2     Brisbane  2189878              NaN     44.11
3        Perth  1896548              NaN     22.95
4     Adelaide  1225235              7.3     26.37
5   Gold Coast   591473              6.4     24.97
6     Canberra   367752              3.5     27.83
7    Newcastle   308308              4.3     41.36
8   Wollongong   292190              4.3     25.57
```

```
9   Logan City   282673              6.4     23.62
10      Hobart   218000              6.2     29.19
```

3. In your Jupyter Notebook write code to create `df_pop` DataFrame (created in previous cells). Slice this data using the following conditions:

   - All the rows where the density is > 300000 and < 1500000, showing the 'cities', 'density' and 'area' columns.
   - All the rows where the `min_Temp` < 4.5 or `max_Temp` > 35, showing all the columns. This will show you the coldest and hottest cities in Australia (according to the data).

View solution

In [10]:

```
# Solution

import pandas as pd
import numpy as np

# All the rows where the density is > 300000 and < 1500000.
# Show the 'cities','density' and 'area' columns
df_pop.loc[(df_pop.density > 300000) & (df_pop.density < 1500000) ,['cities','density','are
a']]
```

Out[10]:

	cities	density	area
4	Adelaide	1225235	3258.00
5	Gold Coast	591473	414.13
6	Canberra	367752	814.20
7	Newcastle	308308	261.80

In [11]:

```
# All the rows where the min_Temp < 4.5 or max_Temp > 35. Showing all the columns
df_pop.loc[(df_pop.min_Temp < 4.5)|(df_pop.max_Temp > 35)]
```

Out[11]:

	cities	area	density	state	unemployed_rate	min_Temp	max_Temp
0	Sydney	12368.00	4627345	New South Wales	4.3	7.50	38.64
2	Brisbane	15826.00	2189878	Queensland	NaN	10.93	44.11
4	Adelaide	3258.00	1225235	South Australia	7.3	4.34	26.37
5	Gold Coast	414.13	591473	Queensland	6.4	4.47	24.97
6	Canberra	814.20	367752	Australian Capital Territory	3.5	4.24	27.83

	cities	area	density	state	unemployed_rate	min_Temp	max_Temp
7	Newcastle	261.80	308308	New South Wales	4.3	4.68	41.36
8	Wollongong	714.00	292190	New South Wales	4.3	2.05	25.57

? Q&A Session

If you have any questions on NumPy, pandas or other content from this topic, please ask for help in this workshop.

(h (h (h (h (h (h (h (h (h (h (h

B2 Topic 6: Motivation & Approaches for Refactoring

Why refactor working code?

The Zen of Python

Python was developed to be a highly readable, expressive and approachable language that emphasises a simple and uncluttered syntax. Python developers have, as a community, defined an idiomatic style of writing code that is described as “pythonic”. When faced with many choices of how to implement a feature in code, this philosophy is summarised in the *Zen of Python* as "there should be one—and preferably only one—obvious way to do it".

Thus:

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.

The Zen of Python

You can find the full “The Zen of Python” by Tim Peters using a Python "easter egg" with the code fragment:

```
import this
```

What is Refactoring?

Refactoring may be described as:

