POWERING POTENTIAL

Data Handling with Python

1 Day Workshop

Yobi Livingstone MSc.

- **Data Scientist 6 years**:
  - Screen2Surgery
    - (Start-up for surgical screening)
  - Corndel (Financial Analyst training)
  - JustIT (NHS Analyst training)

- **Biology teacher 5 years**

- **MSc. Bioinformatics**

(Data Science of Genetics and Proteomics)

- **MA. Bioethics**

(Global Health, Clinical Ethics, Synthetic Biology)

Housekeeping

AM breaks
Lunch
PM breaks

Questions?

INTRODUCTIONS

Introductions

Name

Where do you work?

Knowledge and experience

Your aims for the course?

Course materials

Next up, our first topic…

1. Introduction to Programming for Data Handling

Introduction to programming for data handling

Learning objectives

- Describe the pros and cons of using programming languages to work with data.
- Identify the languages most suitable for data handling.
- Explain the challenges of using programming languages versus data analysis tools.

Expected prior knowledge

- Experience of working with data using data analysis tools such as Excel.

Why programming for data?

Activity: Discussion

- Why do we use programming languages to work with data?
- What benefits do they bring?
- What are the challenges when using them?
- Which programming languages are commonly used for working with data?

Why use programming languages?

- Data exists on machines, and programming is the art of telling machines what to do.

Programming languages let us:

- automate.
- work more flexibly.
- solve specific problems.
- utilise more expansive toolkits.
- integrate into existing applications.

Data

What is data?

Information

- Relevant
- Raw

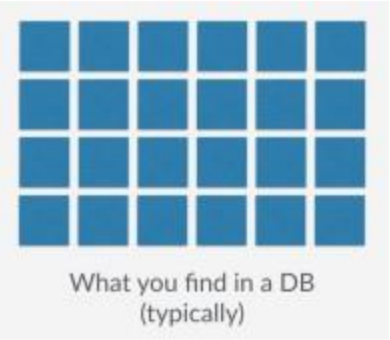
Collected from where?

- Social media bank accounts
- Smart devices
- GPS
- Everywhere...

What are the data types (forms)?

Structured data

- Tabular
- SQL Databases



Unstructured data

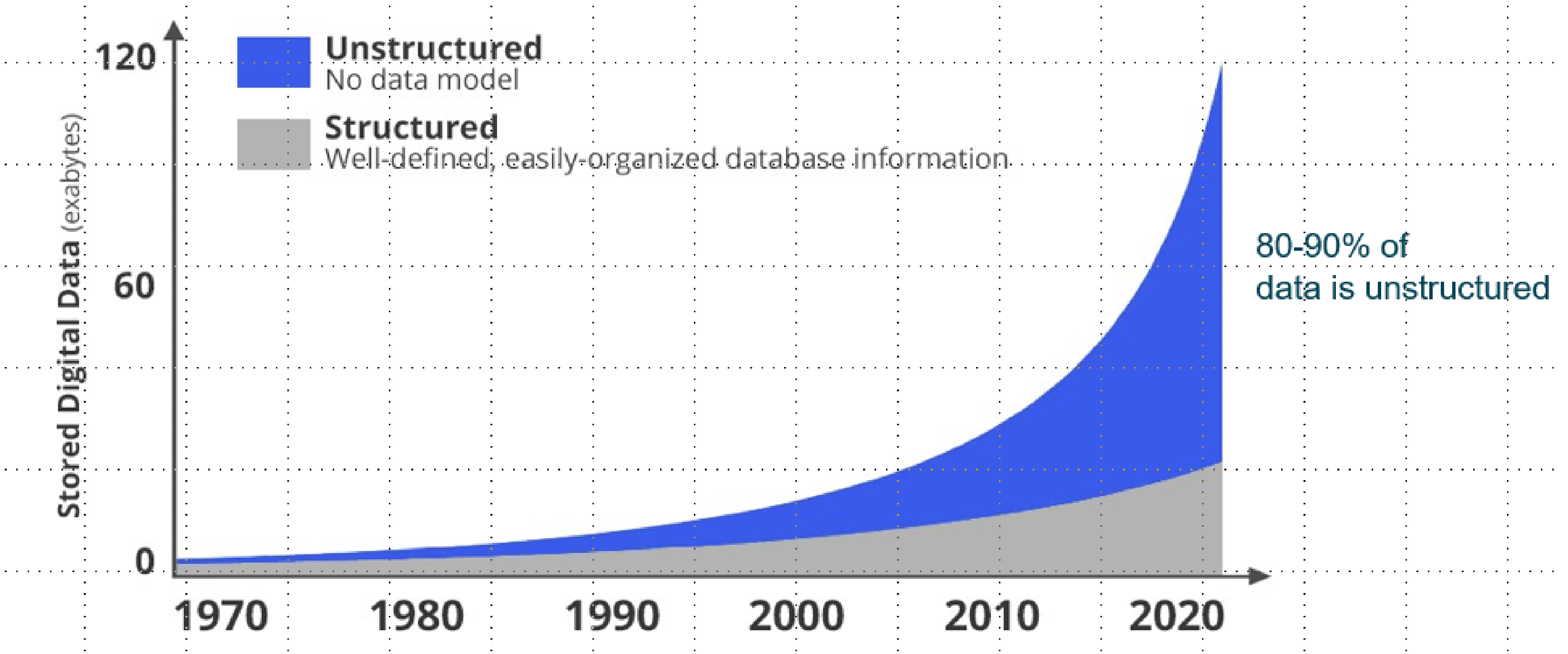
- Photos
- Sounds
- Videos

Why use programming languages with data?

Volume of unstructured data is growing & that growth is accelerating

Structured: Tables, databases

Unstructured: Images, audio, video, Social media posts, reviews, emails



Which tool for data?

Commonly used data tools

Programming languages

R, python, Scala

Databases

MySQL

Command line tools

git

Spreadsheets

Excel

Business intelligence tools

tableau, Power BI

Commonly used programming languages

- Python
  - …and the NumPy ecosystem
- R
- Scala
- Julia
- VBA
- DAX
- Go

- Almost always high-level

Programming language pros and cons

Pros:

- More flexible than analysis tools.
- Can run more efficiently.
- Easier to integrate into software applications.

Cons:

- Require a more technical skillset.
- Need appropriate environment and permissions.
- Can take longer than using a data analysis tool.

Python for Data

Python libraries

NumPy

- Fast numerical arrays.
- Optimised fortran and C extensions.

Pandas

- numpy wrapper.
- Provides ‘data frames’.
- Tabular model over numpy arrays.

matplotlib

- Visualisation and plotting.

seaborn

- Convenience matplotlib wrapper.

Python libraries

Bokeh

- Alternative graphing library (for the web).
- Especially useful for geoplots and other complex plots.

SciKit Learn

- Comprehensive machine learning library.
- Provides good-enough implementations of most key algorithms.

