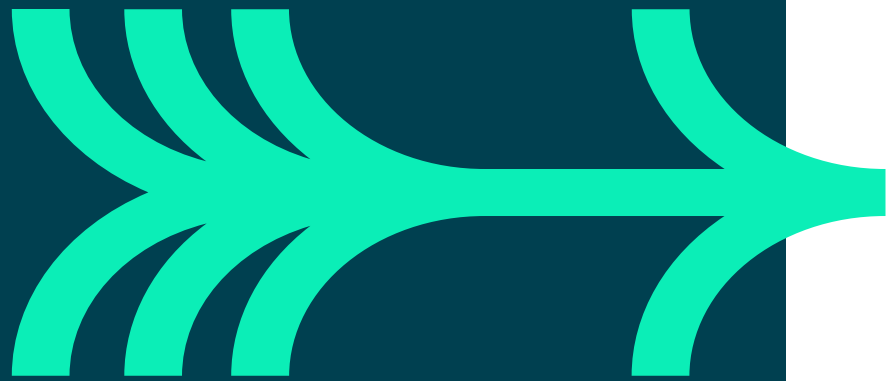
POWERING POTENTIAL

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.

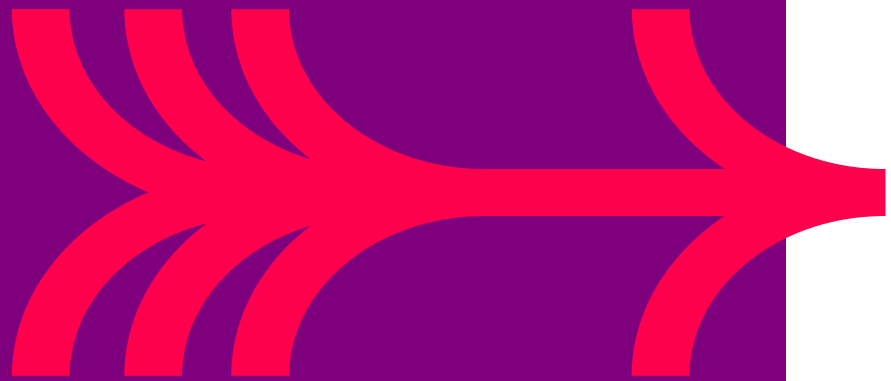
Expected prior knowledge

- 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

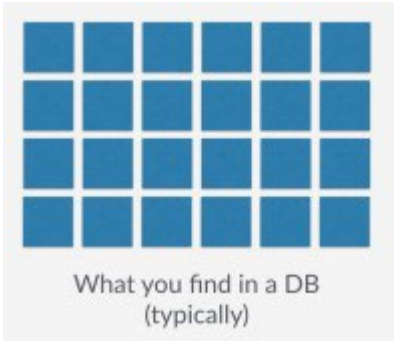
Collected from where?

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

What are the data types (forms)?

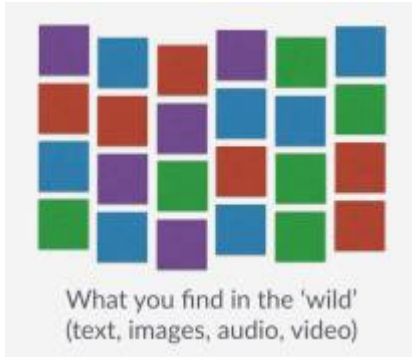
Structured data

- Tabular
- SQL Databases



Unstructured data

- Photos
- Sounds
- Videos



Which tool for data?

Commonly used data tools

Programming languages



Databases


Command line tools



Spreadsheets

Business intelligence tools

Commonly used programming languages

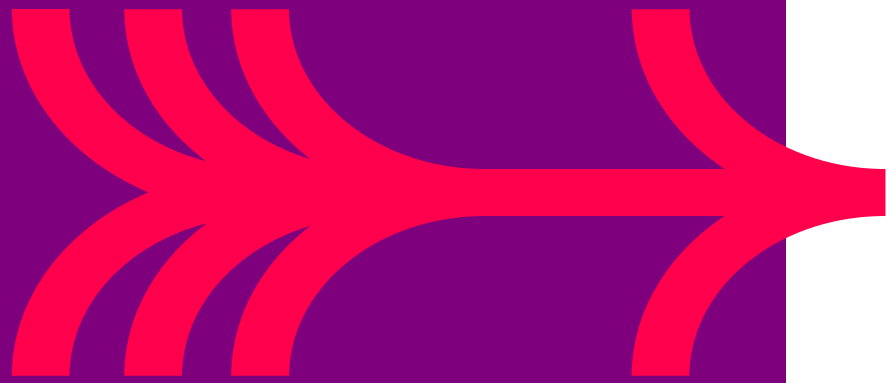
- Python
  - ...and the NumPy ecosystem
- R
- Scala
- Julia
- VBA
- DAX
- Go

- Almost always high-level

Activity: Research

- Research each of the following popular languages with a partner:
  - Python
  - R
  - Scala
  - Julia

- Identify their strengths and weaknesses.



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.

Python IDEs

Integrated development environments for Python

- When working with programming languages, we tend to use Integrated Development Environments (IDEs).

- These typically combine the below into a single application:
  - Code editors
  - Environment management
  - Execution
  - File explorers

Popular Python IDES

- **Jupyter:** For writing .ipynb files.
  - Jupyter Notebook
  - Jupyter Lab

- **PyCharm**: For Python software development.

- **Vscode**: For generic software development.

Choosing an IDE

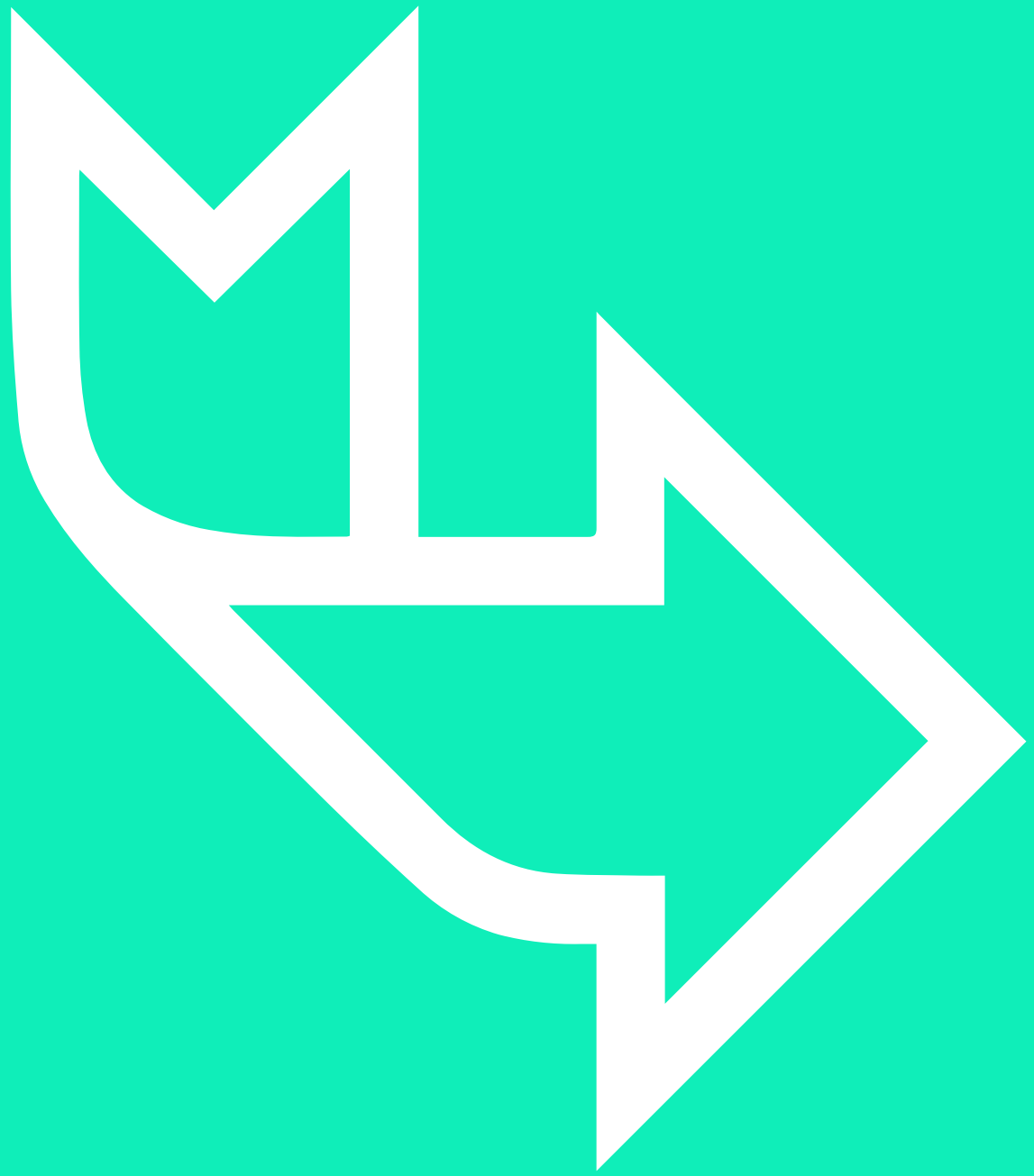
- Python data programs have c. 10s-1,000s of lines.
  - I.e. not a lot.
  - Notebooks are optimal.

- Python software applications have c. 1,000s-1,000,000s of lines.
  - I.e. a lot.
  - ‘Traditional’ IDEs are optimal.

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?

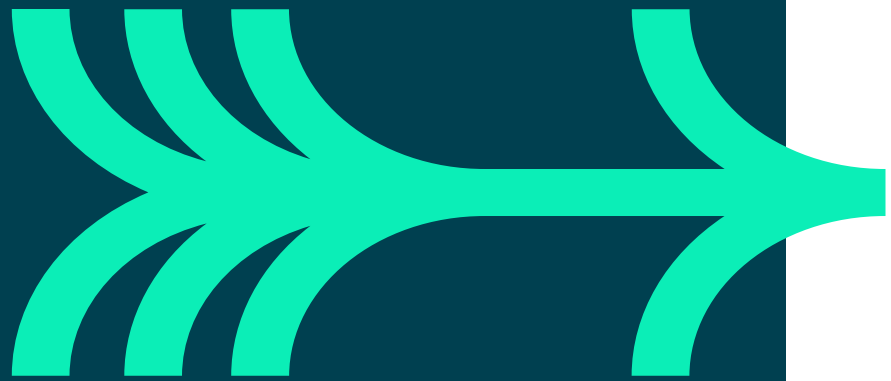
Learning objectives

- 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.

POWERING POTENTIAL

2. Introduction to Python and IDEs

Introduction to Python & IDEs



Learning objectives

- Describe the key attributes of the Python programming language.
- Explain the role of the Jupyter IDE for Python programming.
- Use the Jupyter IDE to write a basic Python program.
- Write a program which uses string, integer, float, and boolean data types.

Expected prior knowledge

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

The Python programming language

Activity: Discussion

- What do you know about the Python programming language?

- For example:
  - What type of language is it?
  - Is it high or low level?
  - How is it organised?
  - Is it compiled or interpreted?
  - How does it enforce type?

What is Python?

Python is an object-oriented scripting language.

- First published in 1991 by Guido van Rossum.
- Designed as an OOP language from day one.
- Does not need knowledge of OO to use.

It is powerful.

- General purpose, fully functional, rich.
- Many extension modules.

It is free.

- Open source: Python licence is less restrictive than GPL.

It is portable.

- UNIX, Linux, Windows, OS X, Android, etc...
- Ported to the Java and .NET virtual machines.

The most common version is written in C.

- No platform specific functions in the base language.

Why Python?

Power

- Garbage collector .
- Native Unicode objects.

Supports many component libraries

- GUI libraries like wxPython, PyQt.
- Scientific libraries like NumPy, DISLIN, SciPy, and many others.
- Mature JSON support.
- Web template systems such as Jinja2, Mako, and many others.
- Web frameworks such as Django, Pyjamas, and Zope.
- Relational Databases support.
- Libraries may be coded in C or C++.

It is ‘easy-’

- -er than other, stricter languages.

Performance downsides

Performance

- Python programs are compiled at runtime into ‘byte code’.
- Byte code does not compile down to PE or ELF.
- Cannot be a fast as well written C.
- Worse CPU usage than Java and C#.
  - On most benchmarks, but not all.
  - Better memory management.

But...

- Roughly equivalent to Perl and Ruby - depending on the benchmark.
- Packages often use compiled extensions.
- Typically, much faster to write.

Virtual environments

Virtual environments

- Best practice is one virtual environment per project.
- A folder containing everything your project needs.
  - Including Python!
  - Modules and packages.
- Managed using package manager.
  - PIP.
  - Conda.

Anaconda & Jupyter

Anaconda and Jupyter

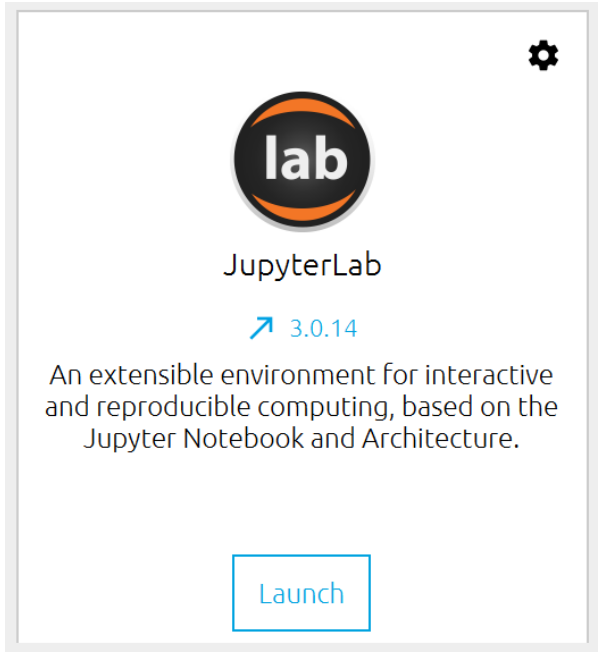
Anaconda is a collection of Python packages, programs, and other software for data analysis and data science.

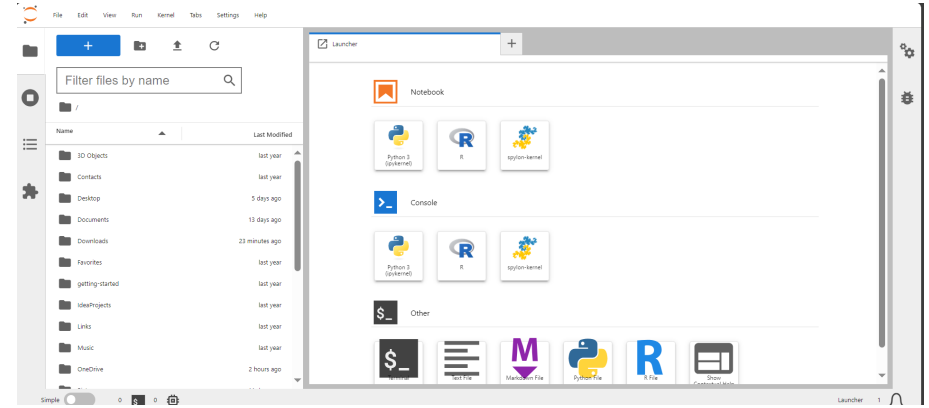
A Jupyter notebooks are a documentation or commentary-first style of programming, where the key elements are explanations and comments. The code is less important, and often only small amounts.