```
Refactoring (noun): a change made to the internal structure of software to make it easier to understan
d and cheaper to modify without changing its observable behaviour.
```

In other words, refactoring is revising the structure of code that is functionally correct but not clear and concise. Refactoring is similar to proofreading a short essay – once the content is there, you revise, reorder and simplify sentences to make it more readable without changing the overall intent.

Motivations for Refactoring

The goal of refactoring is to improve the quality of the code. Refactoring aims to:

- improve readability and make code easier to understand,
- follow a "pythonic" idiom,
- reduce complexity and simplify the flow of execution,
- localise reused code into functions and modules,
- remove unused code or imports,
- improve the runtime efficiency of code,
- ease the future improvement and long term maintenance of a codebase.

How to Refactor

Refactoring rarely involves re-writing a Python script from scratch. The risk in doing so is that a working piece of software fails to meet its explicit functional requirements - but perhaps also breaks implicit requirements understood only by the original developer.

Instead, refactoring can be thought of as applying a series of basic actions from a list of "micro-refactoring" recipes and so, incrementally, improving the code in such small steps that form changes while function is preserved.

In the next section, we'll look at some of these micro-refactoring techniques.

Readings

1. **This article** (https://realpython.com/python-refactoring/) on the website **realpython.com** (https://realpython.com/) has a great overview of motivations for Python refactoring including how to identify complex code through metrics. It is worthwhile background reading for those interested in understanding the wider topic of code quality.
2. Govindaraj, S. (2015). Ch 3: **Code Smells and Refactoring** (https://ebookcentral.proquest.com/lib/adelaide/reader.action?docID=2039888&ppg=62) . *Test-driven python development*. Packt Publishing, Limited.

Next Steps

Next, we'll examine some concrete patterns for refactoring Python code to make it more idiomatic.

(h (h (h (h (h (h (h (h (h (h

B2 Topic 6: Refactoring Techniques

Refactoring Techniques and Patterns

Refactoring, as we have seen, involves the incremental and iterative improvement of a piece of code to improve readability and ease future development and maintenance. In general, much of this effort involves the application of "micro-refactoring" techniques to solve a set of similar, widely experienced problems with the form of the code - while being careful not to alter the function of the code. These problems are then countered by "recipes" to fix each problem.

Refactoring should always be supported with a "before and after" set of tests to ensure that modification has not introduced functional errors. We will discuss more of this later in the week.

While there is no common, industry-wide set of refactoring recipes, there are identified patterns that generally apply to refactoring. For Python, this best practice is codified in a "style guide" called **PEP8** (https://www.python.org/dev/peps/pep-0008/) that covers whitespace usage, naming conventions, programming recommendations and so forth. But, more generally, good Python should be "pythonic" - a more elusive description of the "right" way to author Python code.

In order to practise refactoring in the workshop and to support the quiz this week, we have laid out nine refactoring patterns for Python code. This is not authoritative or exhaustive but should provide insight into the nature and practice of refactoring to improve code quality.

Pattern #1 - Use readable variable and function names

Scanning over a Python program, it can be common to see variable names that do not relate to the value they refer to. Variable names should be clear, concise and suggest their value. It's quite common to see variable and function names that reflect the whim of the programmer on the day. Variable names can be in pothole_case_naming or camelCase (and _others), but for now, using pothole case is suggested as most readable (and PEP8 recommended) while constant values should be in CAPS.

For variable names:

```python
# Not
mvemjsun = ["Mercury", "Venus", "Earth", "Mars", "etc"]

# Instead refactor:
planets = ["Mercury", "Venus", "Earth", "Mars", "etc"]
```

For the names of functions:

```
# Not
def g (name, msg):
    print("Hello", name + ', ' + msg)

# Instead refactor:
def greet_by_name (name, msg):
    print("Hello", name + ', ' + msg)
```

For declaring constant values:

```
# For a constant used in the program, not
mc = 100

# Instead refactor:
MAXIUMUM_CUSTOMERS = 100
```

Pattern #2 - Use boolean values directly

Making comparisons like 'if a == True' can be replaced by directly using the boolean value.

```
# Not this:
if a == True:
  b = False
else:
  b = True
print (a, b)

# Instead refactor:
b = not a
print (a, b)
```

Pattern #3 - Collapse nested 'if' statements into one

With many 'if' statements testing several conditions in series or as a complex nested statement, it is often possible to write more readable code by collapsing the logic into a single conditional test.

```
# Not this way:
if a:
    if b:
        print ("a and b")

# Instead refactor:
if a and b:
    print ("a and b")
```

Pattern #4 - Look for and remove redundant statements

Unused and redundant statements clutter code. When maintaining a Python script, having to modify the same piece of code in several places is a route to miss one redundant instance and create an error. Pulling redundant statements outside of 'if' statements and loops is a great way to simplify the code.

```
# Using a redundant state print() statement
if a &gt; b:
  total += a
  print ("total is now", total)
else:
  total += b
  print ("total is now", total)

# Instead refactor:
if a &gt; b:
  total += a
else:
  total += b
print ("total is now", total)
```

Pattern #5 - Use list comprehensions

When dealing with lists, it's quite common to set up a loop to iterate over the list and perform some function on each list element in turn. Quite often, this leads to "boiler plate" code to perform the iteration, keep track of the index in the list, increment it on each iteration and so forth. A more concise, readable and pythonic approach is to use list comprehensions. Quite often this is an easy refactoring approach to apply.

```
# Rather than this loop:
h_letters = []

for letter in 'human':
    h_letters.append(letter)

print(h_letters)

# Refactor like:
h_letters = [letter for letter in 'human']
print( h_letters)

# it's also possible to apply a conditional statement when using list comprehensions
vowels = ['a', 'e', 'i', 'o', 'u']
h_letters = [letter for letter in 'human' if letter in vowels]
print( h_letters)
```

Pattern #6 - Use any() instead of a loop

Rather than iterate through a list, we can use the any() function and a list comprehension to quickly and pythonically determine if any item in a list conforms to a boolean condition.