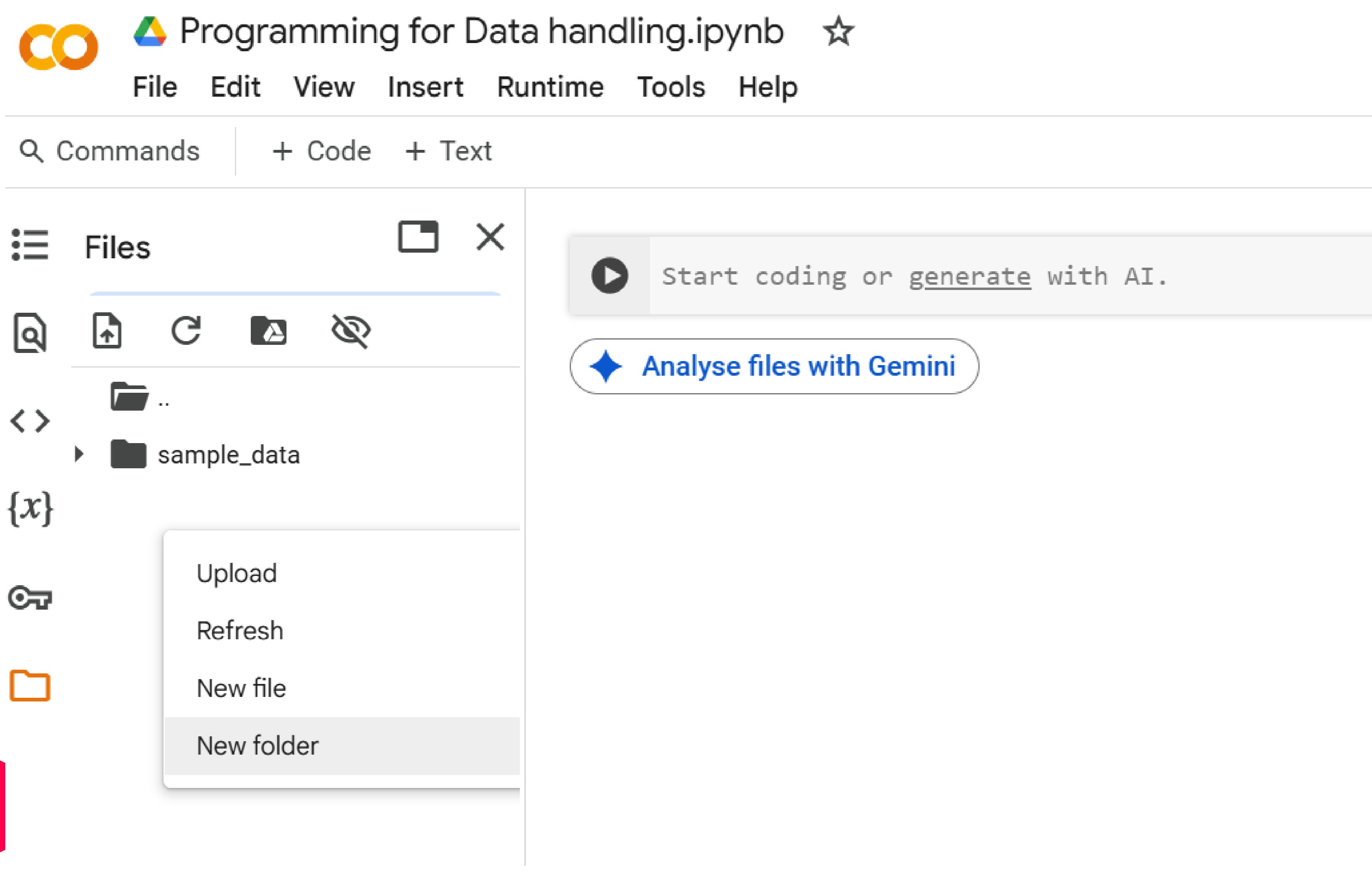
Tensorflow

- Fast (concurrent, distributed, gpu) numerical computing library.
- Describes computations as optimisable graphs.

Keras

- Tensorflow (et al.) wrapper providing neural network abstractions.

Create a new Google Colab notebook - 1) create folder “data” 2) upload contents from local “data” folder

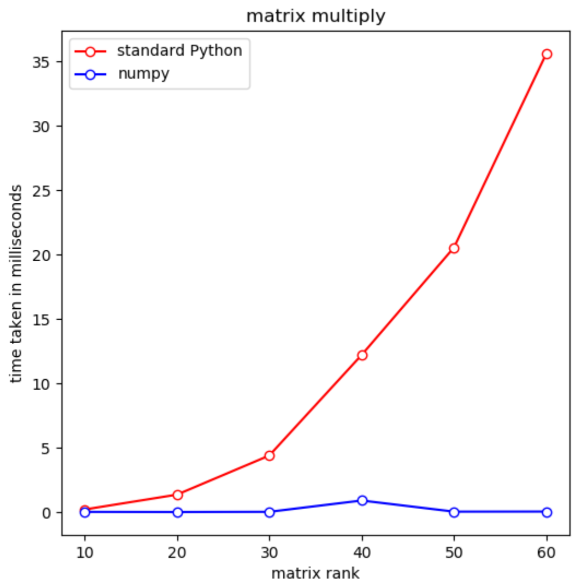


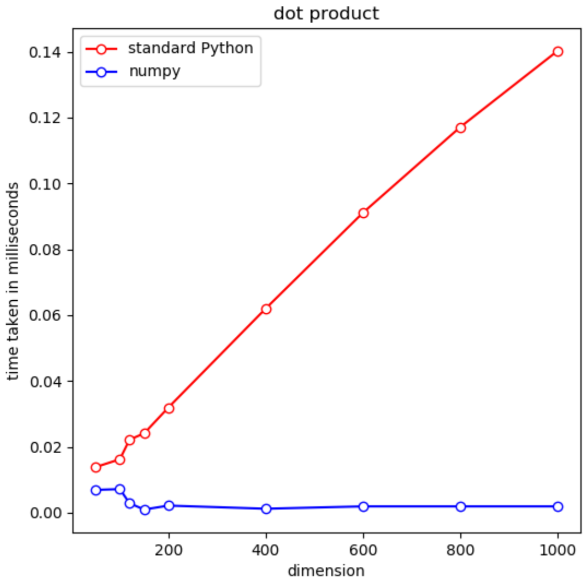
Why NumPy?

PYTHON IS SLOW

Python collections are not designed for computational efficiency.

C and FORTRAN arrays are much more efficient computationally for large datasets.





WHAT IS NUMPY?

NumPy was introduced in 2006 to address the inefficiencies of Python in dealing with large amounts of data.

- Written in C and FORTRAN.
- Internal data structure uses C arrays.
- Python API for seamless integration with Python.
- **Provides its own array types (ND-arrays).**
- **Arrays retain most Python collection behaviours, so that it looks and feels ‘native’ to Python language.**
- Incorporates fast maths libraries, such as OpenBLAS (default, open source), for efficient linear algebraic operations (dot products, matrix multiply, etc.).

Note: To use NumPy it is necessary to import it.

```
import numpy as np
```

NDArrays

ND-ARRAYS

```
payments = np.array([6.99, 12.40, 75.00, 1.55])
```

ND-arrays stands for **N-dimensional arrays.**

- The basic data type in NumPy, intended to replace Python's list.
- Can be created from Python's list using **numpy.array().**
- nd-arrays are **mutable.**
- **numpy.arange()** produces a sequence of numbers contained in an array.

BROADCASTING OPERATIONS

```
payments = np.array([6.99, 12.40, 75.00, 1.55])
transaction_fee = 1.00

payments - transaction_fee
```

```
array([ 5.99, 11.4 , 74.   ,  0.55])
```

```
payments = np.array([6.99, 12.40, 75.00, 1.55])
vat = 1.20

payments * vat
```

```
array([ 8.388, 14.88 , 90.   ,  1.86 ])
```

BROADCASTING OPERATIONS

Elementwise operators

~	NOT
&	AND
|	OR
^	XOR

```
payments = np.array([6.99, 12.40, 75.00, 1.55])

payments > 5.00
```

```
array([ True,  True,  True, False])
```

```
payments = np.array([6.99, 12.40, 75.00, 1.55])

(payments > 5.00) & (payments < 50.00)
```

```
array([ True,  True, False, False])
```

SLICING AND DICING

Standard Python list slices:

- array[i] obtains the i-th element.
- array[n:m] obtains the elements array[n], array[n+1], ..., array[m-1] in a new array.
- array[I,j] obtains the element on row I and column j of a 2-dimensional array.

New to ND-arrays:

Cherry-picking

- array[[2, 4, 5, 1]] obtains the elements array[2], array[4], array[5], array[1] in a new array.
- Cherry picking list can be any Python iterator with integer elements.

Filtering

- array[[True, True, False, ... False, True]] obtains the elements from positions marked as True in a new array and omits those marked by False.
- Filter list can be any Python iterator with Boolean elements, and its length must be the same as the array.
- The filter list is usually computed rather than written by hand

SLICING AND DICING

```python
payments = np.array([6.99, 12.40, 75.00, 1.55])

payments[0:2]
```

```
array([ 6.99, 12.4 ])
```

```python
payments[2:]
```

```
array([75.  ,  1.55])
```

```python
payments[[0, 2]]
```

```
array([ 6.99, 75.  ])
```

ND-ARRAYS

Creating a 1-dimensional array:

```
import numpy as np

arr = np.array([1, 3, 5, 7, 9])

print(arr)
```

```
[1 3 5 7 9]
```

Creating a 2-dimensional array:

```
arr2 = np.array([[1, 3, 5, 7], [2, 4, 6, 8]])

print(arr2)
```

```
[[1 3 5 7]
 [2 4 6 8]]
```

The ndim attribute returns the number of dimensions of the array:

```
arr2.ndim
```

```
2
```

ND-ARRAY SHAPE AND SIZES

The built-in function len() does not work with nd-arrays.

- To find out the size of a nd-array, use **array.size** property.
- To find out the shape (size of each dimension) of an array, use **array.shape** property.
- To change the shape of an array, use **array.reshape().**

Note: It is the programmer's responsibility to make sure the new shape is compatible with the total number of elements.

ND-ARRAY SHAPE AND SIZES

The shape attribute returns a tuple with the number of elements in each dimension.

```
arr2 = np.array([[1, 3, 5, 7], [2, 4, 6, 8]])

print(arr2.shape)
```

```
(2, 4)
```

- 2 rows, 4 columns

Reshaping an array means changing the number of dimensions or changing the number of elements in each dimension. This is done using reshape().

```
arr = np.array([[1, 3, 5, 7, 9, 11], [2, 4, 6, 8, 10, 12]])

print(arr)
```

```
[[ 1  3  5  7  9 11]
 [ 2  4  6  8 10 12]]
```

```
arr2 = arr.reshape(3,4)
print(arr2)
```

```
[[ 1  3  5  7]
 [ 9 11  2  4]
 [ 6  8 10 12]]
```

ND-ARRAYS

arange() creates an array with evenly spaced values.