Why JupyterLab?

JupyterLab is a superb web-based interactive application for working with Jupyter Notebooks.

- User Friendly
- Easy to debug at each line of code





Anaconda navigator

How do I start Anaconda?

Start Menu > Anaconda Navigator

How do I start JupyterLab?

In Anaconda Navigator, press LAUNCH underneath the JupyterLab icon.

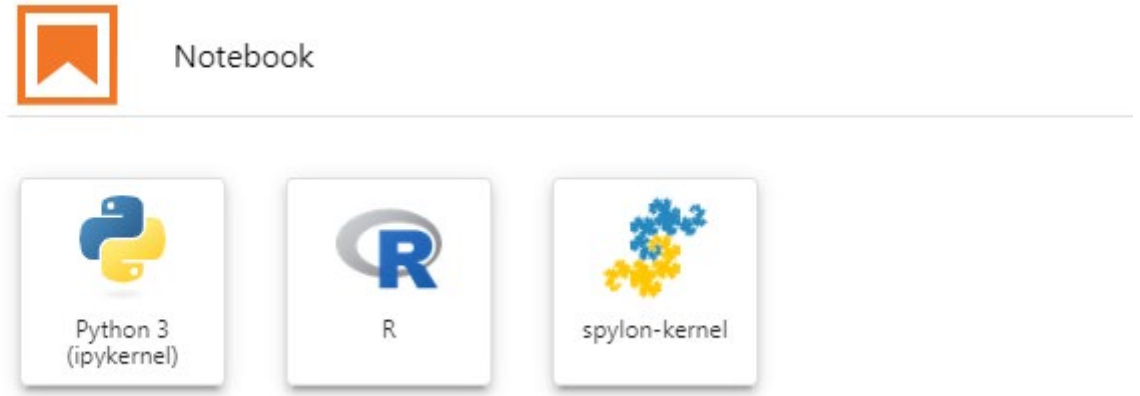
Opening Jupyter

How do I use the file browser?

The file browser can be popped out using the file icon. You can then navigate around, find a notebook to open, or create a new notebook.

How do I create notebooks?

Navigate to the labs folder, press the icon, then select Python3 in the launcher.

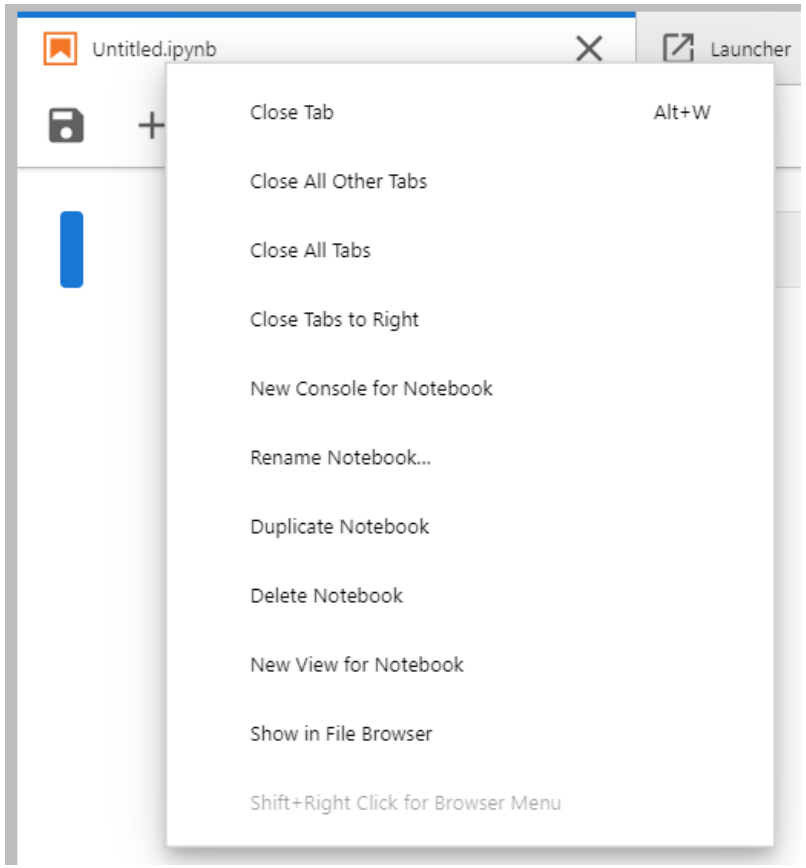


Jupyter will create a new notebook called ‘Untitled’.

Renaming Notebooks

How do I rename a notebook?

Right-click on the title and select rename. Choose a relevant title, e.g., ‘MyNotes’.



Working with cells

How do I add and modify cells?

Command mode:	ESC (blue bar)
Arrows:	Move around notebook
m:	Markdown / text cell
y:	Code cell / Python cell
a:	Insert cell above
b:	Insert cell below
dd:	Delete cell
z:	Undo delete

Ctrl + Enter to run.
Shift + Enter to run **and** move one cell below.

Working with cells

How do I edit the contents of cells?

- Edit mode: **Enter** (green bar).
- Type.
- Use arrows to move around text.
- **Ctrl** + **Enter**: Run cell.
- **Shift** + **Enter**: Run cell and move one cell below.

Working with cells

How do I add Python code to a notebook?

- Add a cell (e.g., press '**b**') and press **Enter** to edit.

How do I run a Python code cell?

- Press **Ctrl** + **Enter** or the **RUN** button.

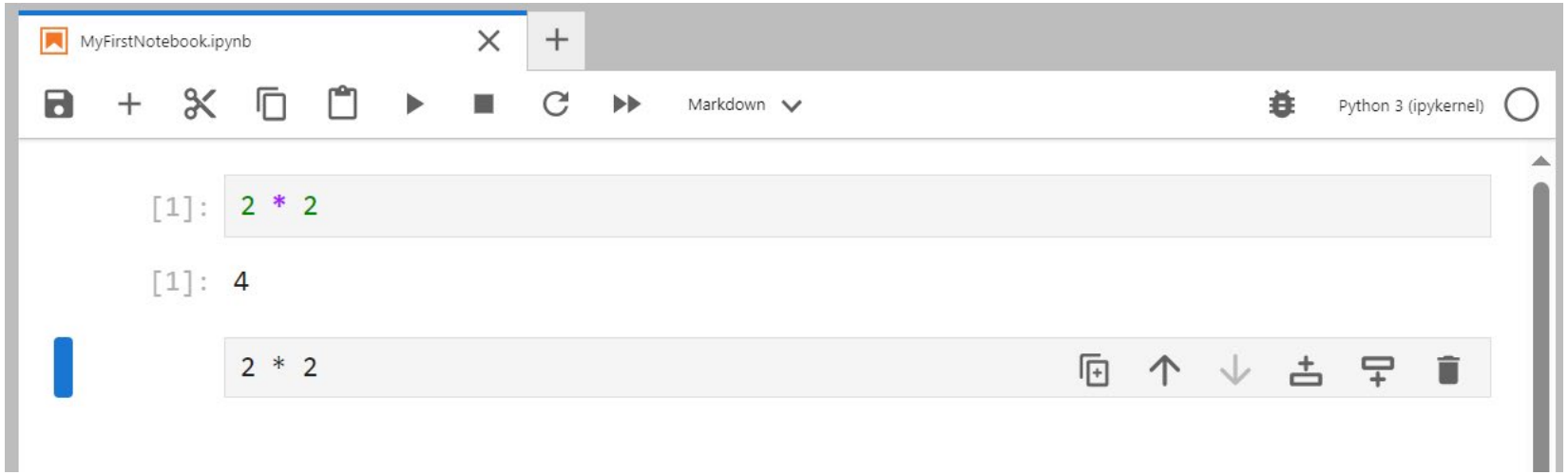
What happens when I run a code cell in Jupyter?

- Hopefully, you don't get any errors!
- Jupyter always 'print()'s the last line.
- In a usual Python IDE, you would 'print()' everything you wish to show on the screen.
  (What does IDE stand for?)

Working with cells

How do I add a text cell?

- Add a cell, change type to markdown.
- Press '**m**' in **command mode**.



How do I get help?

- 'command?'
- 'command??'
- 'help(command)'

Working with cells

How do I find the arguments of a function?

- Import statistics
- statistics.mean?

How do I stop Jupyter from printing the last line of a cell?

- Use ‘;’

How do I run command line programs within Jupyter?

- Exclamation mark
- !dir

How do I install Python packages with Jupyter?

- !conda install plotly pyspark

Working with supplied notebooks

Steps:

1. Open the notebook.
2. Select **File** in the top left, then **Duplicate Notebook**.
3. You can rename this new file by clicking on the file name at the top.
4. Then work from this new notebook:
    - It will retain all of the original code; in case you accidentally edit or delete.
    - it will save some guidance outputs that may be inadvertently overwritten.

Python fundamentals

Variables & objects

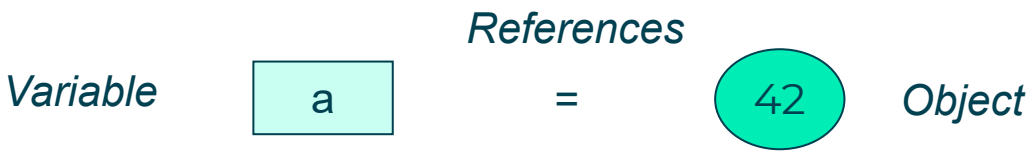
What is an object?

- To a programmer the term describes a specific area of memory.
- Objects have type, state, and identity.

If that is an object, what is a variable?

An object's type is called its class

- Describes the size and format of the area of memory.
- Describes the actions which may be carried out on the object.



Python variables are references to objects

- Variables can be deleted with `del`.
- An object's memory can be reused when it is no longer referenced.

Variable names

Case sensitive

- Must start with an underscore or a letter.
- Followed by any number of letters, digits, or underscores.

Conventions with underscores

- Underscores represent spaces in names.
- Names beginning with one underscore are ‘private’.

The character _ represents the result of the previous command.

Data structures in Python

Data structures in Python:

- Primitive data types.
- Complex data types (collections).

Data structures in Python at a glance:

- Primitive data types: integer, string, Boolean, float.
- Complex data types (collections).

list	Ordered	Changeable	Allow duplicates
tuple	Ordered	Unchangeable	Allow duplicates
set	Unordered	Changeable	No duplicates
dictionary	Unordered / ordered (Python 3.7)	Changeable	No duplicates

Changeable = mutable

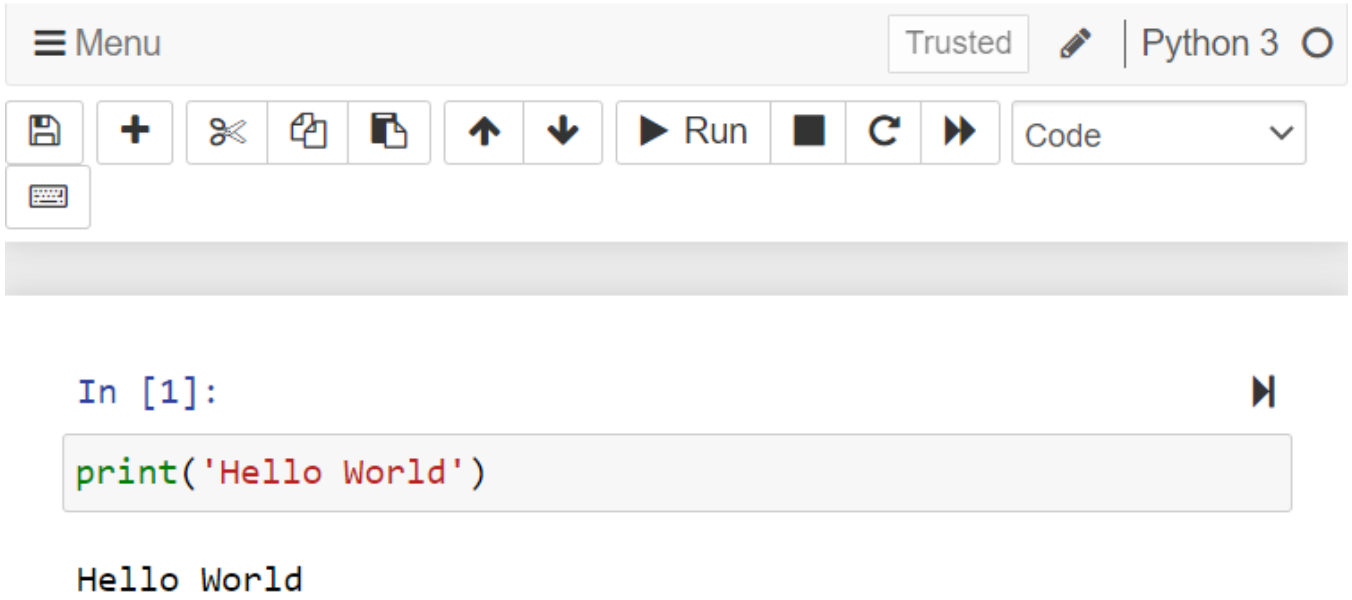
Unchangeable = immutable

Activity: Tutorial

- Code along with the trainer for the following tutorial.
- Ask questions if you encounter errors.
- Think about the code you are writing.

Printing to console window

- print ('Hello World').
- Click ▶ Run or **Shift** + **Enter** to run.



Data types in Python

There are three basic variable types:

- Numbers: Integer and Float
  - **1,2,3, 1.23, 0.0005**
- Character or String
  - **'Hello world' or "Hello world"**
- Boolean **True** or **False** (case-sensitive)

```
age=21
salary = 2000.78
company_name='QA Ltd'
is_registered = True
has_icence = False
```

Type is determined automatically; value can change.

Arithmetic operations

The arithmetic operations in Python are:

- Addition **+**
- Subtraction **–**
- Multiplication *****
- Division **/**
- Floor (integer) division **//**
- Modulus (modular division) **%**
- Exponentiation (raising to a power) ******

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.