```
numbers = [-1, -2, -4, 0, 3, -7]
has_positives = False
for n in numbers:
    if n > 0:
        has_positives = True
        break

# Instead refactor
has_positives = any(n > 0 for n in numbers)
```

Pattern #7 - Replace a manual loop counter with enumerate()

Within a loop, it's common to want to know which is the current iteration - this can be done with a list counter. A better alternative is to let Python keep track of the iteration... it has to anyway! ;-)

```
# Manually counting...
i = 0
for player in players:
    print(i, player)
    i += 1

# Instead refactor:
for i, player in enumerate(players):
    print(i, player)
```

Pattern #8 - Move widely duplicated code to functions

If you find yourself writing the same code two or more times, refactor that code to a function. It's quite a common **#fail** to make errors when working on duplicates of fragments of code by correcting some but not all of the duplicate instances.

Simply relocating duplicated code to a function reduces maintenance and improves readability with concise code.

```
# Not this
if any(gst_included(sale) for sale in quarterly_sales):
  base_value = cost_price + margin
  gst_value = base_value * 1.1
  all_gst_sales.append (base_value * 1.1)
  print (base_value * 1.1)
  # etc...

# Instead refactor the GST calculation to a function
def gst_included_price (sale, gst_applied):
  return sale * gst_applied

if any(gst_included(sale) for sale in quarterly_sales):
  all_gst_sales.append(gst_included_price(cost_price + margin, GST_APPLIED))
```

Pattern #9 - Use default values on functions where appropriate

In Python, you can define a function that takes a variable number of arguments by providing default values that are passed as arguments by default. These provide "expected behaviour" where the programmer can handle most use cases but include the functionality to handle other, non default, arguments.

```
# Not this:
def maybe_repeat (thing, number_of_times):
  # do some stuff in a loop
  pass

maybe_repeat (my_thing, 1)

# Instead refactor:
def maybe_repeat (thing, number_of_times = 1):
  # do some stuff in a loop just once - or more if required
  pass

maybe_repeat (my_thing)
```

Pattern #10 - Reduce logical expressions to their simplest form

Python provides the 'if' statement for conditional execution. Using 'if', 'elif' and 'else' we can craft expressions of arbitrary complexity to determine whether code is executed. In maintaining Python code, reading and understanding complex conditional statements is fraught with danger. Remembering The Zen of Python ideas that "simple is better than complex" and "complex is better than complicated", we can refactor 'if' statements to make their meaning simple, readable and obvious.

One common "anti-pattern" is to write an 'if' statement that, when True, will do nothing. We can refactor this code to adjust the boolean logic (without affecting its function) and so simplify our code. For example:

```
# Not this:
if (a < b):
    # pass is a null operation - when it is executed, nothing happens.
    pass
else:
    print('a is greater than or equal to b')