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

- **start:** The first value in the array.
- **stop:** The number that defines the end of the array. It is not included in the array.
- **step:** The spacing (difference) between each two consecutive values in the array. The default step is 1. Step cannot be zero.
- **dtype:** The type of the elements of the output array. Defaults to None. If dtype is omitted, arange() will try to deduce the type of the array elements from the types of start, stop, and step.

```
MyArray = np.arange(start=1, stop=10, step=2)
print(MyArray)
```

```
[1 3 5 7 9]
```

```
MyArray = np.arange(start=1, stop=10, step=3)
print(MyArray)
```

```
[1 4 7]
```

DTYPE

All arrays can only contain elements of the same data type.

- This is valid for ND-arrays too.

The type of the element in an array is recorded as a dtype object:

- Standard Python data types can be used as dtypes: e.g., int, float, str.
- dtype of an array can be obtained using array.dtype property.
- We can perform type conversion using array.astype(new_type).
- The new_type must be compatible with the original type of the elements.
- If in doubt, NumPy automatically converts an array to an array of strings.

Mathematical & statistical methods

LOGICAL OPERATORS AND FUNCTIONS

Functions acting on entire array

- numpy.all()
- numpy.any()

```
payments = np.array([6.99, 12.40, 75.00, 1.55])
```

```
np.all(payments > 1)
```

```
True
```

```
np.any(payments < 2)
```

```
True
```

DESCRIPTIVE (SUMMARY) STATISTICS

NumPy comes with a full set of statistical functions:

```
payments = np.array([6.99, 12.40, 75.00, 1.55])
```

numpy.sum()
numpy.min()
numpy.max()

```
payments.sum()
```

```
95.94
```

```
payments.min()
```

```
1.55
```

```
payments.max()
```

```
75.0
```

DESCRIPTIVE (SUMMARY) STATISTICS

NumPy comes with a full set of statistical functions:

numpy.mean()

numpy.median()

numpy.var()

numpy.std()

numpy.corrcoef()

```
payments.mean()
```

23.985

```
payments.var()
```

882.225425

```
payments.std()
```

29.70227979465549 2

Ufuncs

UNIVERSAL FUNCTIONS

A universal function, or ufunc, is a function that performs element-wise operations on data in ndarrays.

They are fast!

numpy.sqrt()
numpy.square()
numpy.exp()
numpy.log()
numpy.sign()
numpy.isnan()
numpy.sin()
numpy.add()

```
np.sign(payments)
```

```
array([1., 1., 1., 1.])
```

Learning check

Think about your answers to these questions:

- Describe the pros and cons of using programming languages to work with data.
- Identify the languages most suitable for data handling.
- Explain the challenges of using programming languages versus data analysis tools.

How did you get on?

Learning objectives

- Describe the pros and cons of using programming languages to work with data.
- Identify the languages most suitable for data handling.
- Explain the challenges of using programming languages versus data analysis tools.

2. Data Structures, Functions, & Basic Types

Data structures, flow control, functions, & basic types

Learning objectives

- Construct collections to solve data problems.
- Write reusable functions which can be used to alter data and automate repetitive tasks.
- Use Python's built-in open function to create, read, and edit files.

Expected prior knowledge

- Experience of working with data using data analysis tools such as Excel.

Collections

Python 3 types

Sequences

Immutable

Mutable

- Numbers

`3.142, 42, 0x3f, 0o664`

- Bytes

`b'Norwegian Blue', b"Mr. Khan's bike"`

- Strings

`'Norwegian Blue', "Mr. Khan's bike", r'C:\Numbers'`

- Tuples

`(47, 'Spam', 'Major', 683, 'Ovine Aviation')`

- Lists

`['Cheddar', ['Camembert', 'Brie'], 'Stilton']`

- Bytearrays

`bytearray(b'abc')`

- Dictionaries

`{'Sword':'Excalibur', 'Bird':'Unladen Swallow'}`

- Sets

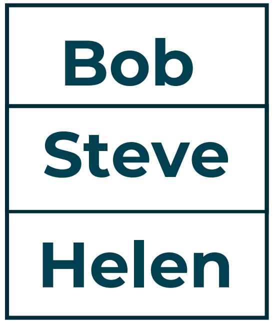
`{'Chapman', 'Cleese', 'Idle', 'Jones', 'Palin'}`

Python lists

Lists store multiple values (elements)

```
numbers = [1,3,5,7]
```


```
names = ['Bob', 'Steve', 'Helen']
```



Lists can store elements of any data type, including other lists.

```
mix = [1,3.14,"fruit",True]
```

Changing the value of elements

```
numbers = [1,3,5,7,5,9,5]

numbers[2] = 999
```

[1, 3, 999, 7, 5, 9, 5]

```
names = ['Bob', 'Steve', 'Helen’]

names[2] = “Chris"
```

['Bob’, ‘Steve’, ‘Chris']

Strings (lists of characters)

Major difference:

- Lists are mutable.
- Strings are immutable.

We can change the value of an element of a list, but not of an element of a string:

```
L = [1, 2, 3, 4, 5]
L[4] = 0
L
```

```
[1, 2, 3, 4, 0]
```

```
S = '12345'
S[4] = 0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-f463423e9604> in <module>
      1 S = '12345'
----> 2 S[4] = 0

TypeError: 'str' object does not support item assignment
```

String methods

Python has a set of built-in methods that can be used on strings.

Note: All string methods returns new values. They do not change the original string.

```
test = 'britain'

test.capitalize()
test
```

```
'britain'
```

```
test1 = test.capitalize()
test1
```

```
'Britain'
```

A list of Python string methods is available here:

https://docs.python.org/3/library/stdtypes.html#string-methods

String methods

Some examples of string methods:

```
testGB = 'great britain'

# Convert the first character to upper case
testGB1 = testGB.capitalize()
testGB1
```

```
'Great britain'
```

```
# Convert the first character of each word to upper case
testGB2 = testGB.title()
testGB2
```

```
'Great Britain'
```

```
# Convert a string into lower case
testGB3 = testGB.lower()
testGB3
```

```
'great britain'
```

String methods - continued

Some examples of string methods:

```
# Convert a string into upper case
testGB4 = testGB.upper()
testGB4
```

'GREAT BRITAIN'

```
# Search the string for a specified value and return the position of where it was found
testGB5 = testGB.find('BRITAIN')
testGB5
```

-1 (Here, means not found)

```
# Python is case sensitive!
testGB5_1 = testGB.find('britain')
testGB5_1
```

String methods - continued

Some examples of string methods:

```
# Return True if all characters in the string are in the alphabet
testGB6 = testGB.isalpha()
testGB6
```

```
False
```

```
# Return True if the string starts with the specified value
testGB6 = testGB.startswith('g')
testGB6
```

```
True
```

The string.Split() method

Used for splitting and extracting elements from a String using a Delimiter:

```
data = 'Bob,Steve,Helen'
names = data.split(',')
print(names)
```

['Bob', 'Steve', 'Helen']

```
data='18/OCT/2020'
parts = data.split('/')
print(parts)
```

['18', 'OCT', '2020']

Operators & type

```
a = 42
b = 9
print(a + b)
print(type(a))

a = 'Hello '
b = 'World!'
print(a + b)
print(type(a))
```

a and **b** refers to integers.

```
51
<class 'int'>
```

a and **b** now refers to strings.

```
Hello World!
<class 'str'>
```

Switching types

Sometimes Python switches automatically.

```
num = 42
pi  = 3.142
num = 42/pi
print(num)
```

`num` gets automatic promotion

`13.367281986`

```
print("Unused port: "  +  count)
TypeError: Can't convert 'int' object to str implicitly
```

```
print("Unused port: " + str(count))
```

Arithmetic operations: The different types of division

```
x1 = 5
x2 = 3

# division
d1 = x1 / x2
# floor (integer) division
d2 = x1 // x2
# modular division (remainder of integer division)
d3 = x1 % x2
```

The results of the three types of divisions are:

d1 = 1.6666666667

d2 = 1

d3 = 2

The floor (integer) division doesn't round, it **truncates** to obtain the integer result.

Functions

User defined functions

The syntax of a Python function is the following:

```
def function_name( parameters ):
    statement1
    statement2
    ...
    ...
    return [expr]
```

- ✓ **def** is a keyword that defines a function.
- ✓ A function may or may not have parameters.
- ✓ A function may or may not return a value.

User defined functions

No parameters, no return value:

```
def hello():
    print("Hello world")
```

Calling the function and output:

```
hello()
```

```
Hello world
```

Function with parameters, no return value:

```
def hello(name):
    print("Hello", name)
```

Calling the function and output:

```
hello("everybody")
```

```
Hello everybody
```

User defined functions

Function with parameters and return value:

```
def rectangle_area(length, width):
    return(length*width)
```

We can save the return value into a variable:

```
area = rectangle_area(5,2)
area
```

```
10
```

... or we can print it:

```
print(rectangle_area(5,2))
```

```
10
```

Create a Function that

Type specific methods

Actions on objects are done by calling *methods*.

- A method is implemented as a *function* - a named code block.

object.*method* ([*arg1*[,*arg2*…]])

- *object* need not be a variable.

Which methods may be used?