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))
```

Careful! Python reserved words

py3

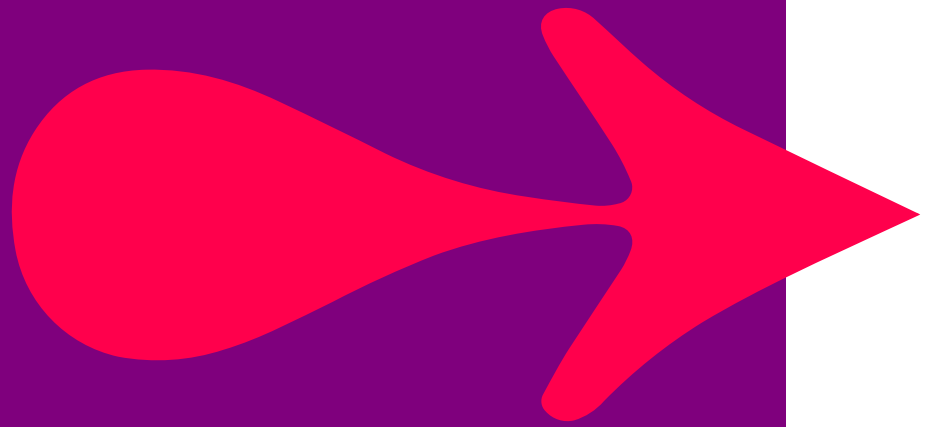
False	None	True	and	as*	assert
async^	await^	break	class	continue	def
del	elif	else	except	exec~	finally
for	from	global	if	import	in
is	lambda	nonlocal+	not	or	pass
raise	return	try	while	with*	yield

* version 2.6 and later
+ version 3.0
~ not in version 3.0
^ version 3.7

exec and **print** were keywords prior to 3.0, now they are built-in functions.

Exercise

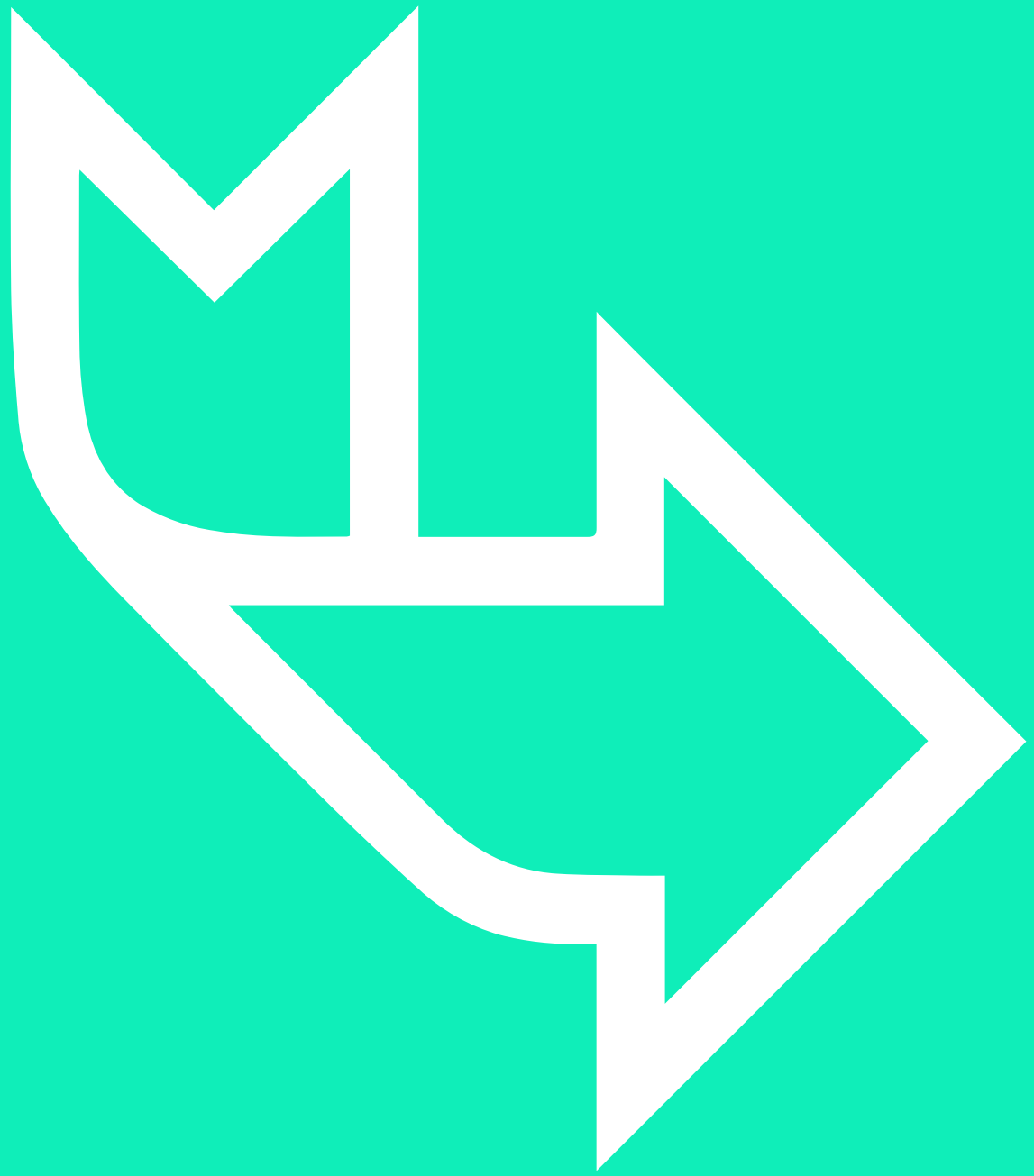
Go to **Exercise 2: Introduction to Python and IDEs** in your exercise guide.



Learning check

Think about your answers to these questions:

- What are the key attributes of the Python programming language?
- What is the Jupyter IDE? Why do we use it?
- What are Python's core data types? What are their properties?
- What do we call changing the type of an object?



How did you get on?

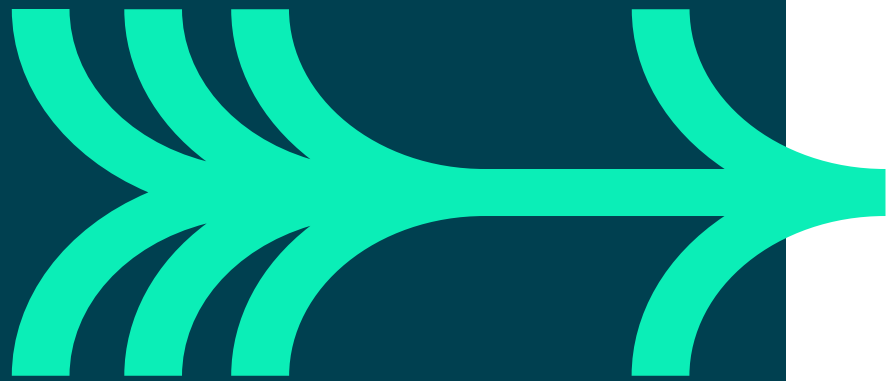
Learning objectives

- Describe the key attributes of the Python programming language.
- Explain the role of the Jupyter IDE for Python programming.
- Use the Jupyter IDE to write a basic Python program.
- Write a program which uses string, integer, float, and boolean data types.

POWERING POTENTIAL

3. Data Structures, Flow Control, Functions, & Basic Types

Data structures, flow control, functions, & basic types



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.

Expected prior knowledge

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

Collections

Python 3 types

- Numbers

```
3.142, 42, 0x3f, 0o664
```

Sequences

- 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')
```

Immutable

- Lists

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

- Bytearrays

```
bytearray(b'abc')
```

Mutable

- Dictionaries

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

- Sets

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

Python lists

Lists store multiple values (elements)

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

1
3
5
7

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

Bob
Steve
Helen

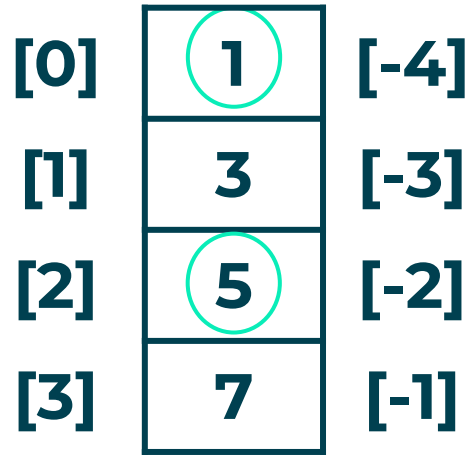
Lists can store elements of any data type, including other lists.

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

1
3.14
“fruit”
True

Addressing an element using its index

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

print(numbers[0])
print(numbers[2])
print(numbers[-2])
```




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

print(names[1])
Print(names[-1])
```

Applies to strings (lists of characters) too

```
s = ‘Hello world!’
print(s[1])
print(s[-5])
```

Addressing multiple elements - slicing

identifier[start:stop:step]

- ***stop*** is the position after the last index.
- Default ***start*** is 0.
- Default ***stop*** is the last index.
- Default ***step*** is 1.

```
numbers = [1,3,5,7,9,11,13,15,17,19,21]

print(numbers[3:10])
print(numbers[2:])
print(numbers[:6])
Print(numbers[::2])
```

```
Applies to strings (lists of characters) too
s = 'Hello world!'
print(s[3:10])
print(s[:6])
Print(s[::-1])
```

Appending elements at the end

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

numbers.append(88)
```

[1, 3, 5, 7, 9, 88]

```
You can start with an empty List:

nos = []
nos.append(10)
nos.append(20)
nos.append(30)
```

[10, 20, 30]

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’, ‘Chris’, 'Helen']

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
```

```
# 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']

Tuples packing and unpacking

When we create a tuple, we normally assign values to it. This is called **packing** a tuple.

```
person = ('James', 'Bond', 7)
```

Extracting the values back into variables is called **unpacking**.

```
(Fname, Lname, ID) = person
print(Fname)
print(Lname)
print(ID)
```

```
James
Bond
7
```

Tuples packing and unpacking

If the number of variables is less than the number of values, you can add an asterisk ***** to the variable name and the values will be assigned to the variable as a list:

```
(Fname, *data) = person
print(Fname)
print(data)
```

```
James
['Bond', 7]
```

If the asterisk is added to another variable name than the last, Python will assign values to the variable until the number of values left matches the number of variables left.

```
person = ('James', 'Bond', 7, 'M')
```

```
(Fname, *data, boss) = person
print(Fname)
print(data)
print(boss)
```

```
James
['Bond', 7]
M
```

Dictionaries

A dictionary is a collection of unique keys, each referring to a value

- The key can be any immutable python object – often a string.
- The position of the value in memory is determined by the key.
- Dictionaries are **unordered** – they are not sequences.
- Dictionaries are similar to sets but are accessed by keys.

✓**Constructed from { }**

***varname* = {*key1*:*object1*,*key2*:*object2*,*key3*:*object3*,...}**

✓**Or using dict()**

***varname* = dict(*key1*=*object1*,*key2*=*object2*,*key3*=*object3*,...)**

✓**Accessed by key**

A key is usually a text string, or anything that yields a text string.

varname*[*key*] = *object

These produce identical dictionaries:

```
mydict = dict(Make='Audi', Model='A4',
Colour='Red', Doors = 4, New = True)

mydict = {'Make':'Audi', 'Model':'A4',
'Colour':'Red', 'Doors':4, 'New':True}
```

Dictionary values

Objects stored can be of any type:

- Lists, tuples, other dictionaries, etc…
- Can be accessed using multiple indexes or keys in `[ ]`.
- Add a new value just by assigning to it.

```
mydict = {'UK':['London’, 'Wigan’, 'Macclesfield’],
          'US':['Miami’, ‘Boston’, 'New York’]}
print(mydict['UK'][2])

homer = 1
print(mydict['US'][homer])

mydict['FR'] = ['Paris', 'Lyon', 'Bordeaux', ‘Lille']
for country in mydict.keys():
    print(country, ': ', mydict[country])
```

```
FR :  ['Paris', 'Lyon', 'Bordeaux', ‘Lille']
US :  ['Miami', ‘Boston', 'New York']
UK :  ['London', 'Wigan', 'Macclesfield']
```

Dictionary values

To add a key, assign using the new key.

To delete a key, use the `del` statement.

```
mydict['Mileage'] = '80k'
mydict['BHP'] = 160
del mydict[‘New’]
print(mydict)
```

To access a single element from a dictionary:

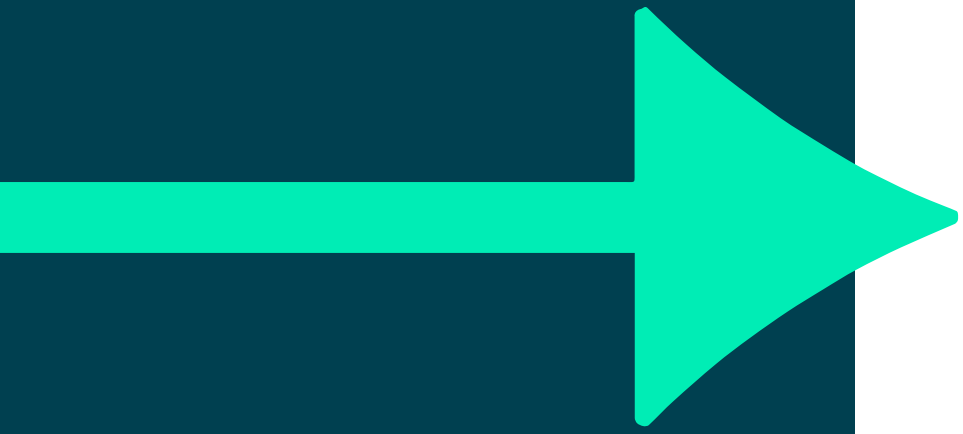
```
print('Colour is', mydict['Colour'])
```

There are several methods used for iterating over dictionaries:

- dict.keys(), dict.values(), dict.items()
- Remember that dictionaries are not ordered.

```
for key,value in mydict.items():
    print key, value
```

Removing items from a dictionary



To remove a single key/value pair:

- `del` *dict*`[`*key*`]` raises a KeyError exception if the key does not exist.
- *dict*`.pop(`*key*`[,`*default*`])` returns *default* if the key does not exist.