# Instead refactor:
if (a >= b):
    print('a is greater than or equal to b')
```

Resources

You can download the example code above as a Jupyter notebook here: [module-c-week-3-refactoring-patterns.ipynb. (https://myuni.adelaide.edu.au/courses/86136/files/12213403?wrap=1)](https://myuni.adelaide.edu.au/courses/86136/files/12213403?wrap=1)

B2 Topic 6: Test Driven Development (TDD)

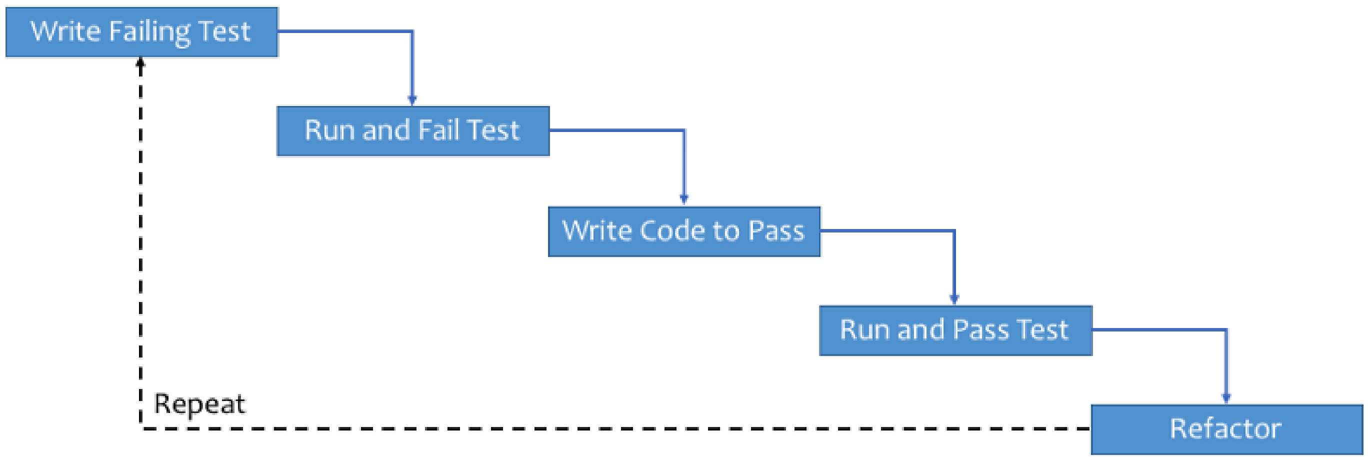
The Nature of Software Testing

When developing software to meet a specified requirement, we often commence by writing code and, to prove that we have met the requirements, we write unit tests and black box test that exercise the code with different input parameters while checking that the output matches the specification. We have discussed the fundamentals of testing here: **B1 | Topic 3: Testing Basics (https://myuni.adelaide.edu.au/courses/86136/pages/b1-%7C-topic-3-testing-basics)** .

Test software often becomes a valuable part of any development and, when changes are made - for example during refactoring or enhancement - we can use the suite of test software to conduct regression testing.

An Alternate Approach

Test-driven development is an alternate development philosophy, expounded this century, that transforms our software requirements directly into coded tests before the specified features are themselves written.



TDD has a workflow that commences by writing a test that fails: a “failing test”. The code is executed and we then write *just enough* code until that test case passes. We consider how we might refactor the newly written and extant code to incrementally improve the quality of our codebase and then iterate by writing our next test.

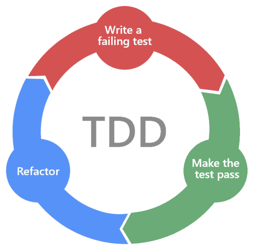
Advantages of a Test Driven Approach

The advantages of using a test-driven approach are several:

- The written code is the minimum viable implementation to meet the requirements.
- A suite of tests is built that facilitates automated testing - many organisations run such tests nightly, continuously, when merging the work of different developers and/or before releasing software.
- Test cases always exist for regression testing when changing the codebase to add new features.

- In refactoring code, we are guaranteed that changing the form of the code does not alter its function.

Red-Green-Refactor and Test Harnesses



Test Driven Development is an iterative process that is often automated with a supporting test harness or test framework. Building a suite of tests with known inputs and expected outputs is fundamental to TDD and the heart of the the "*Red-Green-Refactor*" methodology so called for its three phases:

1. write a *failing test* for a requirement,
2. implement just enough to pass the test for that requirement, and
3. refactor code if required to improve the quality of the implementation (and retest!).

A fully featured test harness focused on *unit testing* of small pieces of code (units) isolates the software under test to a limited scope by ***stubbing*** (https://en.wikipedia.org/wiki/Method_stub) , ***mocking*** (https://en.wikipedia.org/wiki/Mock_object) or otherwise faking dependencies with test software such that errors are localised to the software under test. Larger scale testing, called integration testing, then builds on unit testing to chain together software elements that have passed unit testing to exercise and observe the behaviour of multiple software elements.

Building a Test Harness

We can illustrate the Red-Green-Refactor approach by building our own test harness for TDD:

```
# Define the interface for the function under test
def check_threshold (number, minimum_threshold):
    """ Requirements:
    - 1. accepts two numbers and returns True when number >= minimum_threshold
    - 2. otherwise returns False
    """
    pass

# Write tests for the function under test - we know these will fail hence "failing tests"
def test_case_1 (function_under_test):

    print (f"Testing function: {function_under_test.__name__}, Requirement 1")

    # Test requirement 1
    try:
        assert function_under_test(8, 4) == True
        print ("Requirement 1 passed\n")
    except:
        print ("Requirement 1 failed\n")
```

```
def test_case_2 (function_under_test):

    print (f"Testing function: {function_under_test.__name__}, Requirement 2")

    # Test requirement 2
    try:
        assert function_under_test(4, 8) == False
        print ("Requirement 2 passed\n")
    except:
        print ("Requirement 2 failed\n")