- Depends on the Class (type) of the object.
- `dir(`*object*`)` lists the methods available.
- `help(`*object*`)` often gives help text.

Examples:

```
name.upper()            names.pop()
name.isupper()          mydict.keys()
names.count()           myfile.flush()
```

File handling

The open() function

The open() function takes two parameters: file name and mode. Only the file name is mandatory.

There are four different modes for opening a file:

- **"r" - Read - Default value:** Opens a file for reading, error if the file does not exist.
- **"a" – Append:** Opens a file for appending, creates the file if it does not exist.
- **"w" – Write:** Opens a file for writing, creates the file if it does not exist.
- **"x" – Create:** Creates the specified file, returns an error if the file exists.

In addition, you can specify if the file should be handled as binary or text mode.

- **"t" - Text - Default value**: Text mode.
- **"b" – Binary:** Binary mode (e.g., images).

The open() function

The open() function takes two parameters: file name and mode.

The following are equivalent:

f = open('hello.txt')

f = open('hello.txt', 'r')

f = open('hello.txt', ‘rt’)

File input

You can not only read the whole text, but you can also specify what part of it.

 # The first 5 characters of the file

```
f = open('hello.txt', 'r')
print(f.read(5))
```

 # The first line of the file

```
f = open('hello.txt', 'r')
print(f.readline())
```

 # The first 2 lines of the file

```
f = open('hello.txt', 'r')
print(f.readline())
print(f.readline())
```

Working with a file

Let's read file data.txt (supplied):

```
# Locate the file
a = 'data.txt'
# open the file for reading
f = open(a, 'r')
# read the whole content of that file into a single string variable
b = f.read()
# print it
print(b)
```

```
3
5
-2
11
0
7
1
```

Even though it looks like multiple lines, technically variable b will contain this:

'3\n5\n-2\n11\n0\n7\n1'

There are NO new lines (\n) after the last element, i.e., 1 is the last character).

Working with a file

'3\n5\n-2\n11\n0\n7\n1'

```
# split the string variable b into array of strings
c = b.split('\n')
print(c)
```

```
['3', '5', '-2', '11', '0', '7', '1']
```

The code so far can be written in more concise form:

```
file_content = open('data.txt', 'r').read().split('\n')
print(file_content)
```

```
['3', '5', '-2', '11', '0', '7', '1']
```

And then the list with the file content can be further processed as needed.

Note: The list elements are strings, not numbers.

Input from file with a header

To remove a header first line:

1. **# open the file**

   f = open('data2.txt', 'r')

2. **# skip the first line (by reading and discarding)**

   f.readline()

3. **# read the rest**

   file_content = f.read().split('\n')

```
# declare an empty array (output will be accumulated here)
data = []

# iterate over the array
for x in file_content:
    # print(x)
    x = x.strip()
    # x = int(x) # this will fail because of empty lines
    if (x != ''):
        x = int(x)
        data.append(x)
# print the output
print(data)
```

```
[3, 5, -2, 11, 0, 7, 1, 0]
```

File output

To write to a file, open it with one of the following modes:

- **"a" – Append:** Opens a file for appending, creates the file if it does not exist.
- **"w" – Write:** Opens a file for writing, creates the file if it does not exist.

Use the write() function to write output to the file.

Don’t forget to close the file.

```
f = open('output.txt', 'w’)
f.write(‘Hello’)
f.close()
```

Closing a file

It is a very good practice to always close the file when you have finished with it:

f.close()

Note: In some cases, due to buffering, changes made to a file may not show until the file is closed.

Exercise

Go to **Exercise 3: Data structures, flow control, functions & basic types** in your exercise guide.

Learning check

Think about your answers to these questions:

- Which collections can we use to store data in Python? What are their properties?
- How can we control the flow of a Python program?
- What is a function? Do all Python functions return data?
- Which modes can we open a file in? How can we read a file- line by line?

How did you get on?

Learning objectives

- Construct collections to solve data problems.
- Utilise selection and iteration syntax to control the flow of a Python program.
- Write reusable functions which can be used to alter data and automate repetitive tasks.
- Use Python's built-in open function to create, read, and edit files.

4. Introduction to Pandas

Introduction to Pandas

Learning objectives

- Create, manipulate, and alter Series and DataFrames with Pandas.
- Define and change the indices of Series & Dataframes.
- Use Pandas' functions and methods to change column types, compute summary statistics, and aggregate data.
- Read, manipulate, and write data from csv, xlsx, json, and other structured file formats.

Expected prior knowledge

- Experience of working with data using data analysis tools such as Excel.

Why Pandas?

What is Pandas?

- Pandas is Python’s ETL package for structured data.
- Built on top of NumPy, designed to mimic the functionality of R DataFrames.
- Provides a convenient way to handle tabular data.
- Can perform all SQL functionalities, including group-by and join.
- Compatible with many other data science packages, including visualisation packages such as Matplotlib and Seaborn.
- Defines two main data types:
    - **pandas.Series**
    - **pandas.DataFrame**

DataFrames

DataFrame

- A Pandas **DataFrame** represents a table, and it contains:
  - Data in the form of rows and columns.
  - Row IDs (the index array, i.e., primary key).
  - Column names (ID of the columns).

- Equivalent to collection of Series.

- The row indices by default start from 0 and increases by 1 for each subsequent row.

DataFrames are the data structures most suitable for analytics.

- Rows represent observations.
- Columns represent attributes of different data types.

Creating DataFrames

- Creating from Python lists, or NumPy arrays:

```
data = {
    "age": [34, 42, 27],
    "height": [1.78, 1.82, 1.75],
    "weight": [75, 80, 70]
}
df = pd.DataFrame(data)
print(df)
```

```
   age  height  weight
0   34    1.78      75
1   42    1.82      80
2   27    1.75      70
```

- Use a dictionary with column names as keys and a list of the row values .
- Creating from CSV files:

  pandas.read_csv(csv_file_name)

  - The first row is used for column names.

Reading in data

Reading CSV files

- read_csv reads a comma delimited file into a DataFrame.
- Can pass a path or URL to be read from.
- Parameters control how to read.
  - E.g., whether to parse dates or not.

```python
df = pd.read_csv("data/loan_data.csv")

df[:2]
```

	ID	Income	Term	Balance	Debt	Score	Default
0	567	17500.0	Short Term	1460.0	272.0	225.0	False
1	523	18500.0	Long Term	890.0	970.0	187.0	False

Reading Excel files

- read_excel reads an excel file into a DataFrame.
- Pass a path to be read from, as well as the sheet.
- Parameters control how to read.
  - E.g., Whether to parse dates or not.

```
df = pd.read_excel("data/loan_data.xlsx", sheet_name="March")

df[:2]
```

	ID	Income	Term	Balance	Debt	Score	Default
0	567	17500	Short Term	1460	272	225.0	False
1	523	18500	Long Term	890	970	187.0	False

Reading XML & JSON

- read_json reads a JSON file into a DataFrame.
- read_xml reads a XML file into a DataFrame.
- Can pass a path or URL to be read from.
- Parameters control how to read.

```
weather = pd.read_json("data/weather.json", orient="split")
weather
```

	temp	humidity	sun_hrs
2023-07-15	15.68	73.18	6.40
2023-07-16	25.16	83.88	8.06
2023-07-17	13.26	80.05	4.89
2023-07-18	24.63	82.37	9.13
2023-07-19	12.78	83.10	17.10
2023-07-20	23.52	85.35	0.72
2023-07-21	17.80	85.64	5.79
2023-07-22	24.98	76.81	10.95
2023-07-23	23.48	80.86	3.77
2023-07-24	23.30	79.96	14.62

Querying SQL Tables

- read_sql reads the result of a SQL query into a DataFrame.
- Requires appropriate connection to be set up.
  - Including correct credentials.

```
db_conn = sqlite3.connect(r"data/movies_db.sqlite")

movies = pd.read_sql(r"SELECT * FROM movies", db_conn)
movies
```

	id	name	year	rating
0	1	Who's Afraid of Virginia Woolf?	1966	10
1	2	Zardoz	1974	6
2	3	2001: A Space Odyssey	1968	9

Changing types

Changing column types

- Ensuring data is of the correct type is important, both technically and statistically.
- The astype method can be used to do this to Series and DataFrames.

```
df = pd.read_csv("data/loan_data.csv")

df['ID'].head()
```

```
0    567
1    523
2    544
3    370
4    756
Name: ID, dtype: int64
```

```
df['ID'].astype("string").head()
```

```
0    567
1    523
2    544
3    370
4    756
Name: ID, dtype: string
```

Parsing dates & times

- Dates and times are often read in as objects by Pandas.
  - Essentially strings.
- A specific function called to_datetime is used to parse these into datetime objects.
  - Uses strftime.
  - Can be done on file read but it is discouraged.

```
weather['time']
```

```
0    2023-07-15
1    2023-07-16
2    2023-07-17
3    2023-07-18
4    2023-07-19
5    2023-07-20
6    2023-07-21
7    2023-07-22
8    2023-07-23
9    2023-07-24
Name: time, dtype: object
```

```
pd.to_datetime(weather['time'], format="%Y-%m-%d")
```

```
0    2023-07-15
1    2023-07-16
2    2023-07-17
3    2023-07-18
4    2023-07-19
5    2023-07-20
6    2023-07-21
7    2023-07-22
8    2023-07-23
9    2023-07-24
Name: time, dtype: datetime64[ns]
```