```
>>> fred={}
>>> del fred['dob']
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    del fred['dob']
KeyError: 'dob'
>>> fred.pop('dob', False)
False
```

Also:

- *dict*`.popitem()` removes the next key/value pair used in iteration.
- *dict*`.clear()` removes all key/value pairs from the dictionary.

Dictionary methods

dict.clear()	Remove all items from *dict.*
dict.copy()	Return a copy of *dict*.
dict.fromkeys(*seq*[,*value*])	Create a new dictionary from *seq.*
dict.get(*key*[,*default*])	Return the value for *key*, or *default* if it does not exist.
dict.items()	Return a view of the key-value pairs.
dict.keys()	Return a view of the keys.
dict.pop(*key*[,*default*])	Remove and return *key's* value, else return *default.*
dict.popitem()	Remove the next item from the dictionary.
dict.setdefault(*key*[,*default*])	Add *key* if it does not already exist.
dict.update(*dictionary*)	Merge another dictionary into *dict*.
dict.values()	Return a view of the values.

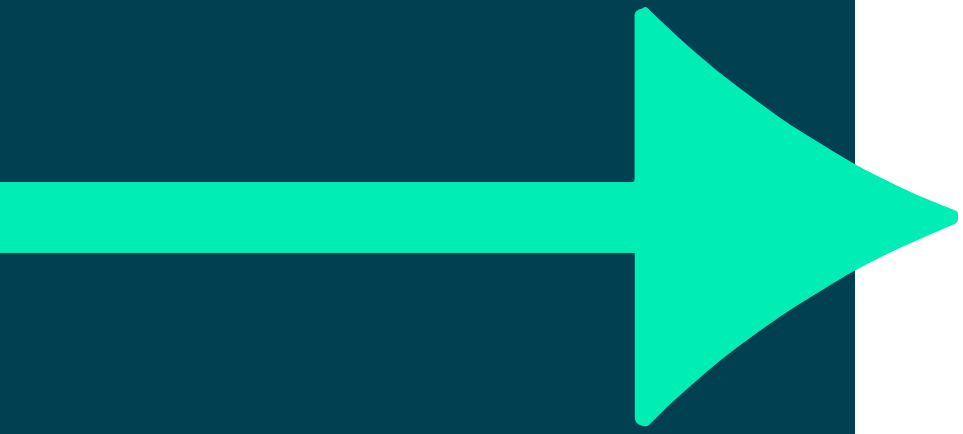
NOTE: Return values from `keys()`, `values()`, and `items()` are *view objects*.

Flow control

What is control flow?

Control flow controls the order in which we do things:

- **Sequence**
  - Running code step by step, in order.
- **Selection**
  - Deciding which lines of code should run.
- **Iteration**
  - Doing the same thing many times, i.e., in a loop.

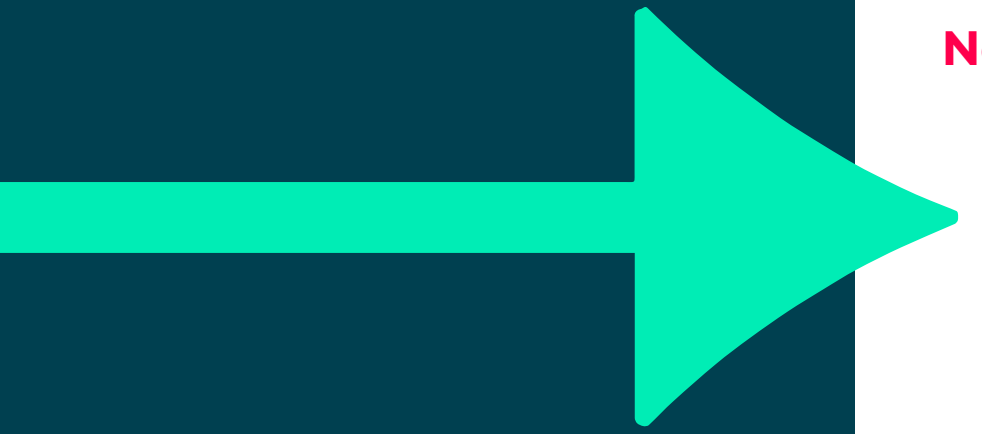


Control flow - selection

Making decisions

- **if:** do this.
- **else:** do something else.

Comparing values



Relational operators for making a selection:

>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to comparison
!=	Not equal to comparison

Note!

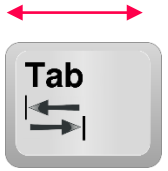
== is not the same as **=**

Selection example – IF statement

```
x = 5

if x == 5:
    print("X equals 5")
```

Condition:
Is x equal to 5?

Yes, it is
... so, do this.



Use the Tab key to include lines in an IF statement

* In Jupyter Notebooks, it automatically tabs for you.

Selection example – IF … ELSE

```
X = 5

if x == 5:
    print("X equals 5")
else:
    print("X does not equal 5")
```

Is x equal to 5?

Yes, it is... so, do this.

No, it isn't ... so, do this instead.

Control flow using ‘else if’ - elif

```
salary = 2500

if salary > 100000:
    print('Band A')
elif salary > 55000:
    print('Band B')
elif salary > 32000:
    print('Band C')
elif salary > 25000:
    print('Band D')
else:
    print('Band E')
```

Creates a chain of tests.

When a test passes, the other tests will not execute, and the statement ends.

The last else is optional.

The if-elif conditions must not overlap. Otherwise, the result will depend on their order, which is not a good practice.

Checking for existence

Use the **in** command to check if an item is present in a list.

```
adminIDs = [12,33,84,45,67,36,16,66,67,99]
id = int(input('Enter your ID '))
if id in adminIDs:
    print('Welcome!')
```

Numeric

```
names = ['David','John','Joanne','Sean','Sonia']
if 'Sean' in names:
    print('Sean is in the list!')
```

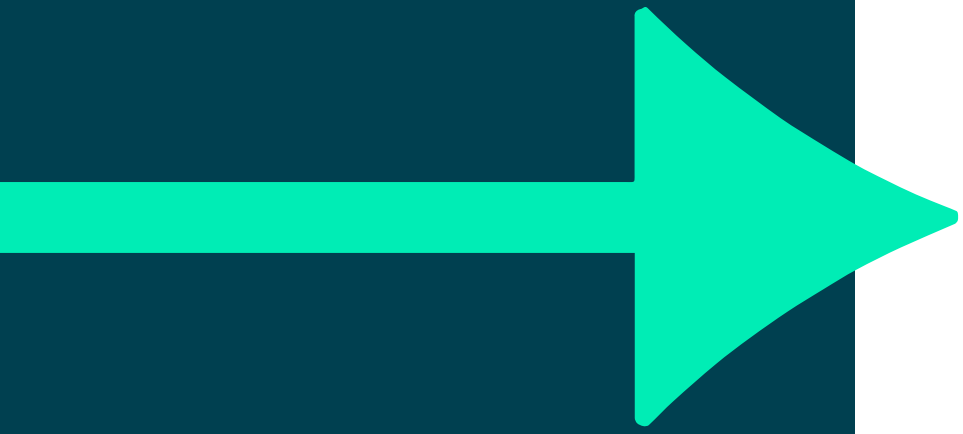
Strings

```
names = "David,John,Joanne,Sean,Sonia"
if 'John' in names:
    print('John is in the list!')
```

List of characters
(string)

Iterating through a list - For Loop

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

for name in names:
    print(name)
```



```
Bob
Steve
Helen

Press any key to continue . . .
```

Iterating through a list – While Loop

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

i=0     # used as index

while(i < len(names) ):
    print(names[i])
    i += 1
```

```
Bob
Steve
Helen

Press any key to continue . . .
```

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:

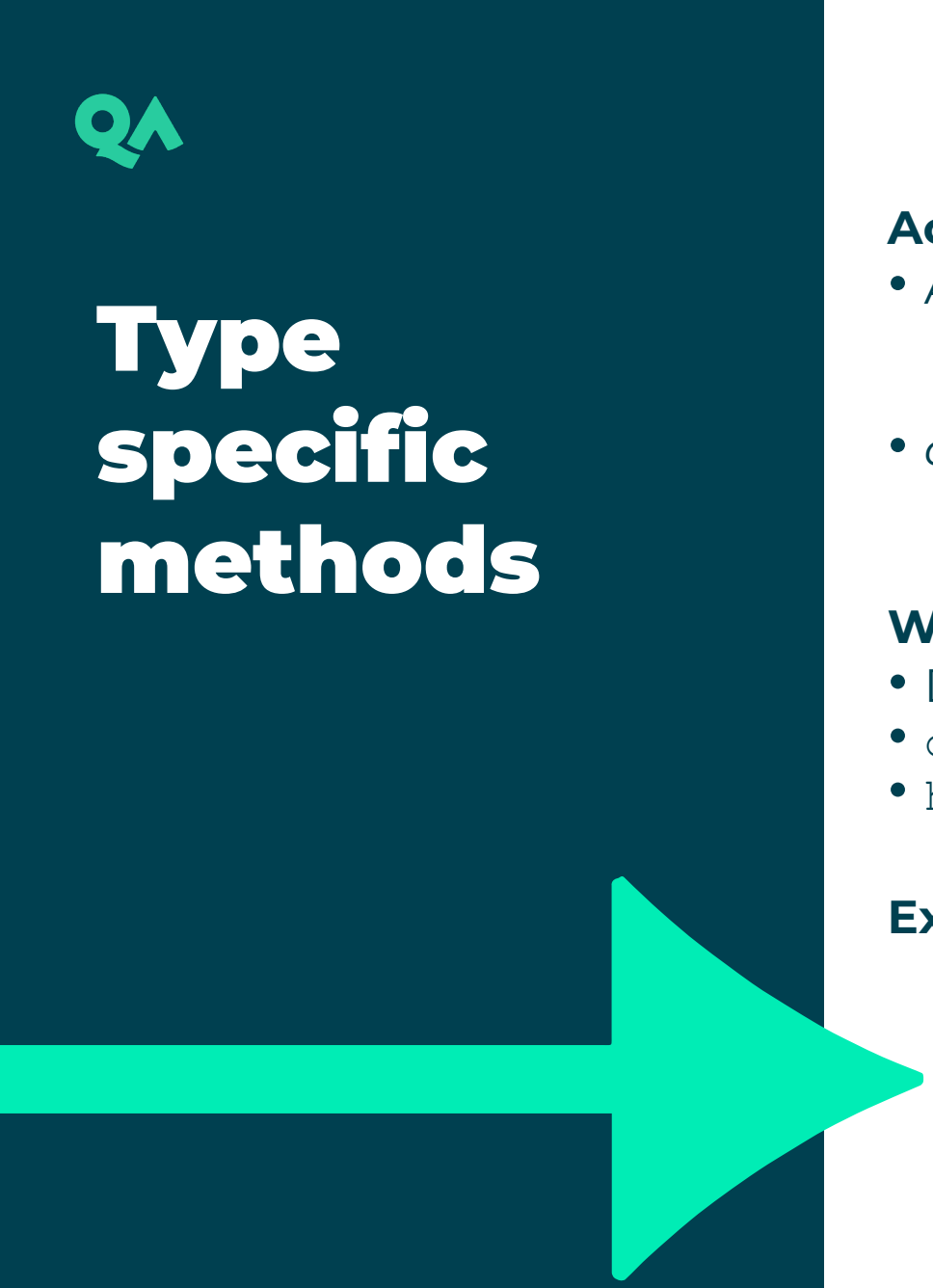
```
area = rectangle_area(5,2)
area
```

```
10
```

... or we can print it:

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