# Run failing tests
test_case_1 (check_threshold)
test_case_2 (check_threshold)

# Implement just enough of the requirements to pass the test but not more
def check_threshold (number, minimum_threshold):
    """ Requirements:
    - accepts two numbers and returns True when number >= minimum_threshold
    - otherwise returns False
    """
    if number >= minimum_threshold:
        return True
    else:
        return False

# Rerun failing tests to confirm our implemented requirement has passed.
test_case_1 (check_threshold)
test_case_2 (check_threshold)

# Consider refactoring (and retesting) the implemented requirements if required.
# One cycle of red-green-refactor has been completed.
```

```
Testing function: check_threshold, Requirement 1
Requirement 1 failed

Testing function: check_threshold, Requirement 2
Requirement 2 failed

Testing function: check_threshold, Requirement 1
Requirement 1 passed

Testing function: check_threshold, Requirement 2
Requirement 2 passed
```

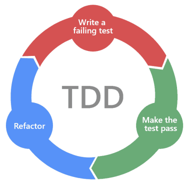
Working through the example, we observe the following approach:

- write a function definition for the code to be tested,
- write failing tests for the function using the `assert` statement to test for expected behaviour,
- prove the test fail,
- implement just enough of the requirements to pass the tests,
- re-test, and
- consider refactoring the code - sometimes iterative development leads to simple and expedient choices that must be more elegantly refactored to maintain code quality.

Functions as Parameters

The calls to the test cases (`test_case_1` and `test_case_2`) pass the function under test as an argument. Passing a function as an argument is a sophisticated, pythonic feature of Python programming and a common element of many programming and scripting languages. You

B2 Topic 6: A Worked Example of TDD



We'll now examine the workflow of a test driven approach starting with our requirements for the software we are to create.

Functional Requirements:

We wish to implement a Python function with two functional requirements:

- “the function shall accept a list of integers and return their sum”, and
- "the function shall throw an exception if the argument is not a list or is a list of zero length".

In specifying software requirements, the word "shall" indicates a function that **must** be performed whilst "should" is used, as the name suggests, as a non-mandatory desire for the software to function as described.

Unit Test Development:

We first design, write and execute a test case that:

- creates a list of test data with a known sum,
- invokes the function that will calculate the sum, and
- tests using assert() that the returned sum matches the pre-calculated sum.

Info: Python assert statement

As we have seen here: **B1 | Topic 3: Testing Basics (https://myuni.adelaide.edu.au/courses/86136/pages/b1-%7C-topic-3-testing-basics)** , “assert” is a built-in statement used to test a condition and halts the program throwing an AssertionError if the condition is not met.

For the software under test (in this case, the function sum_array()), we use a "stub" - a function that advertises the interface (name, argument list etc) of the software under test but does nothing.

In running this test against a yet to be implemented function, we know that the unit test will fail. We have created our first "failing test case".

Writing the unit test as a function itself allows us to build up our own library of tests. Typically, test software and the software under test would live in separate files.

Our example unit test might look like this:

In [1]:
```
import random

def sum_array (list):
    pass

def test_list_sum ():

    test_data = []
    expected_result = 0

    for n in range(0,12):
        num = random.randint(1,10)
        expected_result += num
        test_data.append (num)

    actual_result = sum_array(test_data)

    assert actual_result == expected_result, "test_list_sum() failed"

test_list_sum()
```

Out[1]:
```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/var/folders/jl/gkj9001n6453qdzx_kbdzk_80000gn/T/ipykernel_76352/1664273468.py in
     22     assert actual_result == expected_result, "test_list_sum failed"
     23
---> 24 test_list_sum()

/var/folders/jl/gkj9001n6453qdzx_kbdzk_80000gn/T/ipykernel_76352/1664273468.py in test_lis
t_sum()
     20     print (f"actual result: {actual_result}")
     21
---> 22     assert actual_result == expected_result, "test_list_sum failed"
     23
     24 test_list_sum()

AssertionError: test_list_sum failed
```

We are now free to implement the first (and only the first) requirement by completing the stub for sum_array() - certain that, when it passes the test we have previously written, the requirement has been satisfied. In this case, our job is easy and we can add some diagnostic output as well:

In [2]:
```
import random

def sum_array (list):
    return sum(list)

def test_list_sum ():

    test_data = []
    expected_result = 0

    for n in range(0,12):
        num = random.randint(1,10)
        expected_result += num
        test_data.append (num)

    actual_result = sum_array(test_data)

    print (f"test data: {test_data}")
    print (f"expected result: {expected_result}")
    print (f"actual result: {actual_result}")

    assert actual_result == expected_result, "test_list_sum() failed"
```

```
test_list_sum()
```

Out[2]:

```
test data: [7, 5, 3, 9, 3, 2, 4, 3, 10, 10, 7, 7]
expected result: 70
actual result: 70
```

The process then iterates as we write a next failing test case for the requirement "the function shall throw an exception if the argument is not a list or is a list of zero length" before writing just enough code for the next test case to pass.

We can now integrate this code into a test harness and practice n iteration of the Red Green Refactor methodology:

```
import random

# definition of the function under test
def sum_array (slist):
    """ returns the sum of elements in the list"""
    pass

# test case
def test_list_sum (function_under_test):

    test_data = []
    expected_result = 0

    for n in range(0,12):
        num = random.randint(1,10)
        expected_result += num
        test_data.append (num)

    try:

        print (f"test data: {test_data}")
        print (f"expected result: {expected_result}")

        actual_result = function_under_test(test_data)

        print (f"actual result: {actual_result}")

        # test actual result against known test data
        assert actual_result == expected_result, "test_list_sum() failed"

        print (f"test of {function_under_test.__name__} PASSED\n")
    except:
        print (f"test of {function_under_test.__name__} FAILED\n")

# failing test - RED
test_list_sum(sum_array)

# implementation
def sum_array (slist):
    """ returns the sum of elements in the list"""
    return sum(slist)

# successful test - GREEN
test_list_sum(sum_array)

# consider refactoring - REFACTOR
```

```
test data: [9, 1, 8, 8, 1, 3, 4, 7, 4, 1, 6, 1]
expected result: 53
actual result: None
test of sum_array FAILED
test data: [9, 5, 7, 2, 2, 7, 7, 1, 7, 4, 2, 5]
expected result: 58
actual result: 58
test of sum_array PASSED
```

Once again, note that we use Python's facility to pass the function under test as an argument to the unit test to allow the same function definition to be reused in the Red phase (failing test) and the Green phase (passing test). We'll use this same approach in this week's TDD practical.

Also note that it would be better practice to control the input data for repeatability rather than generate as we have done here. However, hopefully the illustration of the TDD approach shines through.

Next Steps

At this point, the process of TDD should be sufficiently clear that you can use this as a process in development. We address further examples of TDD in the workshop.

(h (h (h (h (h (h (h (h (h (h

B2 Topic 6: Workshop 1 Activities (Refactoring)

Reflective Learning

Last week, we worked with two key libraries for scientific computing and data science: NumPy and pandas. In the pandas practical, we were introduced to a famous dataset that encompasses details of the passengers on the first and last voyage of the Titanic.

Using pandas, we are able to rapidly draw out meaning from the data. For example, if we examine the survival rates of passengers based on the "class" of their accomodation ("1st class" being the most expensive and luxurious whilst "3rd class" berths were low down in the hull of the ship), we can immediately see the effect of being housed closer to the deck when the ship went down:

```
import pandas as pd

df = pd.read_csv ('titanic.csv')
print(round(df [['Pclass','Survived']].groupby(['Pclass']).mean() * 100, 1))
```

```
        Survived
Pclass
1           63.0
2           47.3
3           24.2
```

In this case we use **pandas.DataFrame.groupby (https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html)** method to split and then gather the data according to the `Pclass` value before calculating the mean. This is a powerful and simple approach to analysing data and shows some of the strengths of pandas in this area.

As an exercise before the Workshop, calculate the survival rate based on *both* sex and passenger class by modifying the code above. The results illustrate the catch cry of those on the Titanic: **"Women and Children First!" (https://en.wikipedia.org/wiki/Women_and_children_first)**.

Practising Refactoring Patterns

In this workshop, we will practise refactoring based on the patterns identified in **C | Week 3 : Refactoring Techniques. (https://myuni.adelaide.edu.au/courses/86136/pages/c-%7C-week-3-refactoring-techniques)**

Recall that we are modifying the form of the program not the function. In order to prove this, first execute the program and record its output. Once you have completed refactoring, rerun the program and prove the function is the same by comparing "before and after" output. Ideally, we could use a test suite to automate and guarantee this process.

Exercise 1 - Try out the "easter egg" that reveals The Zen of Python

Try out the import statement below in a Jupyter notebook or on the Python REPL. The REPL is the Python interpreter accessed from the command line and is great for testing fragments of code. "REPL" is short for Read, Eval, Print and Loop.

Which of the listed aphorisms (short phrases) relates to:

- Pattern 1 - Use readable variable and function names, and
  Pattern 3 - Collapse nested 'if' statements into one