Indexing DataFrames

Column retrieval

Getting entire columns:

my_dataframe[column_name]

```
weather['temp']
```

```
0    15.68
1    25.16
2    13.26
3    24.63
4    12.78
5    23.52
6    17.80
7    24.98
8    23.48
9    23.30
Name: temp, dtype: float64
```

```
weather[['temp', 'humidity']]
```

	temp	humidity
0	15.68	73.18
1	25.16	83.88
2	13.26	80.05
3	24.63	82.37
4	12.78	83.10
5	23.52	85.35
6	17.80	85.64
7	24.98	76.81
8	23.48	80.86
9	23.30	79.96

Row retrieval

Getting entire rows:

my_dataframe.loc[row_id]

```
weather.loc[0]
```

```
time        2023-07-15
temp             15.68
humidity         73.18
sun_hrs            6.4
Name: 0, dtype: object
```

Row with index 0

```
weather.loc[[0, 1]]
```

	time	temp	humidity	sun_hrs
0	2023-07-15	15.68	73.18	6.40
1	2023-07-16	25.16	83.88	8.06

Rows with indices 0 and 1

Named row retrieval

Indices can be named:

```
weather.set_index("time", inplace=True)
weather
```

	temp	humidity	sun_hrs
time			
2023-07-15	15.68	73.18	6.40
2023-07-16	25.16	83.88	8.06
2023-07-17	13.26	80.05	4.89

```
weather.loc["2023-07-17"]
```

Row with index "2023-07-17"

```
temp        13.26
humidity    80.05
sun_hrs      4.89
Name: 2023-07-17, dtype: float64
```

```
weather.iloc[2]
```

Row with position 2

```
temp        13.26
humidity    80.05
sun_hrs      4.89
Name: 2023-07-17, dtype: float64
```

Slicing DataFrames

Getting entire columns:

my_dataframe.loc[:, col_name]

my_dataframe.iloc[y:,col_position]

```
weather.loc[:, "temp"]
```

Column “temp”

```
time
2023-07-15    15.68
2023-07-16    25.16
2023-07-17    13.26
```

```
weather.iloc[:, 0]
```

Column with position 0

```
time
2023-07-15    15.68
2023-07-16    25.16
2023-07-17    13.26
```

Slicing DataFrames

Getting individual elements from row and column IDs:

my_dataframe.loc[row_id, col_name]

my_dataframe.iloc[i, j]

```
weather.loc["2023-07-15", "humidity"]
```

Row index "2023-07-15" Column "humidity"

```
73.18
```

```
weather.iloc[0, 1]
```

Row 0 Column 1

```
73.18
```

Slicing summary

my_dataframe.loc[[id1, id2, id3], :]

returns rows id1, id2 and id3, all columns

my_dataframe.loc[:, [col1, col2, col3]]

returns columns col1, col2 and col3, all rows

my_dataframe.loc[[id1, id2, id3], [col1, col2, col3]] returns 3 by 3 table of rows id1, id2 and id3, columns col1, col2, and col3

Querying DataFrames

Broadcasting operations

- Like NumPy, Pandas broadcasts operations.
- I.e., we can perform calculations with columns like we do with single values.

```
df['Income'].head()
```

```
0    17500
1    18500
2    20700
3    21600
4    24300
Name: Income, dtype: int64
```

```
(df['Income'] / 12).head()
```

```
0    1458.333333
1    1541.666667
2    1725.000000
3    1800.000000
4    2025.000000
Name: Income, dtype: float64
```

Boolean operators

Symbolic Boolean operators can be used to combine conditions.

```
(df["Income"] > 20000) & (df["Debt"] == 0)
```

```
0      False
1      False
2      False
3       True
4       True
       ...
851    False
852     True
853    False
854    False
855    False
Length: 856, dtype: bool
```

Filtering

- DataFrames can be filtered row-wise using a sequence of Trues & Falses.
- These can be generated by queries.

```
df[(df["Income"] > 20000) & (df["Debt"] == 0)]
```

	ID	Income	Term	Balance	Debt	Score	Default
3	370	21600	Short Term	920	0	NaN	False
4	756	24300	Short Term	1260	0	495.0	False
6	373	20400	Short Term	1200	0	556.0	False
7	818	24600	Short Term	1470	0	301.0	False
9	621	25400	Short Term	1130	0	729.0	True
...	...	...	...	...	...	...	...
847	96	26300	Long Term	1760	0	489.0	False
848	762	29200	Long Term	1500	0	755.0	False
849	516	36200	Short Term	1510	0	812.0	False
850	627	27000	Short Term	1510	0	436.0	False
852	932	42500	Long Term	1550	0	779.0	False

299 rows × 7 columns

Aggregation

GROUP BY

Group table rows into sub-groups according to a specified criteria.

DataFrame

Index	Name	Gender	Age
0	Alice	Female	23
1	Bob	Male	26
2	Charlie	Male	25
3	Dave	Male	24

my_dataframe

Series

Index	Group
0	A
1	B
2	A
3	B

criteria

Group	Name	Gender	Age
A	Alice	Female	23
	Charlie	Male	25

Group	Name	Gender	Age
	Bob	Male	26
B	Dave	Male	24

my_dataframe.groupby(criteria)

GROUP BY

GROUP BY and:

- Counting the number of rows in each group:

my_dataframe.groupby(criteria).size()

- Sum of every numerical column in each group:

my_dataframe.groupby(criteria).sum()

- Mean of every numerical column in each group:

my_dataframe.groupby(criteria).mean()

```
df[["Term", "Balance"]].groupby("Term").sum()
```

	Balance
Term	
Long Term	362870
Short Term	676600

Transform

- Transform is uses to calculate quantities over a group but return as many rows as input.
- Can be used to add, e.g., a grouped average column.

```
df['MeanTermDebt'] = df.groupby("Term")['Debt'].transform(np.mean)
df.head()
```

	ID	Income	Term	Balance	Debt	Score	Default	MeanTermDebt
0	567	17500	Short Term	1460	272	225.0	False	610.232877
1	523	18500	Long Term	890	970	187.0	False	715.823529
2	544	20700	Short Term	880	884	85.0	False	610.232877
3	370	21600	Short Term	920	0	NaN	False	610.232877
4	756	24300	Short Term	1260	0	495.0	False	610.232877

Exercise

Go to **Introduction to Pandas** in your exercise guide.

HOW DID YOU GET ON?

Learning objectives

- Create, manipulate, and alter Series and DataFrames with Pandas.
- Define and change the indices of Series and Dataframes.
- Use Pandas' functions and methods to change column types, compute summary statistics and aggregate data.
- Read, manipulate, and write data from csv, xlsx, JSON, and other structured file formats.

05. Data cleaning with Pandas

Learning objectives

- Identify missing data and apply techniques to deal with it.
- Deduplicate, transform, and replace values.
- Use DataFrame string methods to manipulate text data.
- Write regular expressions which munge text data.

Expected prior knowledge

- Experience of working with data using data analysis tools such as Excel.

Data quality dimensions - ISO9001

Accuracy: Truthful representation

Completeness: All data %

Uniqueness: Avoid duplication

Timeliness: Is current and relevant

Validity: Correct format and structure

Consistency: Data is same across system

Missing values - Completeness

Missing Values

What is the problem with missing data?

How do we deal with missing it?

There are three main options:

1) Removal.
2) Imputation – requires skill.
3) Leave as is; some models can deal with missing values.

Representing missing values

- Pandas represents all missing values as NaN.
  - None will be converted to NAN.

```
df = pd.DataFrame({
    'participant': [1,2,3,4],
    'age': [50, None, 30, np.NaN],
    'satisfaction': [None, 8, 9, None]
})
```

```
df
```

	participant	age	satisfaction
0	1	50.0	NaN
1	2	NaN	8.0
2	3	30.0	9.0
3	4	NaN	NaN

Finding missing values

```
df.isna().any()
```

```
participant      False
age               True
satisfaction      True
dtype: bool
```

```
df.isna().sum()
```

```
participant     0
age             2
satisfaction    2
dtype: int64
```

```
df[df.isna().any(axis=1)]
```

	participant	age	satisfaction
0	1	50.0	NaN
1	2	NaN	8.0
3	4	NaN	NaN

```
df[~df.isna().any(axis=1)]
```

	participant	age	satisfaction
2	3	30.0	9.0

Deleting missing values

```
df.dropna()
```

	participant	age	satisfaction
2	3	30.0	9.0

```
df.dropna(thresh=2)
```

	participant	age	satisfaction
0	1	50.0	NaN
1	2	NaN	8.0
2	3	30.0	9.0

```
df.dropna(subset=['age', 'satisfaction'], how='all')
```

	participant	age	satisfaction
0	1	50.0	NaN
1	2	NaN	8.0
2	3	30.0	9.0

Filling in missing values

- If filling in values, it is common to use an average.
- Use fillna to specify a value (depending on the column) to replace each NaN with.

```
df.fillna({'age'          : df['age'].mean(),
           'satisfaction': df['satisfaction'].mode()[0]
          })
```

	participant	age	satisfaction
0	1	50.0	8.0
1	2	40.0	8.0
2	3	30.0	9.0
3	4	40.0	8.0

Deduplication - Uniqueness

Duplicates

```python
loans_dup = pd.read_csv("data/loan_data2.csv")

loans_dup
```