```
10
```



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

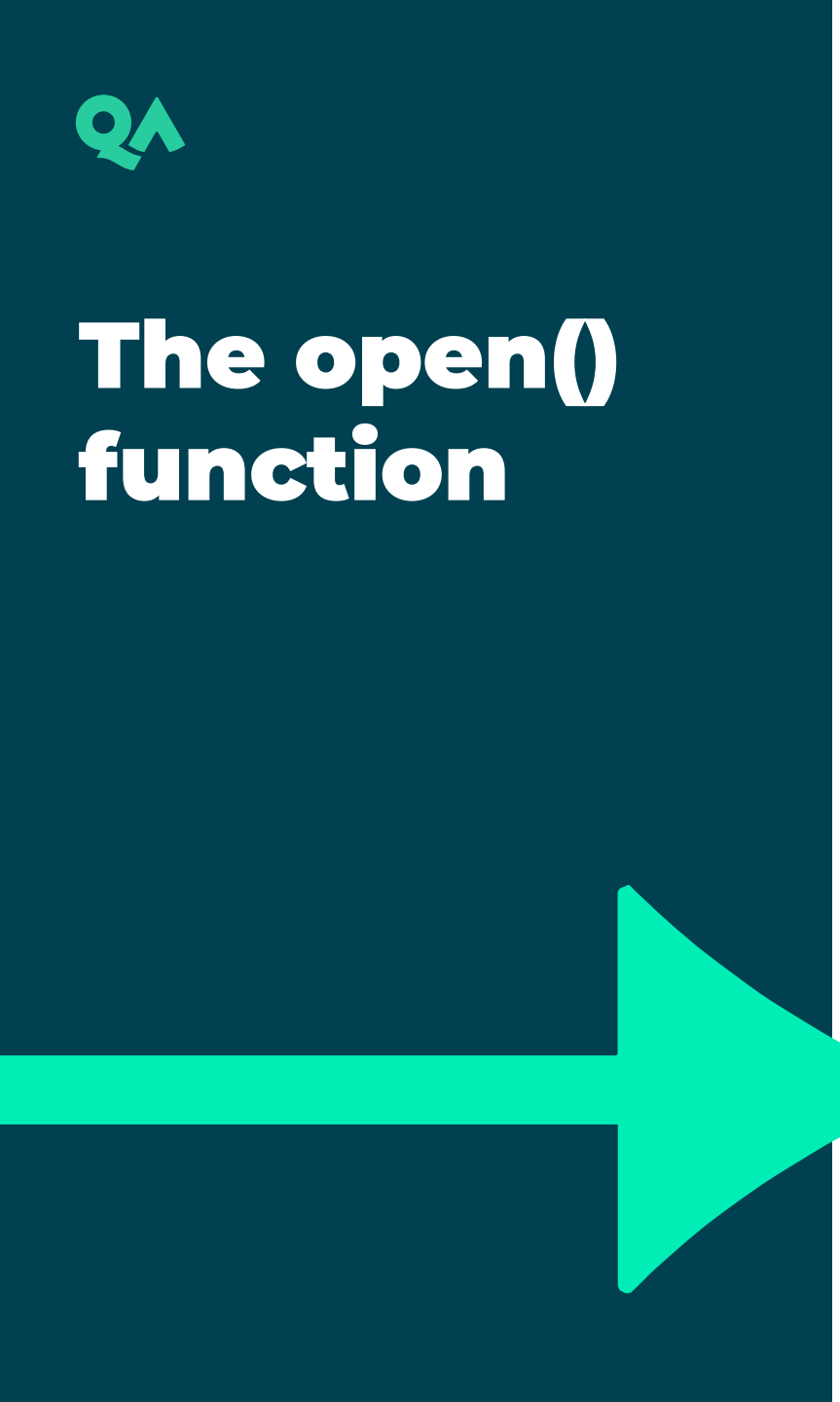
The open() function takes two parameters: file name and mode. Only the file name is mandatory.

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

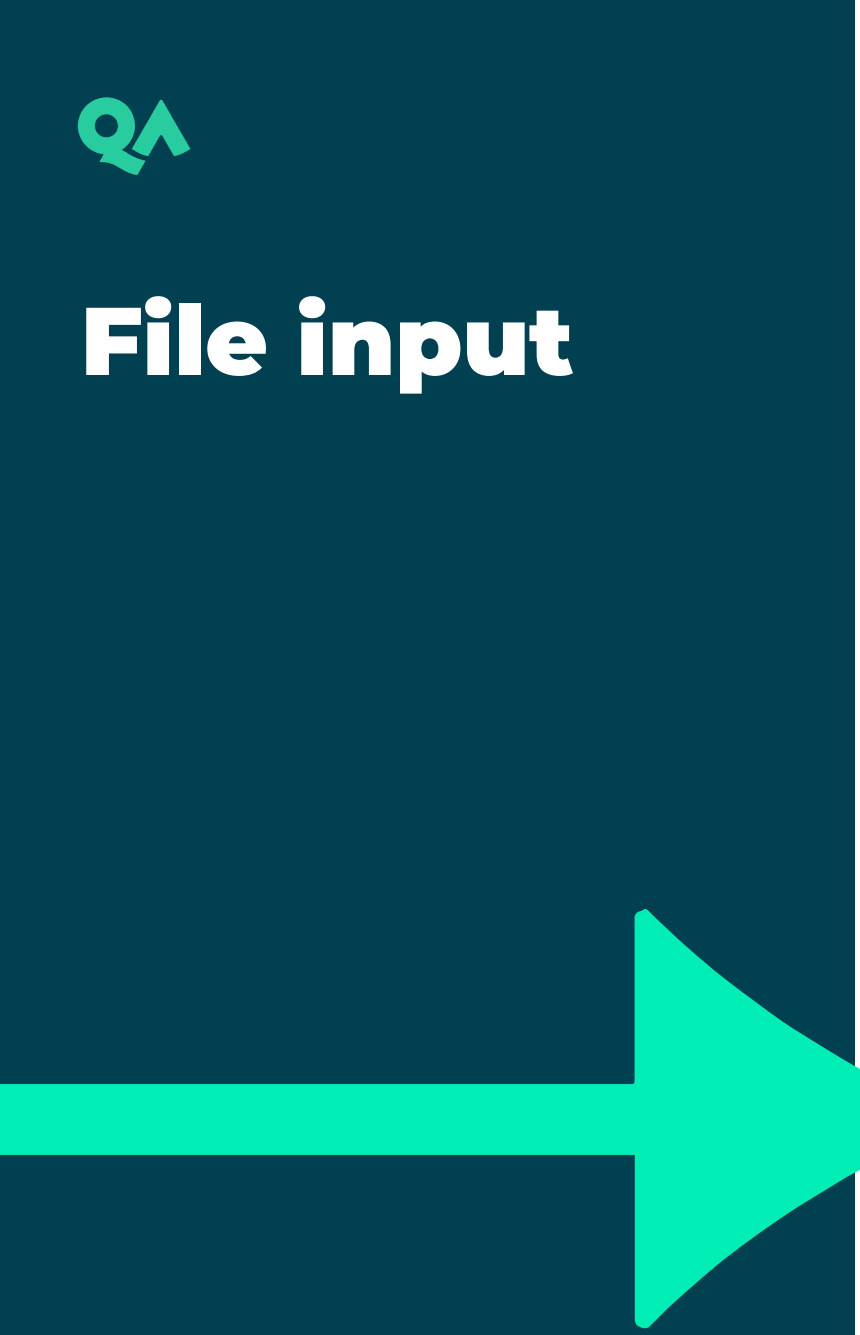
The open() function takes two parameters: file name and mode.

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.

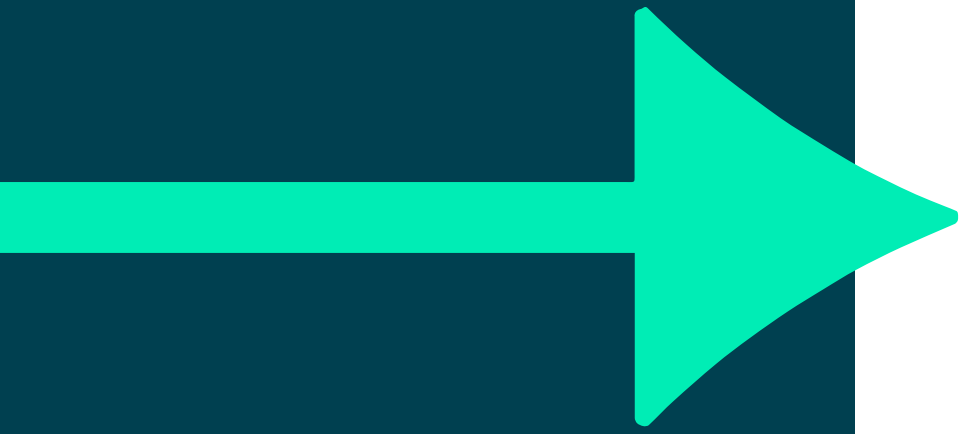
```
f = open('output.txt', 'w’)
f.write(‘Hello’)
f.close()
```

Closing a file

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

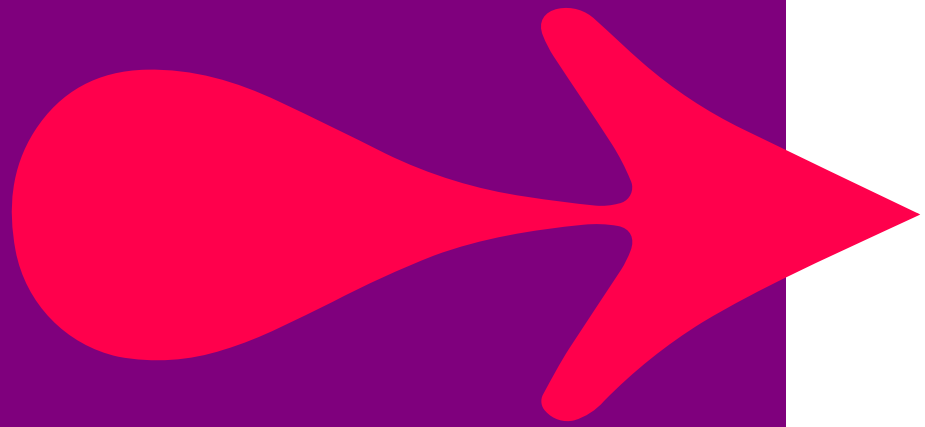
f.close()

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



Exercise

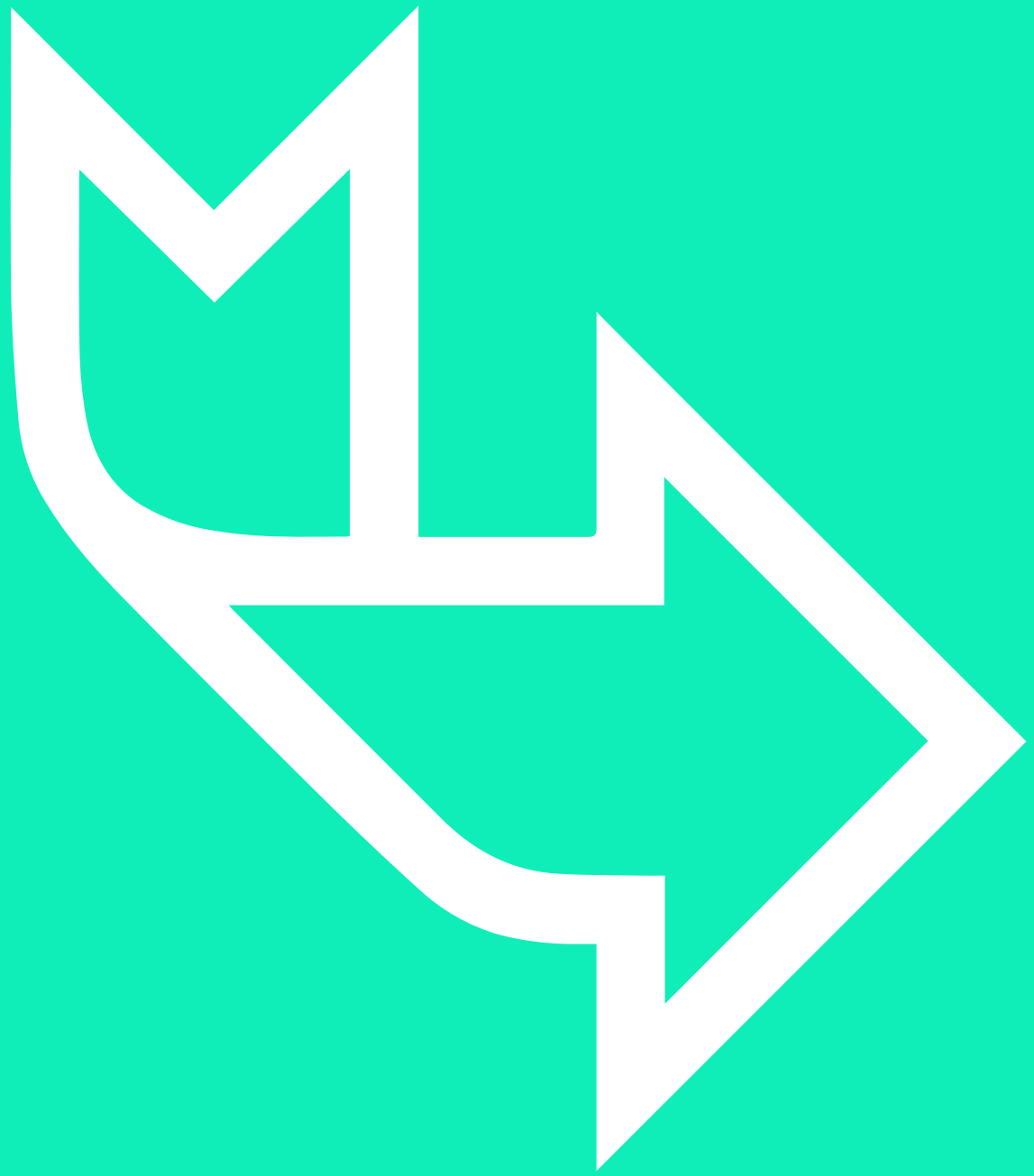
Go to **Exercise 3: Data structures, flow control, functions & basic types** in your exercise guide.



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?

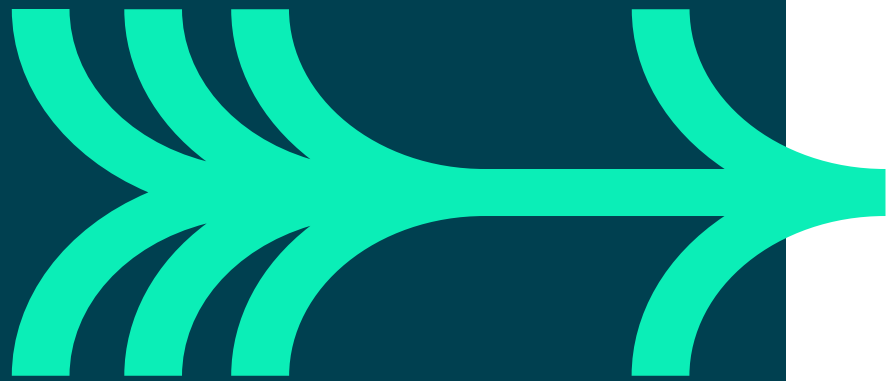
Learning objectives

- 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.

POWERING POTENTIAL

4. Mathematical and Statistical Programming with NumPy

Mathematical & statistical programming



Learning objectives

- Describe the core features of NumPy arrays.
- Create, index, and manipulate NumPy arrays to solve data problems.
- Use masking and querying syntax to retrieve desired values.
- Use vectorised ufuncs.

Expected prior knowledge

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

Activity: Tutorial

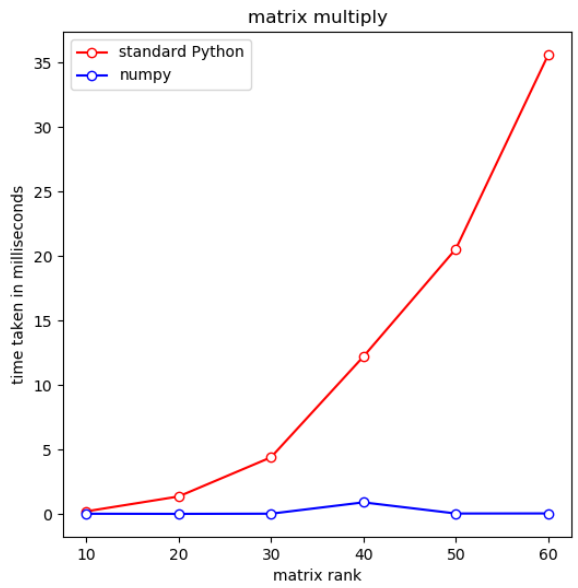
- Code along with the trainer for the following tutorial.
- Ask questions if you encounter errors.
- Think about the code you are writing.

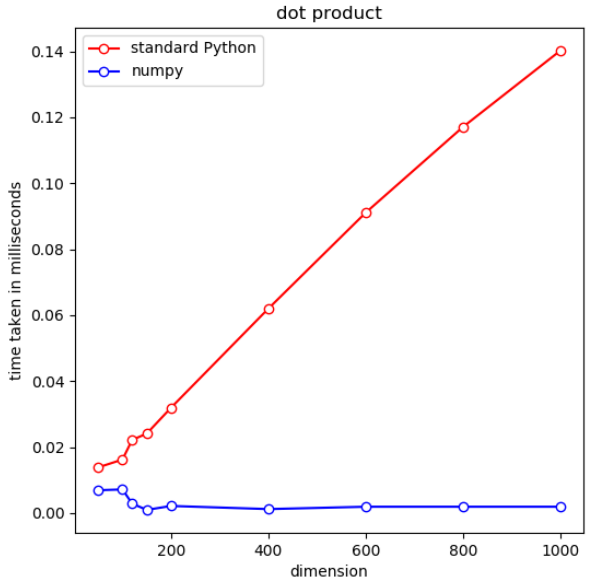
Why NumPy?

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.

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-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.

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]]
```

Array manipulation

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

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

payments[0:2]
```

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

```
payments[2:]
```

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

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

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

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()
```

```
29.702279794655492
```

Ufuncs

Universal functions

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

They are fast!

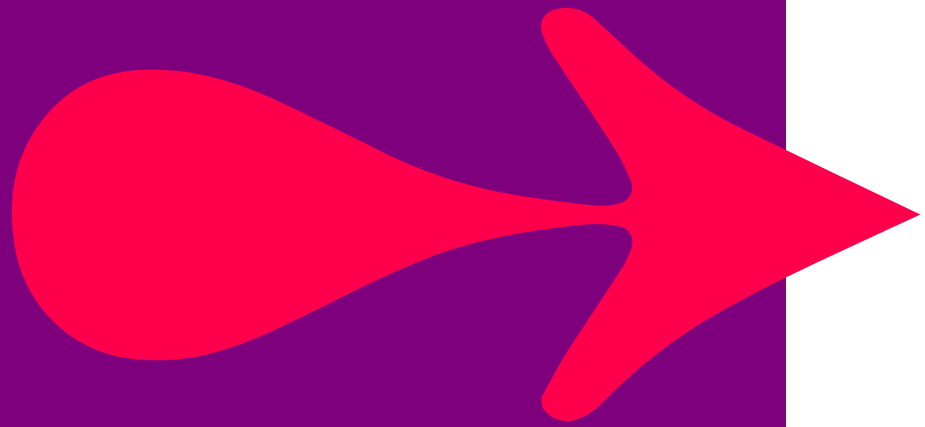
numpy.sqrt()
numpy.square()
numpy.exp()
numpy.log()
numpy.sign()
numpy.isnan()
numpy.sin()
numpy.add()