```
import this
```

Exercise 2 - Refactor this program by using patterns #1, #2 and #3

Apply these patterns to the following code:

```
import random

# Given some numbers
num_1 = random.randint(6,10)
num_2 = random.randint(1,5)
num_3 = random.randint(11,20)

print ('1 =>', num_1, ', 2 =>', num_2, ', 3 =>;', num_3)

if num_1 > num_2:
  if num_3 > num_1:
    bat = True
    print("both conditions are ", bat == True)
```

Exercise 3 - Identify and remove redundant statements using pattern #4

Using pattern #4, we can refactor the code to avoid setting and printing the label more than once.

```
# Given some values to computer a discount price when a minimum amount of an item are sold

MINIMUM_AMOUNT_FOR_DISCOUNT = 100
NORMAL_PRICE = 25
DISCOUNT_PRICE = 20
```

```
number_sold = random.randint(50,150)

if number_sold > MINIMUM_AMOUNT_FOR_DISCOUNT:
  total = number_sold * DISCOUNT_PRICE
  label = f'Final Total: {total}'
  print (label)

else:
  total = number_sold * NORMAL_PRICE
  label = f'Final Total: {total}'
  print (label)
```

Exercise 4 - Use patterns #5 and #7 to refactor this loop

Here we can use a list comprehension and the enumerate() function to simplify this program.

```
planets = ["mercury", "venus", "earth", "mars", "jupiter", "saturn", "uranus", "neptune"]
gas_giants = ["neptune", "uranus", "saturn", "jupiter"]

distance_from_sun = 0

for p in planets:

  distance_from_sun += 1

  if p in gas_giants:
    print ("Planet", p, "is a gas giant which is the number", distance_from_sun, "planet from the su
n.")
```

Exercise 5 - Combine the patterns learned to refactor a more complex example

In this example, simplify and refactor the code by creating a function that calculates GST. Since GST in Australia is constant, we can use a default value on the function argument to make the function call easier to read. GST can be defined as a 10% surcharge on prices.

Use patterns #8 and #9 to create a function with a default value for the amount of GST added to a sale in order to remove the duplicated calculation of GST from this example.

The program below iterates over a list of dollar values. For values over $110.00, the program adds a margin of profit ($55) and then calculates GST as an added 10%.

Try to declare and use a CONSTANT for the "magic number" that tells us GST as a fraction of added cost - 10%.

Remember to start refactoring by recording the program output unmodified in order to confirm that refactoring has not altered the function of the program once refactoring is complete.

```
# The dollar value of sales our company has made this quarter
quarterly_sales = [123.00, 120.10, 100.00, 55.90, 563.50]
```

```
# Minimum sale price for GST. Sales above this price add 10% GST
MIN_GST_SALE_IN_DOLLARS = 110.00

# All of the sales above MIN_GST_SALE_IN_DOLLARS to which we will add GST#
all_gst_sales = []

# The margin of profit added to each sale before GST is calculated
MARGIN_IN_DOLLARS = 55.00

def is_gst_added (price):
  return price > MIN_GST_SALE_IN_DOLLARS

for i, sale in enumerate(quarterly_sales):

  if is_gst_added(sale):

    base_value = sale + MARGIN_IN_DOLLARS
    gst_value = (sale + MARGIN_IN_DOLLARS) * 1.1
    all_gst_sales.append (base_value * 1.1)
    print ("Sale with #", i, "with margin added and GST applied: $", base_value * 1.1)

print ("All our sales with margin plus GST:", all_gst_sales)
```

? Q&A Session

If you have any questions on refactoring or other content from this module, please ask for help in this workshop.

(h (h (h (h (h (h (h (h (h (h

B2 Topic 6: Workshop 2 Activities (Test Driven Development)

Reflective Learning

This week we have discussed refactoring Python code. Before this Workshop, take time to apply the refactoring patterns **described here (https://myuni.adelaide.edu.au/courses/86136/pages/c-%7C-week-3-refactoring-techniques)** to these fragments of Python code.