	ID	Income	Term	Balance	Debt	Score	Default
0	567	17500	Short Term	1460	272	225.0	False
1	523	18500	Long Term	890	970	187.0	False
2	544	20700	Short Term	880	884	85.0	False
3	370	21600	Short Term	920	0	NaN	False
4	756	24300	Short Term	1260	0	495.0	False
...	...	...	...	...	...	...	...

```python
loans_dup.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 956 entries, 0 to 955
Data columns (total 7 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   ID       956 non-null    int64  
 1   Income   956 non-null    int64  
 2   Term     956 non-null    object 
 3   Balance  956 non-null    int64  
 4   Debt     956 non-null    int64  
 5   Score    936 non-null    float64
 6   Default  956 non-null    bool   
dtypes: bool(1), float64(1), int64(4), object(1)
memory usage: 45.9+ KB
```

```python
loans_dup.duplicated()
```

```
0      False
1      False
2      False
3      False
4      False
       ...  
951     True
952     True
953     True
954     True
955     True
Length: 956, dtype: bool
```

```python
loans_dup.duplicated().sum()
```

```
100
```

Identifying and removing duplicates

```
loans_dup[loans_dup.duplicated()]
```

	ID	Income	Term	Balance	Debt	Score	Default
856	526	66200	Long Term	1700	0	1000.0	False
857	773	63700	Short Term	1630	1912	1000.0	False
858	317	64000	Short Term	2420	0	1000.0	False
859	439	61700	Long Term	1380	0	629.0	False
860	383	56300	Long Term	2020	2542	957.0	False
...	...	...	...	...	...	...	...

```
loans_dup.drop_duplicates()
```

	ID	Income	Term	Balance	Debt	Score	Default
0	567	17500	Short Term	1460	272	225.0	False
1	523	18500	Long Term	890	970	187.0	False
2	544	20700	Short Term	880	884	85.0	False
3	370	21600	Short Term	920	0	NaN	False
4	756	24300	Short Term	1260	0	495.0	False
...	...	...	...	...	...	...	...

Data transformation – Validity

Dropping values

```
df.drop("ID", axis=1)
```

	Income	Term	Balance	Debt	Score	Default
0	17500	Short Term	1460	272	225.0	False
1	18500	Long Term	890	970	187.0	False
2	20700	Short Term	880	884	85.0	False
3	21600	Short Term	920	0	NaN	False
4	24300	Short Term	1260	0	495.0	False
...	...	...	...	...	...	...

Column "ID"

```
df.drop(0, axis=0)
```

Row 0

	ID	Income	Term	Balance	Debt	Score	Default
1	523	18500	Long Term	890	970	187.0	False
2	544	20700	Short Term	880	884	85.0	False
3	370	21600	Short Term	920	0	NaN	False
4	756	24300	Short Term	1260	0	495.0	False
5	929	22900	Long Term	1540	1229	383.0	False
...	...	...	...	...	...	...	...

Data transformation

We often want to change how we represent data. This could involve:

- replacing some values with others.
- binning continuous variables.
- deriving new columns.
- applying functions.

```
df = pd.read_csv("data/loan_data.csv")
df
```

	ID	Income	Term	Balance	Debt	Score	Default
0	567	17500	Short Term	1460	272	225.0	False
1	523	18500	Long Term	890	970	187.0	False
2	544	20700	Short Term	880	884	85.0	False
3	370	21600	Short Term	920	0	NaN	False

Replacing values

```python
df.replace(to_replace={
    'Long Term': 1,
    'Short Term': 0
}).head()
```

	ID	Income	Term	Balance	Debt	Score	Default
0	567	17500	0	1460	272	225.0	False
1	523	18500	1	890	970	187.0	False
2	544	20700	0	880	884	85.0	False
3	370	21600	0	920	0	NaN	False
4	756	24300	0	1260	0	495.0	False

```python
df.replace(to_replace={
    r'Long': "12 Month",
    r'Short': "6 Month"
}, regex=True).head()
```

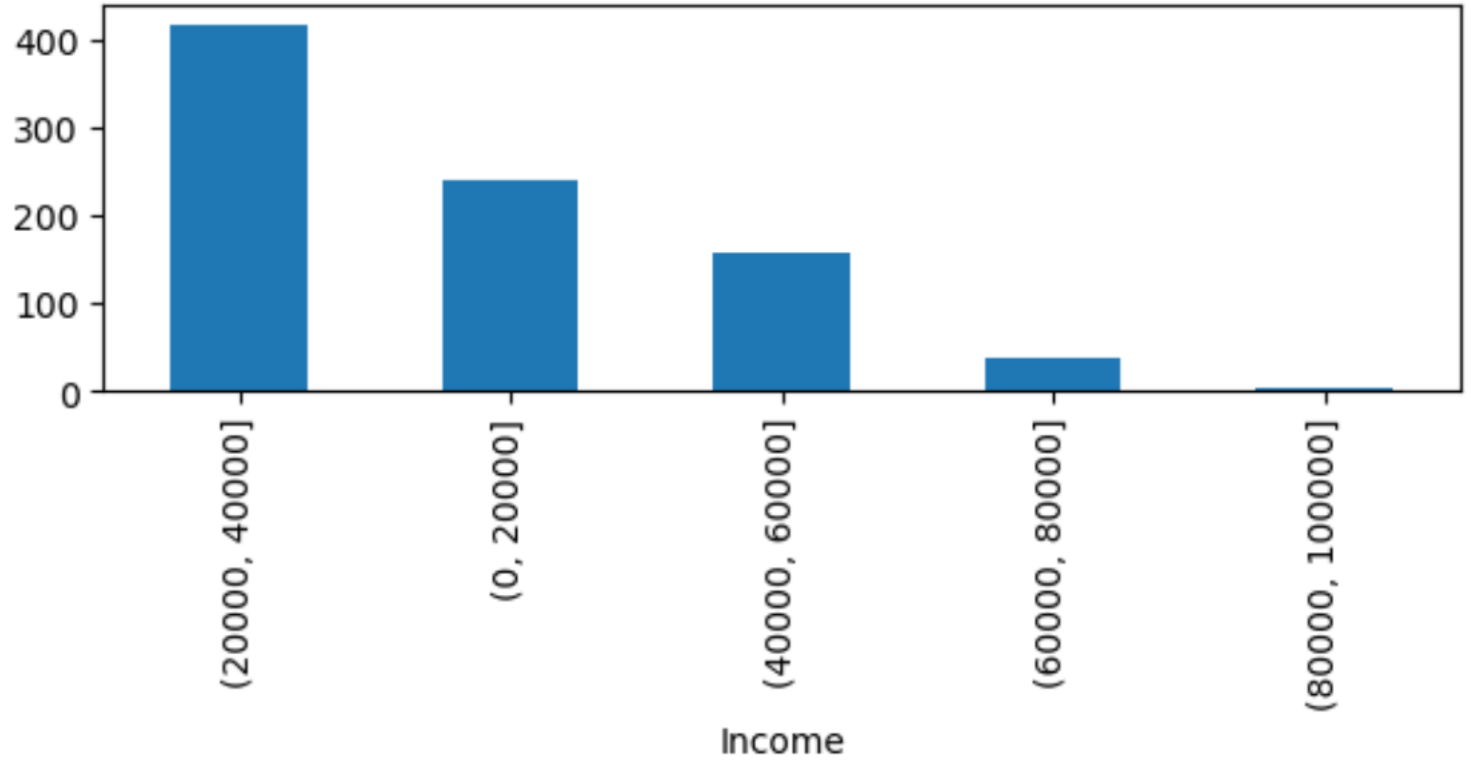
	ID	Income	Term	Balance	Debt	Score	Default
0	567	17500	6 Month Term	1460	272	225.0	False
1	523	18500	12 Month Term	890	970	187.0	False
2	544	20700	6 Month Term	880	884	85.0	False
3	370	21600	6 Month Term	920	0	NaN	False
4	756	24300	6 Month Term	1260	0	495.0	False

Discretisation & binning

```
pd.cut(x=df['Income'],
       bins=[0, 20_000, 40_000, 60_000, 80_000, 100_000]
      ).head()
```

```
0        (0, 20000]
1        (0, 20000]
2    (20000, 40000]
3    (20000, 40000]
4    (20000, 40000]
Name: Income, dtype: category
Categories (5, interval[int64, right]): [(0, 20000] < (20000, 40000] < (40000, 60000] < (60000, 80000] < (80000, 1
00000]]
```

```
pd.cut(x=df['Income'],
       bins=[0, 20_000, 40_000, 60_000, 80_000, 100_000]
      ).value_counts().plot(kind='bar', figsize=(7,2));
```