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

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

Exercise

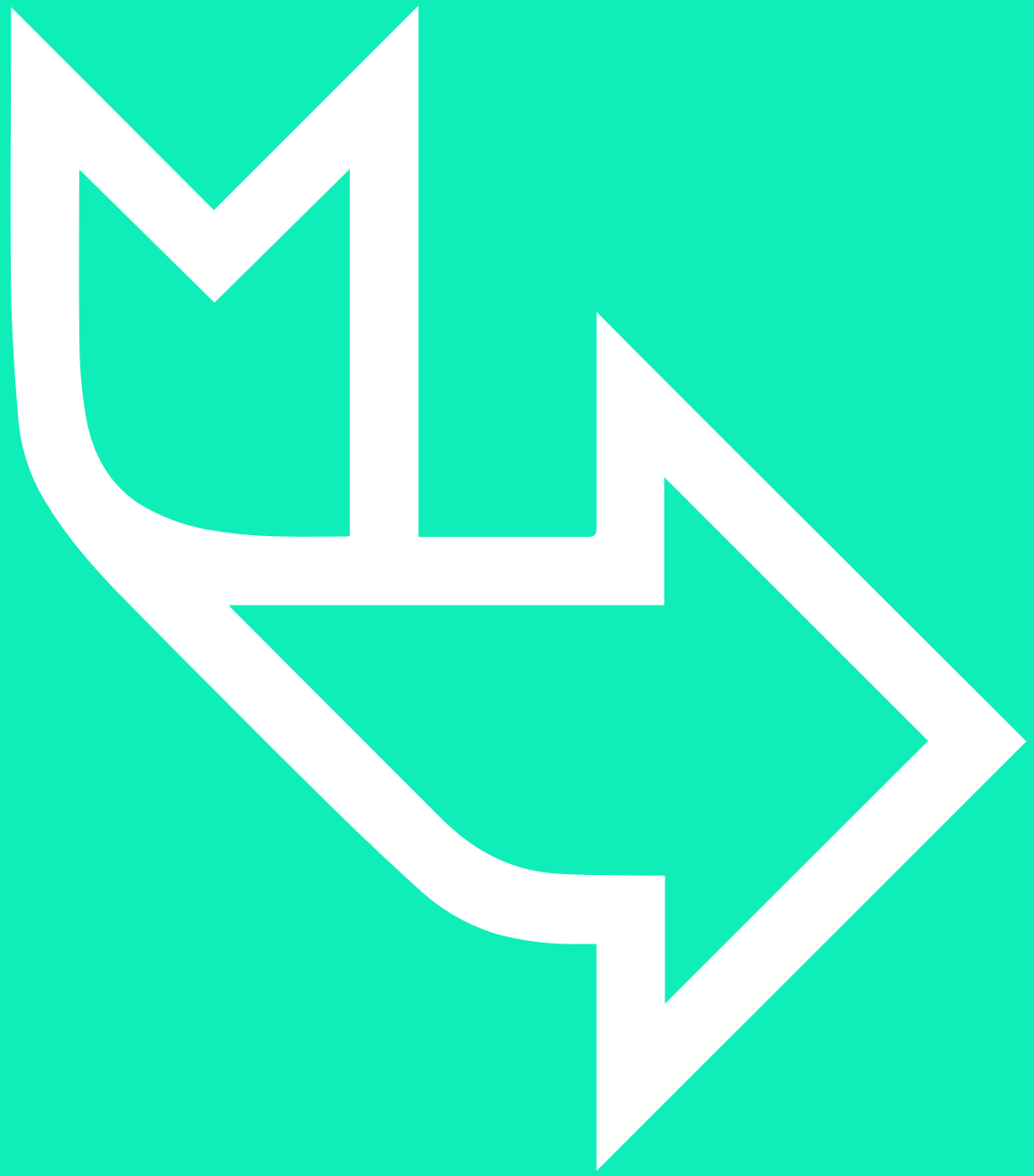
Go to **Exercise 4: Mathematical and statistical programming** in your exercise guide.



Learning check

Think about your answers to these questions:

- What are the core features of NumPy arrays?
- When manipulating NumPy arrays, what do we need to keep track of?
- How does NumPy allow us to query data stored in arrays?
- What are ufuncs? What are some examples of them?



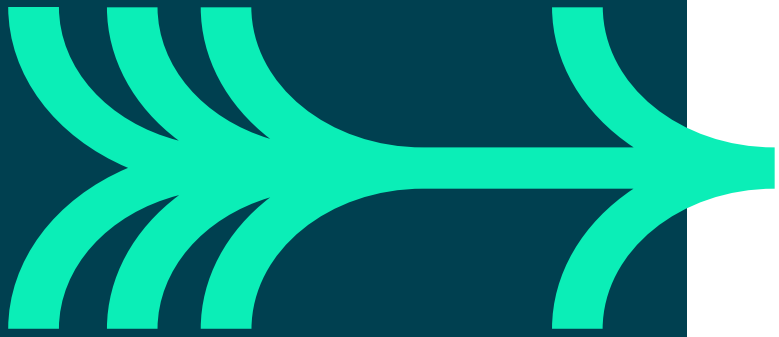
How did you get on?

Learning objectives

- Describe the core features of NumPy arrays.
- Create, index, and manipulate NumPy arrays to solve data problems.
- Use masking and querying syntax to retrieve desired values.
- Use vectorised ufuncs.

5. 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**

Series

Series

- A **Series** is a one-dimensional, single type array.
  - Represents a column in a table.
- It consists of two NumPy arrays:
  - **Index array**
  - **Values array**
- Each element has a unique index (ID).
  - Indices do not have to be sequential.
  - Like primary keys.

Creating Series

A **Pandas Series** can be created from a Python list or a NumPy array:

```
import pandas as pd

X = [1, 3, 5, 7]
mySeries = pd.Series(X)
print(mySeries)
```

```
0    1
1    3
2    5
3    7
dtype: int64
```

Index array

values array

- The index starts from 0 and increases by 1 for each subsequent element in the Series.
- The index is used to access the corresponding value.

```
print(mySeries[1])
```

```
3
```

Querying Series

Series can be used like an array, except the indices must correspond to the elements in the index array.

series.index returns the index array.

series.values returns the values array.

series[ind] is equivalent to **series.loc[ind]**, returns the element in the series with ID equal to ind.

series.iloc[i] returns the i-th element in the series.

Querying Series

```
import pandas as pd

X = [1, 3, 5, 7]
mySeries = pd.Series(X,index=[9, 8, 7, 6])
print(mySeries)
```

```
9    1
8    3
7    5
6    7
dtype: int64
```

```
mySeries.index
```

```
Int64Index([9, 8, 7, 6], dtype='int64')
```

```
mySeries.values
```

```
array([1, 3, 5, 7], dtype=int64)
```

```
mySeries.loc[7]
```

← Index 7

```
5
```

```
mySeries.iloc[0]
```

← Position 0

```
1
```

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.

```
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

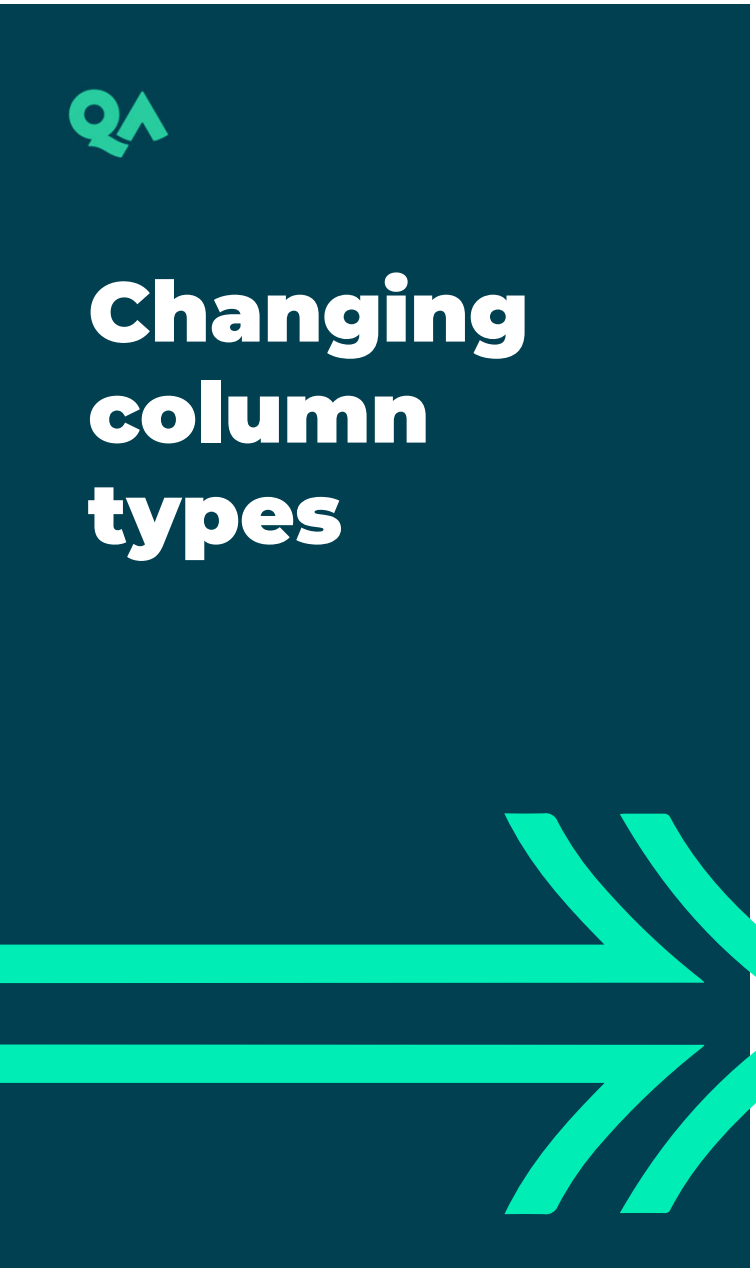
Querying SQL Tables

- 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:

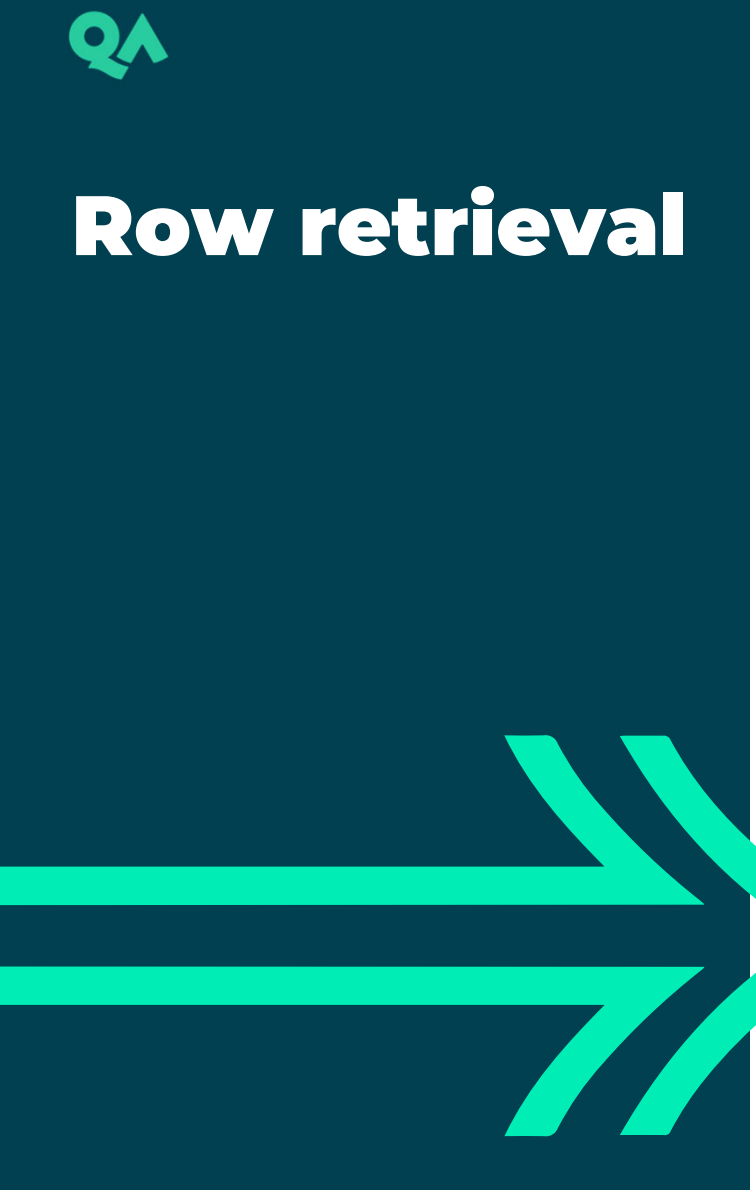
my_dataframe[column_name]

```
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
```

time	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

```
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"]
```

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

← Column “temp”

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

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

← Column with position 0

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
A	Charlie	Male	25

Group	Name	Gender	Age
B	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()
```

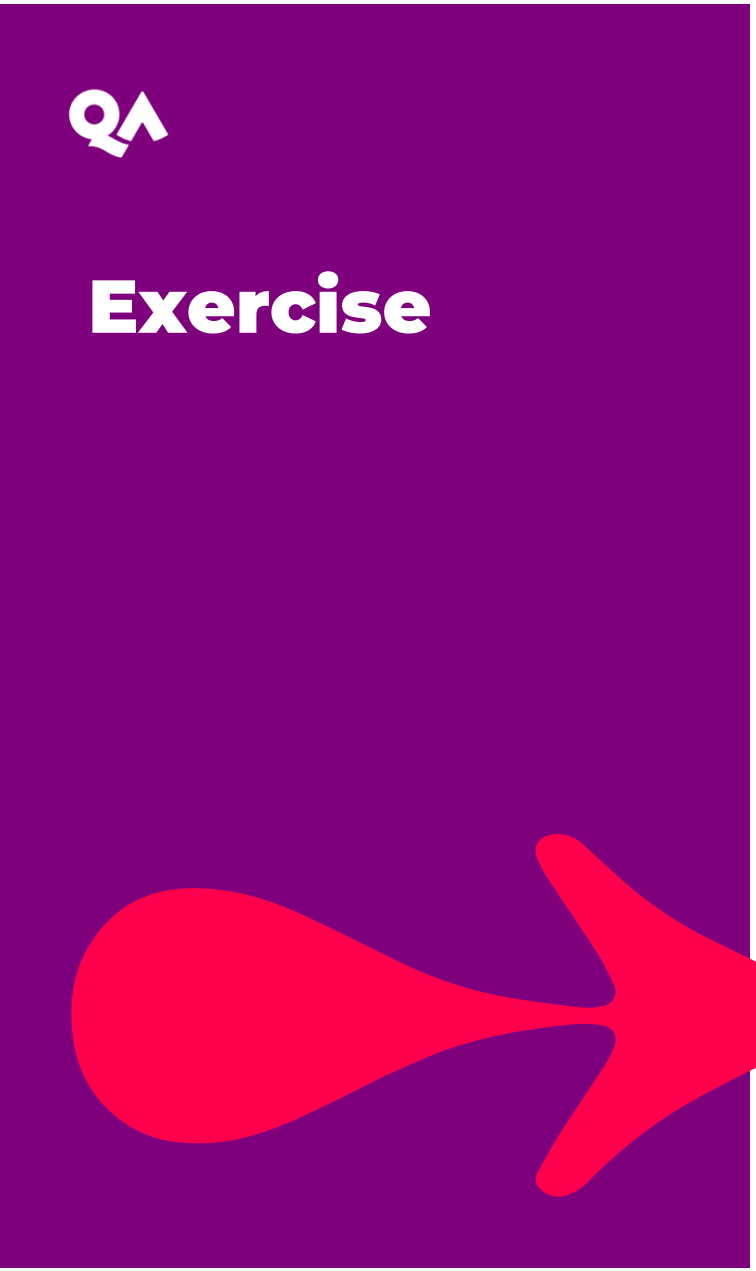
Term	Balance
Long Term	362870
Short Term	676600