```
# Apply a list comprehension.

numbers = list(range(5))

squares = []
for number in numbers:
    if number%2 == 0:
        squares.append(number ** 2)
print(squares)
```

```
# Iteration with enumerate.

students = ['John', 'Anusha', 'Yue', 'June']

for i in range(len(students)):
    student = students[i]
    print(f"# {i+1}: {student}")
```

```
# Use any() function instead of loop.

things = ['car', 'boat', 'submarine', 'go-cart', 'ute']

found = False
for thing in things:
  if thing == 'ute':
    found = True
    break
found
```

For the Section Test next week, it's important to be able to apply the refactoring patterns presented.

Get ready...

In this workshop, we will practice test driven development by:

- analysing a set of software requirements,
- devising a unit test for each requirement,
- implementing just the requirement under test until the unit test is passed,

- considering if refactoring is required, and
- returning to the beginning of this workflow for the next requirement.

This workshop is designed around revisiting Module C, Week 1 content on Files. In those workshops, we examined reading and writing files. In this case, we will follow a test driven approach to write unit tests for our requirements before writing the code that implements them.

We'll start by downloading **this text file (https://myuni.adelaide.edu.au/courses/86136/files/12213501?wrap=1)** **(https://myuni.adelaide.edu.au/courses/86136/files/12213501/download?download_frd=1)** once more and **this Jupyter Notebook (https://myuni.adelaide.edu.au/courses/86136/files/12213405?wrap=1)** **(https://myuni.adelaide.edu.au/courses/86136/files/12213405/download?download_frd=1)** . Make sure that the text file and the notebook are in the same directory.

Requirements

We will implement unit tests for the following requirements:

1. The program shall open a named file.
2. The program shall be able to open a named file in read-only mode.
3. The program shall be able to open a named file in read-write mode.

Write a Unit Test for Requirement 1

Opening the notebook supplied, we'll now write a unit test for requirement 1.

We first write a stub for a function called `open_file()` (or use the one provided ;-)). Once we have written a unit test that calls this stub function and proved that we can fail the test, we'll then proceed to complete this function.

```
# FUNCTION DEFINITIONS
def open_file (filename, mode='rw'):
    ...
```

We now write our first failing test by calling the stub function. We must design a test to prove that the software functioned as specified. There are several ways to confirm that an open file was returned - perhaps data could be read and printed for example. Whatever you decide, use the statement `assert` to test a condition that would prove proper function.

This part of the process where we consider how we might design tests for requirements is an essential step of TDD and rarely considered in other styles of development until the software is written. At this point, we have a cognitive bias to battle against when considering the ways in which our beautiful "working" software might be broken.

```
def test_open_file ():

    # Your code here...
    test_file = open_file(...)

    # Make sure the file is really open by maybe reading a line from the file
    # although you can be creative here - designing the test is an important
    # element of TDD.
    test_data = test_file.read ...

    # Prove your requirement or throw an AssertionError with a message
    assert what_i_got == what_i_expected, "test_open_file failed"

    # And call this at the conclusion of each test to ensure the file
    # is closed ready for the next test.
    cleanup_after_test(test_file)

    # At the end of your test, it's good practice to return True - but optional.
```

We should now have a failing test that we can execute. We know it won't work until we've implemented the function that opens files!

After each test, we call a cleanup utility function to make sure we have closed open test files. In most unit testing frameworks, a setup and cleanup function would be automatically called before and after each test to reset our environment to a known state. Housekeeping...

Implement Requirement 1

Having written and executed a failing test, we can now write just enough code to fulfil requirement 1.

Write a Unit Test for Requirement 2

And so the process continues. For requirement 2 we need to prove that our open file is read only. We must therefore design a test that proves we cannot modify the open file. Once we have a failing test, we can then extend (or indeed refactor) code to implement this feature. One approach we might take is to assert that `file.readable() and not file.writable()`.

Iterate...

Finally, we wish to address requirement 3: to allow the file to be opened for writing. This presents a more complex challenge for designing a test. To prove we can write to the file, we might cause some modification to a test file before closing the file and then perhaps opening it read-only and proving we have modified the content of the file. But there is another easier method that we used in testing requirement 2.

Consolidation and Review

Use this section of the workshop to discuss the following:

- If there is anything unclear in the module content.
- If there is anything unclear in the previous practice exercises.
- If there is anything unclear in the previous workshop activities.
- If there is anything unclear in the previous practicals.

Module Test

Since this is your second Section Test, you are already aware of the section test guidelines. However, your tutors will go through them again. If you have any questions, please ask your tutors.

Q&A Session

If you have any further questions on the Section Test, TDD or any other content, please clarify them in this workshop.

(h (h (h (h (h (h (h (h (h (h