Deriving new columns

```
df['DebtAssetRatio'] = df['Debt'] / df['Balance']
df.head()
```

	ID	Income	Term	Balance	Debt	Score	Default	DebtAssetRatio
0	567	17500	Short Term	1460	272	225.0	False	0.186301
1	523	18500	Long Term	890	970	187.0	False	1.089888
2	544	20700	Short Term	880	884	85.0	False	1.004545
3	370	21600	Short Term	920	0	NaN	False	0.000000
4	756	24300	Short Term	1260	0	495.0	False	0.000000

Applying functions over columns

```
df['Debt'].tail()
```

```
851    3779
852       0
853    3032
854    2498
855    2355
Name: Debt, dtype: int64
```

```
df['Debt'].map(lambda debt: 'High' if debt > 1000 else 'Low').tail()
```

```
851    High
852     Low
853    High
854    High
855    High
Name: Debt, dtype: object
```

Applying functions over columns

```
df.select_dtypes(np.number).apply(lambda col: col.round(2))
```

	ID	Income	Balance	Debt	Score	DebtAssetRatio
0	567	17500	1460	272	225.0	0.19
1	523	18500	890	970	187.0	1.09
2	544	20700	880	884	85.0	1.00
3	370	21600	920	0	NaN	0.00
4	756	24300	1260	0	495.0	0.00
...	...	...	...	...	...	...
851	71	30000	1270	3779	52.0	2.98
852	932	42500	1550	0	779.0	0.00
853	39	36400	1830	3032	360.0	1.66
854	283	42200	1500	2498	417.0	1.67
855	847	30800	1190	2355	177.0	1.98

Working with text data

Text data

Text data provides unique challenges and needs specific processing and preparation.

Pandas can use Pythons string methods.

Pandas also implements regular expression functions.

- These allow you to do anything with text!

String methods

```
df['Term'].str.lower().head()
```

```
0     short term
1      long term
2     short term
3     short term
4     short term
Name: Term, dtype: object
```

```
df['Term'].str.find('Long').head()
```

```
0   -1
1    0
2   -1
3   -1
4   -1
Name: Term, dtype: int64
```

```
df['Term'].str.isalpha().head()
```

```
0     False
1     False
2     False
3     False
4     False
Name: Term, dtype: bool
```

Regular expressions (Regex)

```
df['Term'].str.findall(r'(Long|Short) (Term)')
```

```
0       [(Short, Term)]
1        [(Long, Term)]
2       [(Short, Term)]
3       [(Short, Term)]
4       [(Short, Term)]
             ...
851      [(Long, Term)]
852      [(Long, Term)]
853      [(Long, Term)]
854      [(Long, Term)]
855      [(Long, Term)]
Name: Term, Length: 856, dtype: object
```

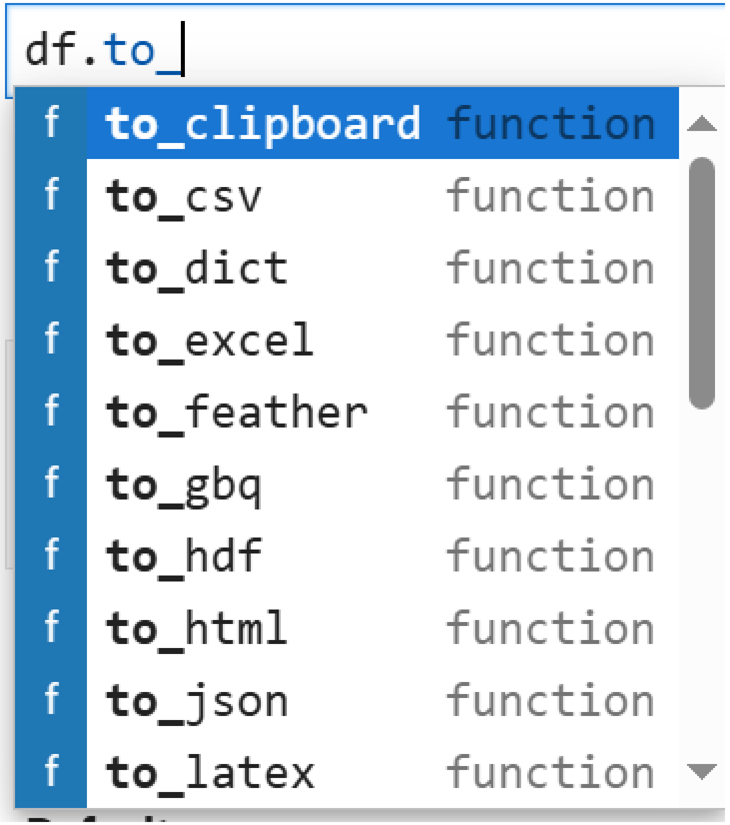
Elementary extended RE meta-characters

`.`	match any single character	Character Classes
`[a-zA-Z]`	match any char in the `[...]` set	
`[^a-zA-Z]`	match any char *not* in the `[...]` set	
`^`	match beginning of text	Anchors
`$`	match end of text	
x`?`	match 0 or 1 occurrences of *x*	Quantifiers
x`+`	match 1 or more occurrences of *x*	
x`*`	match 0 or more occurrences of *x*	
x`{`*m*`,`*n*`}`	match between *m* and *n* *x*'s	
`abc`	match `abc`	
`abc|xyz`	match `abc` or `xyz`	Alternation

Writing DataFrames to Files

```python
df.to_csv("data/cleaned_df.csv")
```

```python
df.to_json("data/cleaned_df.json")
```



Exercise

Go to **Exercise: Data cleaning with Pandas** in your exercise guide.

Learning check

Think about your answers to these questions:

- Why do we treat missing data? How can we do it with Python?
- Which Python functions can we use to identify duplicates?
- How can we alter column values?
- How can we process text using Pandas?
- What are regular expressions?

How did you get on?

Learning objectives

- Identify missing data and apply techniques to deal with it.
- Deduplicate, transform, and replace values.
- Use DataFrame string methods to manipulate text data.
- Write regular expressions which munge text data.

POWERING
POTENTIAL

6. Data Manipulation with Pandas

Data manipulation with Pandas

Learning objectives

- Construct Pivot tables in Pandas.
- Time series manipulation.
- Stream data into Pandas to handle data size problems.

Expected prior knowledge

- Experience of working with data using data analysis tools such as Excel.

Pivot tables

Creating pivot tables

- Pivot_table can be used to construct Excel style pivot tables in Pandas.
  - View statistics across category groups.

```
df.pivot_table(values=['Income'],
               index='Default',
               columns='Term').round()
```

	Income	
Term	**Long Term**	**Short Term**
Default		
False	34819.0	28190.0
True	31287.0	23888.0

```
df.pivot_table(values=['Income'],
               index=['Default', df['Balance'].map(lambda balance : '>1000' if balance > 1000 else '0-999')],
               columns='Term').round()
```

		Income	
	Term	**Long Term**	**Short Term**
Default	**Balance**		
False	**0-999**	26165.0	22914.0
	>1000	38015.0	31522.0
True	**0-999**	25660.0	20136.0
	>1000	32369.0	26246.0

Time series

Time data

Time data provides unique challenges:

- Unique mathematical rules.
- Different formats.
- Periods at can be defined at multiple levels.
  - E.g., seconds, hours, days, weeks, etc.
- Time zones.

Pandas has tools to handle these issues.

Indexing by time

```
df = pd.read_json(json.dumps(weather), orient="split")
df
```

	temp	humidity	sun_hrs
2023-07-15	29.83	79.43	8.57
2023-07-16	32.94	75.12	10.49
2023-07-17	28.86	78.19	10.41
2023-07-18	30.37	83.87	9.43
2023-07-19	31.15	81.41	10.01
2023-07-20	33.50	82.00	10.80
2023-07-21	30.06	75.28	8.54
2023-07-22	26.86	82.26	9.50
2023-07-23	30.78	80.98	10.28
2023-07-24	27.13	86.67	10.43

```
df.index
```

```
DatetimeIndex(['2023-07-15', '2023-07-16', '2023-07-17', '2023-07-18',
               '2023-07-19', '2023-07-20', '2023-07-21', '2023-07-22',
               '2023-07-23', '2023-07-24', '2023-07-25', '2023-07-26',
               '2023-07-27', '2023-07-28', '2023-07-29', '2023-07-30',
               '2023-07-31', '2023-08-01', '2023-08-02', '2023-08-03',
               '2023-08-04', '2023-08-05', '2023-08-06', '2023-08-07',
               '2023-08-08', '2023-08-09', '2023-08-10', '2023-08-11',
               '2023-08-12', '2023-08-13'],
              dtype='datetime64[ns]', freq=None)
```