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



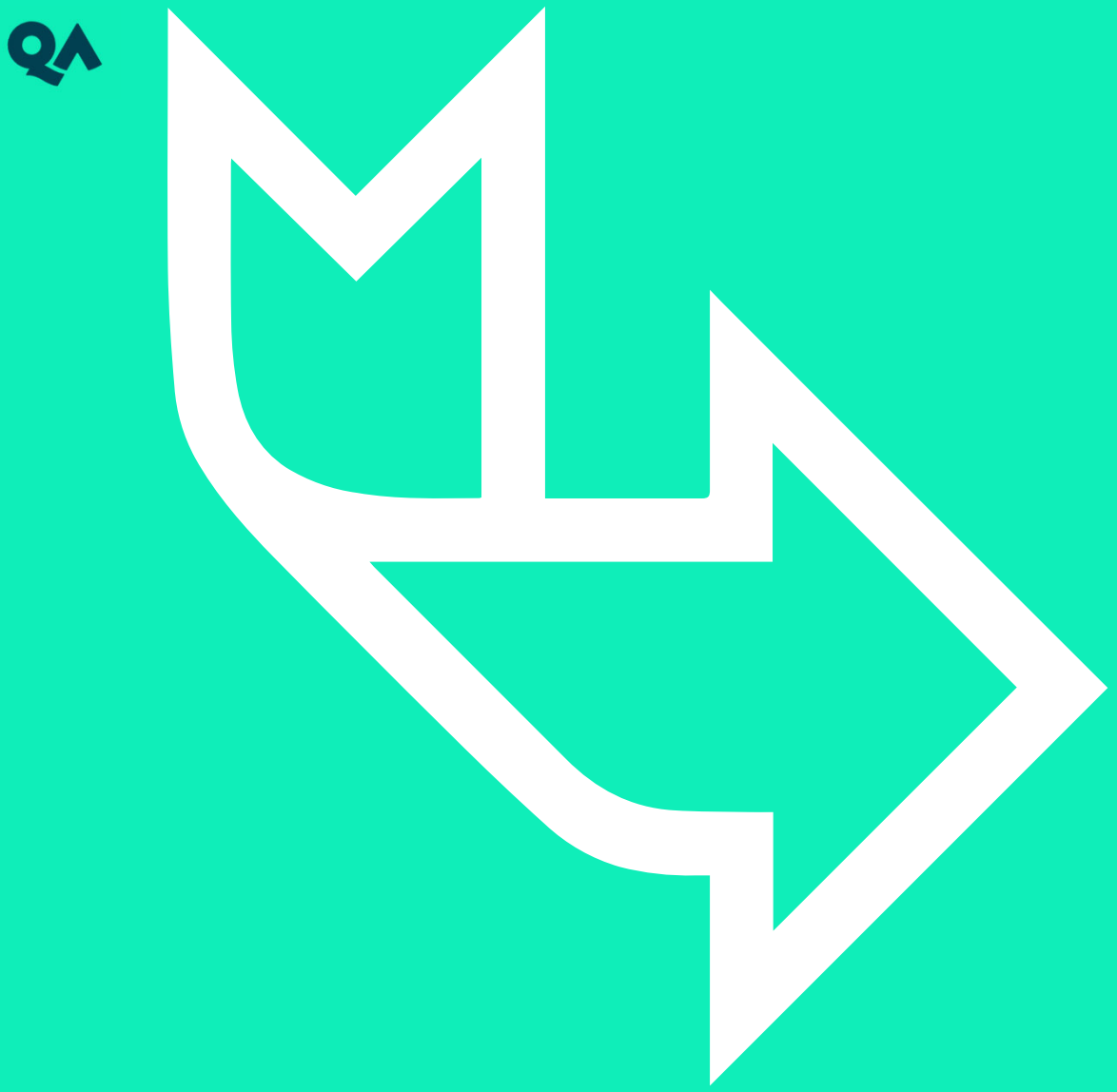
Exercise

Go to **Exercise 5: Introduction to Pandas** in your exercise guide.

Learning check

Think about your answers to these questions:

- What is the difference between Series and DataFrames?
- Are DataFrame indices fixed?
- Which functions can we use to change column types? What types does Pandas use? Do we use the same functions for time data?
- Which types of file can Pandas read & write from?

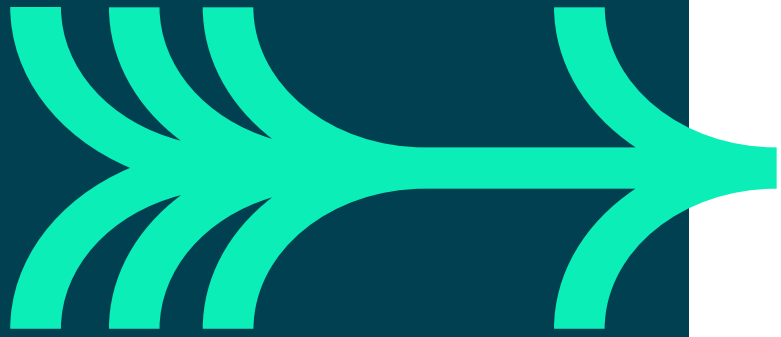


HOW DID YOU GET ON?

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.

Data cleaning with Pandas



Learning objectives

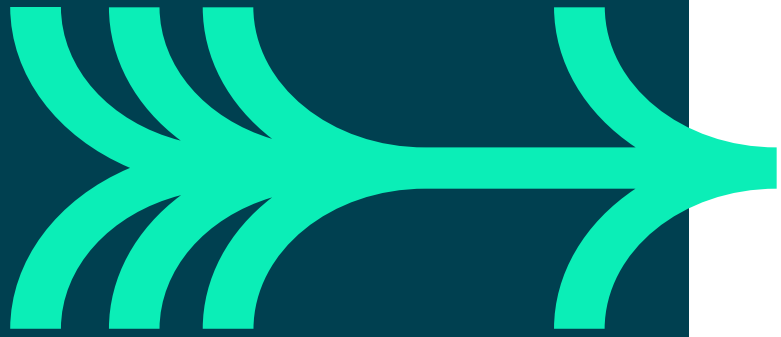
- Identify missing data and apply techniques to deal with it.
- Deduplicate, transform, and replace values.
- Use DataFrame string methods to manipulate text data.
- Write regular expressions which munge text data.

Expected prior knowledge

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

Missing values

Data quality dimensions



- Accuracy
- Completeness
- Uniqueness
- Timeliness
- Validity
- Consistency

Missing values

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

Duplicates

```
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
...	...	...	...	...	...	...	...

```
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
```

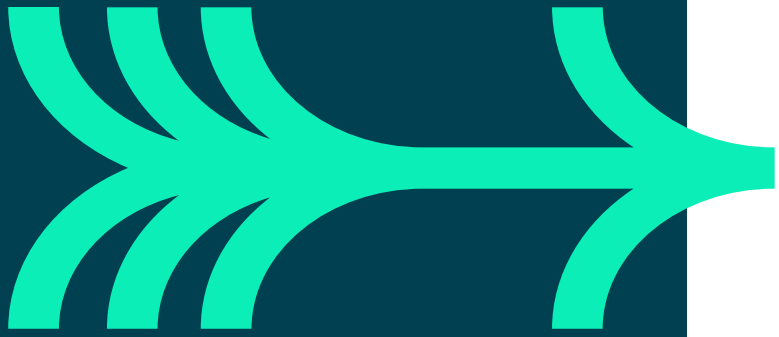
```
loans_dup.duplicated()
```

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

```
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

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)
```

	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
...	...	...	...	...	...	...	...

← Row 0

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

```
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

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

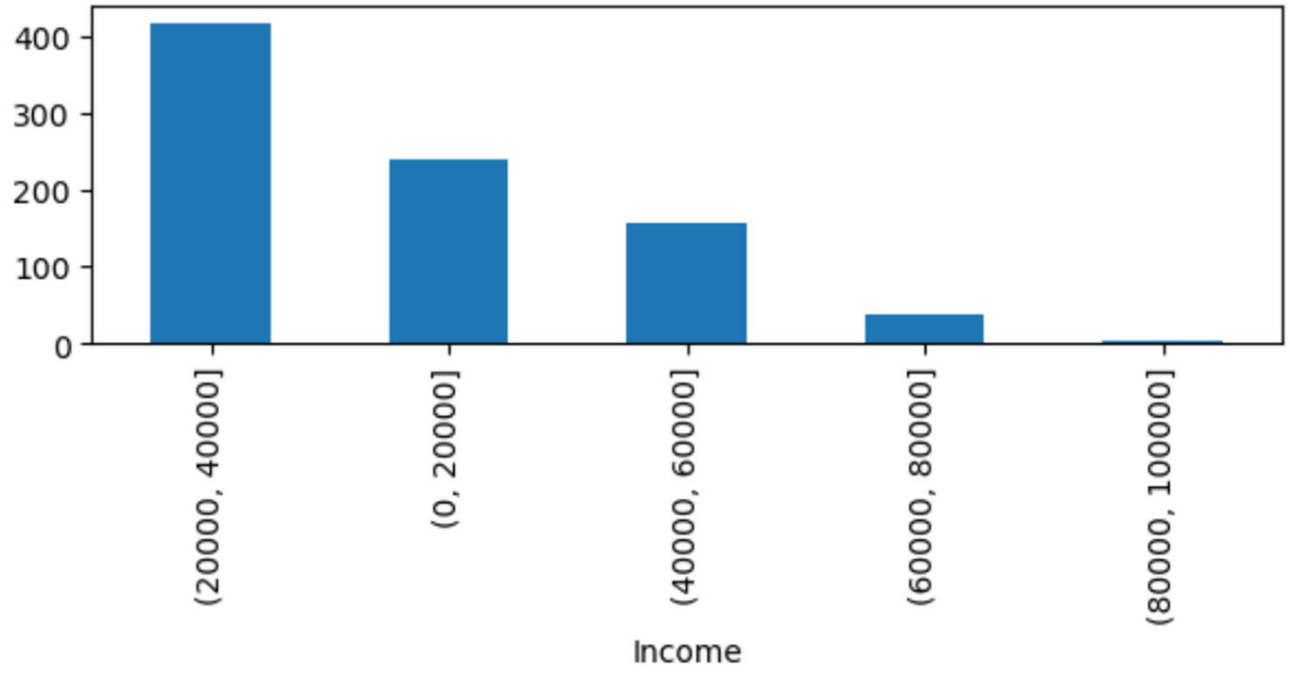
	ID	Income	Term	Balance	Debt	Score	Default
0	567	17500	6 Month Term	1460	272	225.0	False
1	523	18500	12 Month Term	890	970	187.0	False
2	544	20700	6 Month Term	880	884	85.0	False
3	370	21600	6 Month Term	920	0	NaN	False
4	756	24300	6 Month Term	1260	0	495.0	False

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, 100000]]
```

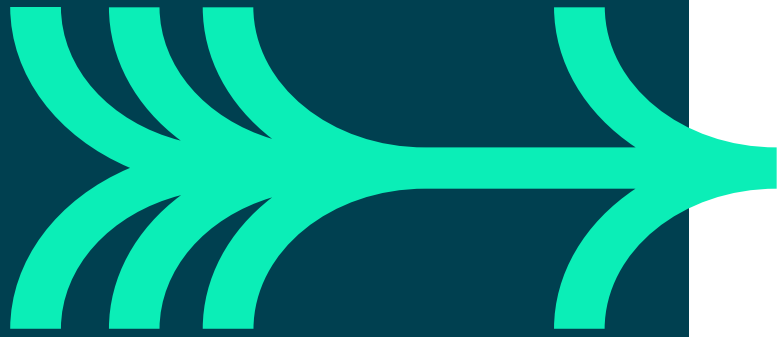
```
pd.cut(x=df['Income'],
       bins=[0, 20_000, 40_000, 60_000, 80_000, 100_000]
      ).value_counts().plot(kind='bar', figsize=(7,2));
```



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



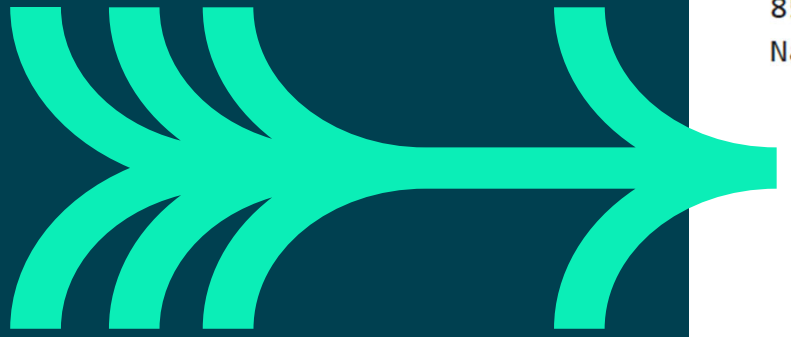
Applying functions over columns

```
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()
```

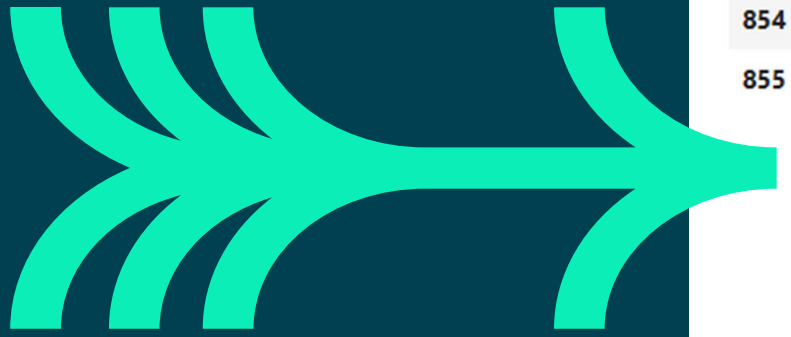
```
851    High
852     Low
853    High
854    High
855    High
Name: Debt, dtype: object
```



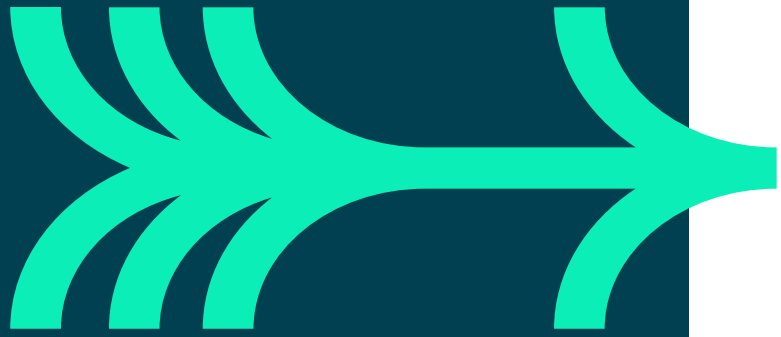
Applying functions over columns