Slicing by time

```
df.loc['2023-07-15', :]
```

```
temp        32.18
humidity    71.25
sun_hrs     10.34
Name: 2023-07-15 00:00:00, dtype: float64
```

```
df.loc['2023-07-15':'2023-07-20', :]
```

	temp	humidity	sun_hrs
2023-07-15	32.18	71.25	10.34
2023-07-16	31.30	83.51	10.48
2023-07-17	30.95	75.79	10.47
2023-07-18	27.44	83.31	8.72
2023-07-19	30.73	84.17	8.61
2023-07-20	30.20	82.34	8.91

```
df.loc['2023-08', :]
```

	temp	humidity	sun_hrs
2023-08-01	29.65	80.34	9.19
2023-08-02	29.35	76.06	10.94
2023-08-03	32.96	77.64	9.37

Offsets and frequencies

- Date ranges can be defined very flexibly.
- Using Pandas' offsets, we can add intervals to times.

```
pd.date_range(start='2020',
              end='2024',
              freq='Q')
```

```
DatetimeIndex(['2020-03-31', '2020-06-30', '2020-09-30', '2020-12-31',
               '2021-03-31', '2021-06-30', '2021-09-30', '2021-12-31',
               '2022-03-31', '2022-06-30', '2022-09-30', '2022-12-31',
               '2023-03-31', '2023-06-30', '2023-09-30', '2023-12-31'],
              dtype='datetime64[ns]', freq='Q-DEC')
```

```
pd.date_range(start='2020',
              end='2024',
              freq='Q') + pd.tseries.offsets.Day(1)
```

```
DatetimeIndex(['2020-04-01', '2020-07-01', '2020-10-01', '2021-01-01',
               '2021-04-01', '2021-07-01', '2021-10-01', '2022-01-01',
               '2022-04-01', '2022-07-01', '2022-10-01', '2023-01-01',
               '2023-04-01', '2023-07-01', '2023-10-01', '2024-01-01'],
              dtype='datetime64[ns]', freq=None)
```

Dealing with time zones

```
pd.date_range(start='2020',
              end='2024',
              freq='Q',
              tz='UTC')
```

```
DatetimeIndex(['2020-03-31 00:00:00+00:00', '2020-06-30 00:00:00+00:00',
               '2020-09-30 00:00:00+00:00', '2020-12-31 00:00:00+00:00',
               '2021-03-31 00:00:00+00:00', '2021-06-30 00:00:00+00:00',
               '2021-09-30 00:00:00+00:00', '2021-12-31 00:00:00+00:00',
               '2022-03-31 00:00:00+00:00', '2022-06-30 00:00:00+00:00',
               '2022-09-30 00:00:00+00:00', '2022-12-31 00:00:00+00:00',
               '2023-03-31 00:00:00+00:00', '2023-06-30 00:00:00+00:00',
               '2023-09-30 00:00:00+00:00', '2023-12-31 00:00:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='Q-DEC')
```

```
pd.date_range(start='2020',
              end='2024',
              freq='Q',
              tz='UTC').tz_convert('Europe/Madrid')
```

```
DatetimeIndex(['2020-03-31 02:00:00+02:00', '2020-06-30 02:00:00+02:00',
               '2020-09-30 02:00:00+02:00', '2020-12-31 01:00:00+01:00',
               '2021-03-31 02:00:00+02:00', '2021-06-30 02:00:00+02:00',
               '2021-09-30 02:00:00+02:00', '2021-12-31 01:00:00+01:00',
               '2022-03-31 02:00:00+02:00', '2022-06-30 02:00:00+02:00',
               '2022-09-30 02:00:00+02:00', '2022-12-31 01:00:00+01:00',
               '2023-03-31 02:00:00+02:00', '2023-06-30 02:00:00+02:00',
               '2023-09-30 02:00:00+02:00', '2023-12-31 01:00:00+01:00'],
              dtype='datetime64[ns, Europe/Madrid]', freq='Q-DEC')
```

Time periods

- Datetimes can be converted to periods
  - E.g., months.
- Index doesn't need to be unique!
  - Multiple values returned at for each period.

```
df.to_period('M').sample(5)
```

	temp	humidity	sun_hrs
2023-07	28.30	81.01	9.26
2023-07	30.18	74.20	10.47
2023-07	30.95	75.79	10.47
2023-08	28.55	81.74	9.27
2023-08	28.91	75.76	8.89

Moving window functions

- Window functions allow evaluation over sets of rows.
- Windows can be static row sets or dynamic periods.

```
df.rolling(window=7,
           min_periods=2).mean().round(2).head()
```

	temp	humidity	sun_hrs
2023-07-15	NaN	NaN	NaN
2023-07-16	31.74	77.38	10.41
2023-07-17	31.48	76.85	10.43
2023-07-18	30.47	78.46	10.00
2023-07-19	30.52	79.61	9.72

```
df.rolling(window='15D',
         min_periods=15).mean().round(2).tail()
```

	temp	humidity	sun_hrs
2023-08-09	30.25	82.11	9.91
2023-08-10	30.35	82.58	9.78
2023-08-11	30.26	82.00	9.79
2023-08-12	30.27	82.05	9.79
2023-08-13	30.18	82.21	9.78

Combining tables

Merging DataFrames

- Merge allows for SQL like joins between DataFrames.
- Used to combine tables based on condition.

```
df = pd.read_csv("data/loan_data.csv")

df[:2]
```

	ID	Income	Term	Balance	Debt	Score	Default
0	567	17500.0	Short Term	1460.0	272.0	225.0	False
1	523	18500.0	Long Term	890.0	970.0	187.0	False

```
locations.head()
```

	ID	nation
0	567	Scotland
1	523	England
2	544	England
3	370	England
4	756	England

```
pd.merge(left=df,
         right=locations,
         on='ID').head()
```

	ID	Income	Term	Balance	Debt	Score	Default	nation
0	567	17500	Short Term	1460	272	225.0	False	Scotland
1	523	18500	Long Term	890	970	187.0	False	England
2	544	20700	Short Term	880	884	85.0	False	England
3	370	21600	Short Term	920	0	NaN	False	England
4	756	24300	Short Term	1260	0	495.0	False	England

Merging multiple DataFrames

Chained merges join further tables.

```
pd.merge(left=df,
         right=locations,
         on='ID').merge(right=business_accounts,
                        on='ID').head()
```

	ID	Income	Term	Balance	Debt	Score	Default	nation	has_business_account
0	567	17500	Short Term	1460	272	225.0	False	Scotland	False
1	523	18500	Long Term	890	970	187.0	False	England	False
2	544	20700	Short Term	880	884	85.0	False	England	False
3	370	21600	Short Term	920	0	NaN	False	England	True
4	756	24300	Short Term	1260	0	495.0	False	England	False

Concatenating DataFrames

Concat sticks DataFrames together without a condition.

```
df.tail()
```

	temp	humidity	sun_hrs
2023-08-09	27.05	80.59	10.25
2023-08-10	31.08	80.66	8.84
2023-08-11	29.09	78.38	9.77
2023-08-12	30.33	72.94	10.38
2023-08-13	30.69	78.45	10.48

```
df_next.head()
```

	temp	humidity	sun_hrs
2023-08-15	30.04	74.26	8.14
2023-08-16	25.62	78.69	9.77
2023-08-17	24.91	84.11	9.29
2023-08-18	26.85	79.82	11.12
2023-08-19	29.54	81.25	9.86

```
pd.concat([df, df_next]).loc['2023-08-12':'2023-08-19', :]
```

	temp	humidity	sun_hrs
2023-08-12	30.33	72.94	10.38
2023-08-13	30.69	78.45	10.48
2023-08-15	30.04	74.26	8.14
2023-08-16	25.62	78.69	9.77
2023-08-17	24.91	84.11	9.29
2023-08-18	26.85	79.82	11.12
2023-08-19	29.54	81.25	9.86

Splicing together DataFrames

Used to fill in gaps in indices.

```
incomplete_df
```

	temp	humidity	sun_hrs
2023-08-12	30.33	72.94	10.38
2023-08-13	30.69	78.45	10.48
2023-08-15	30.04	74.26	8.14
2023-08-16	25.62	78.69	9.77
2023-08-17	24.91	84.11	9.29
2023-08-18	26.85	79.82	11.12
2023-08-19	29.54	81.25	9.86

```
missing_vals
```

	temp	humidity	sun_hrs
2023-08-14	28.4	77.86	9.80
2023-08-15	28.0	76.43	9.13

```
incomplete_df.combine_first(missing_vals)
```

	temp	humidity	sun_hrs
2023-08-12	30.33	72.94	10.38
2023-08-13	30.69	78.45	10.48
2023-08-14	28.40	77.86	9.80
2023-08-15	30.04	74.26	8.14
2023-08-16	25.62	78.69	9.77
2023-08-17	24.91	84.11	9.29
2023-08-18	26.85	79.82	11.12
2023-08-19	29.54	81.25	9.86

Exercise

Go to **Exercise: Data manipulation with Pandas** in your exercise guide.

Learning check

Think about your answers to these questions:

- What do Pivot tables do?
- What are common time data problems?
- What does Pandas offer to deal with large files?

How did you get on?

Learning objectives

- Construct Pivot tables in Pandas.
- Time series manipulation.
- Stream data into Pandas to handle data size problems.