```
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

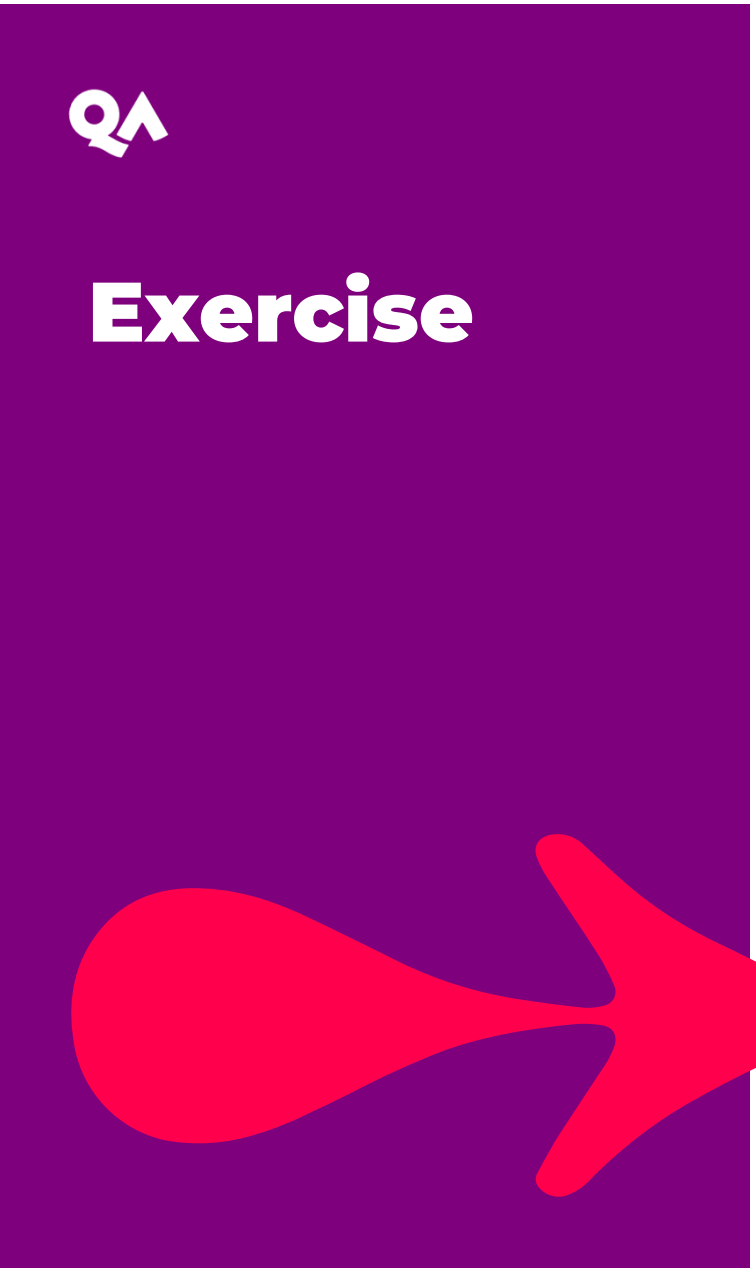
Writing DataFrames to Files

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

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

```
df.to_
```

- f to_clipboard function
- f to_csv function
- f to_dict function
- f to_excel function
- f to_feather function
- f to_gbq function
- f to_hdf function
- f to_html function
- f to_json function
- f to_latex function



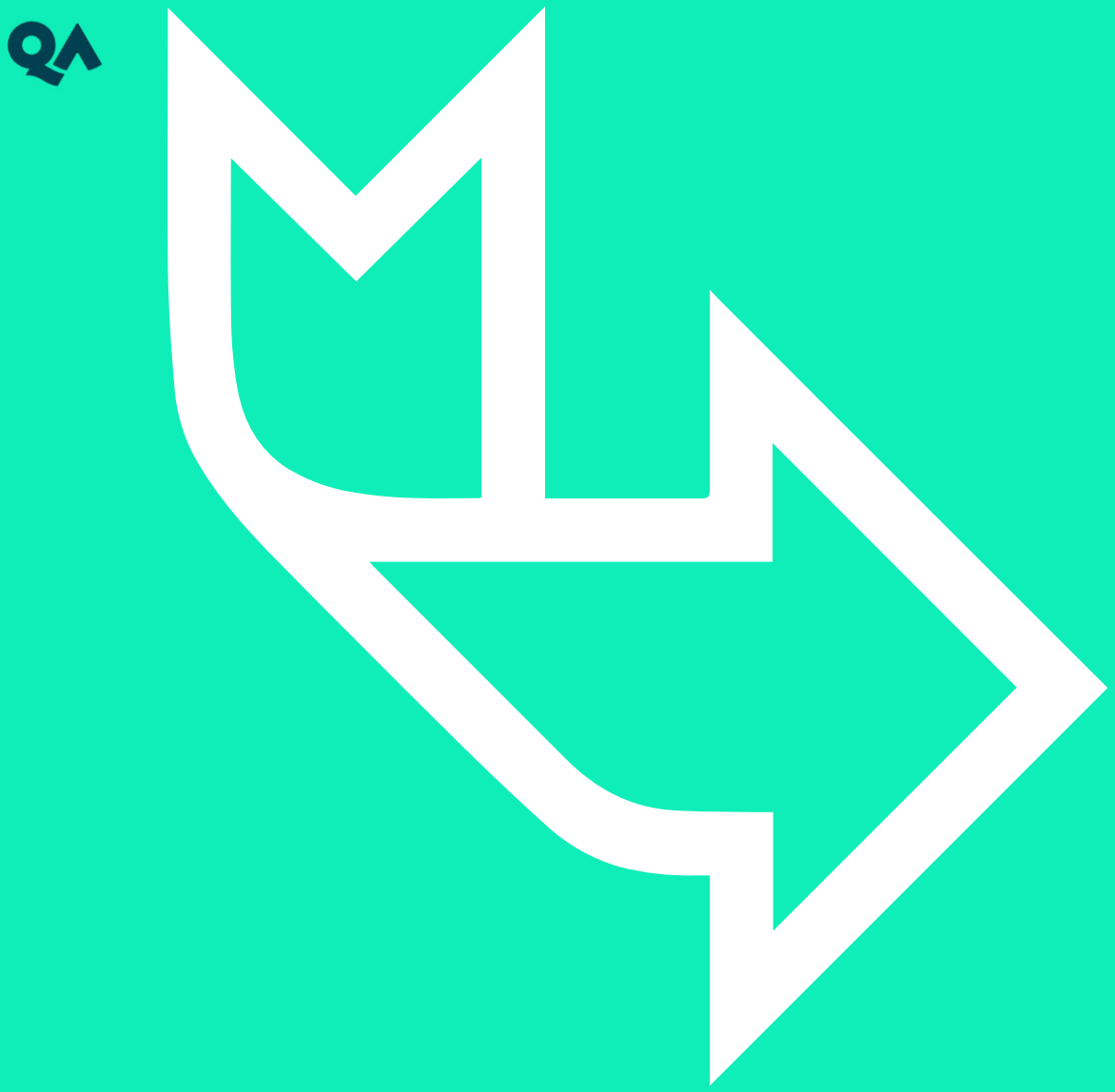
Exercise

Go to **Exercise 6: 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?

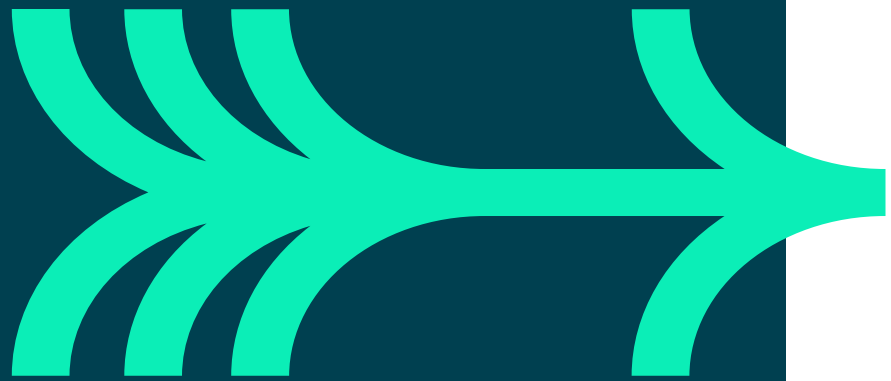
Learning objectives

- 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

7. 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.

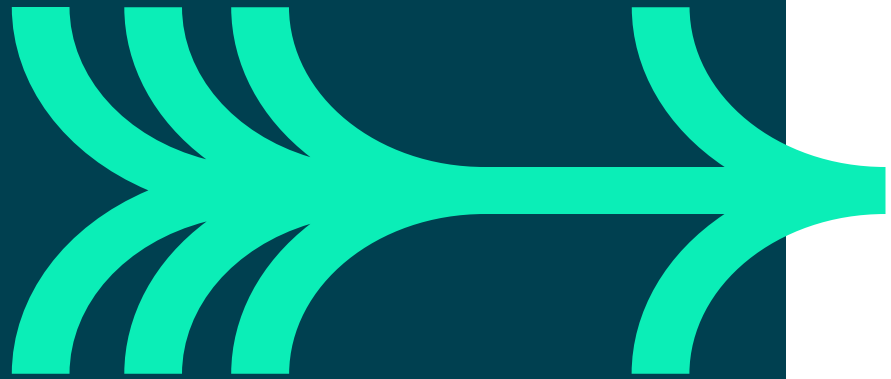
```
df.pivot_table(values=['Income'],
               index='Default',
               columns='Term').round()
```

	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', :]
```

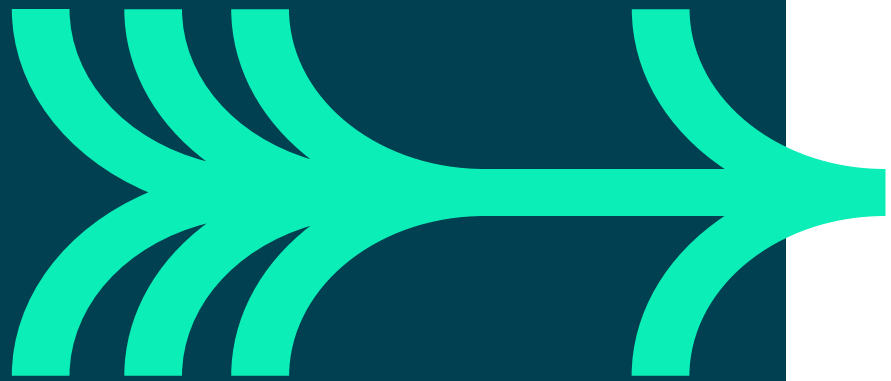
```
temp          32.18
humidity      71.25
sun_hrs       10.34
Name: 2023-07-15 00:00:00, dtype: float64
```

```
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.

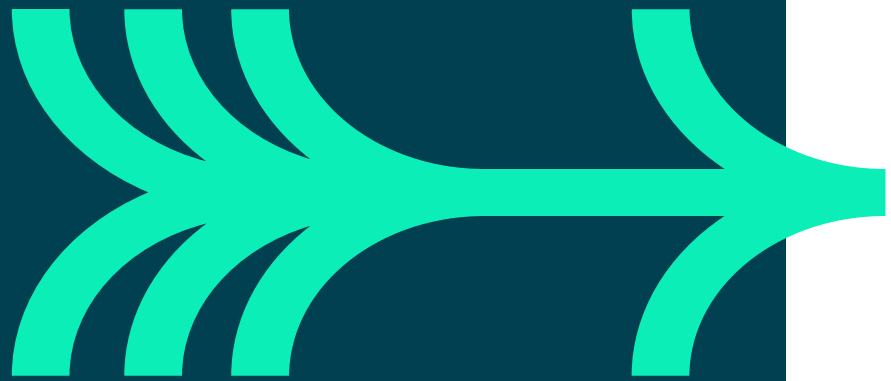
```
pd.date_range(start='2020',
              end='2024',
              freq='Q')
```

```
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')
```

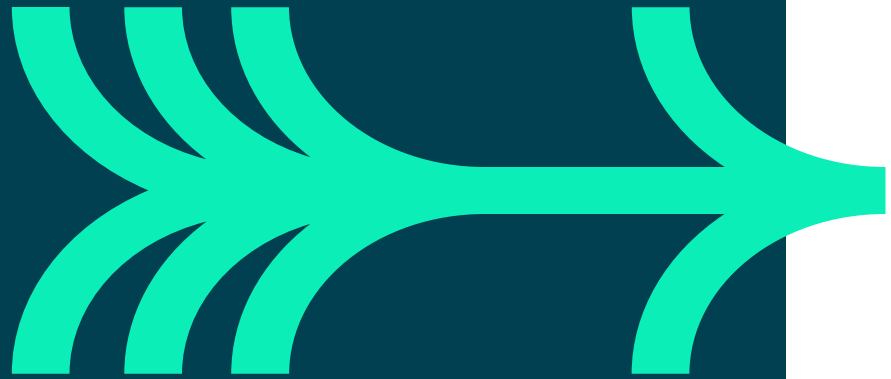
```
DatetimeIndex(['2020-03-31 02:00:00+02:00', '2020-06-30 02:00:00+02:00',
               '2020-09-30 02:00:00+02:00', '2020-12-31 01:00:00+01:00',
               '2021-03-31 02:00:00+02:00', '2021-06-30 02:00:00+02:00',
               '2021-09-30 02:00:00+02:00', '2021-12-31 01:00:00+01:00',
               '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

Streaming data

Streaming large files

- When a file is too large ($\approx$ RAM), it won't process all at once.
- Read functions allow chunks to be read in.
  - File processed N rows at a time.
  - Need to work iteratively.

```
for i, chunk in enumerate(pd.read_csv('data/loan_data.csv', chunksize=100)):
    if i == 0:
        null_count = chunk.isna().sum()
    else:
        null_count += chunk.isna().sum()
null_count
```

```
ID          0
Income      0
Term        0
Balance     0
Debt        0
Score      20
Default     0
dtype: int64
```

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]
```

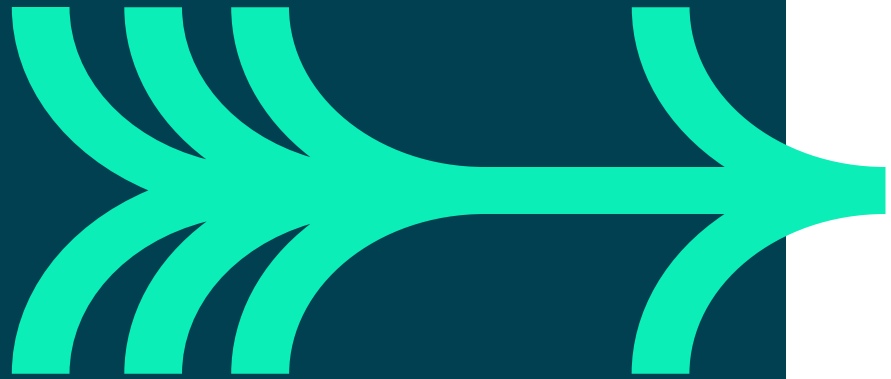
	ID	Income	Term	Balance	Debt	Score	Default
0	567	17500.0	Short Term	1460.0	272.0	225.0	False
1	523	18500.0	Long Term	890.0	970.0	187.0	False

```
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

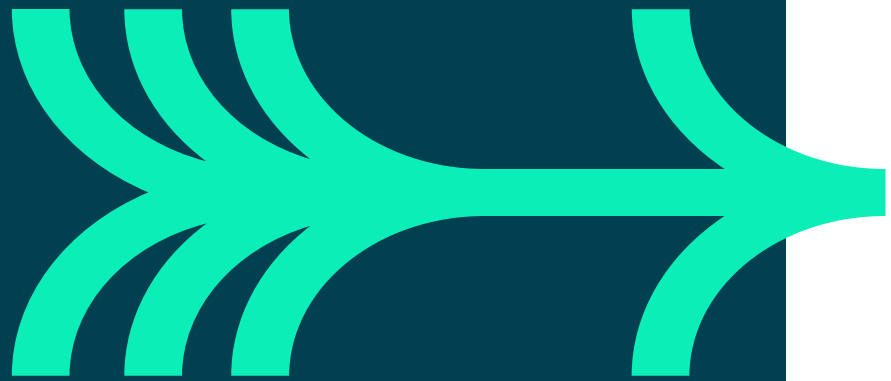
```
df_next.head()
```

	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
```

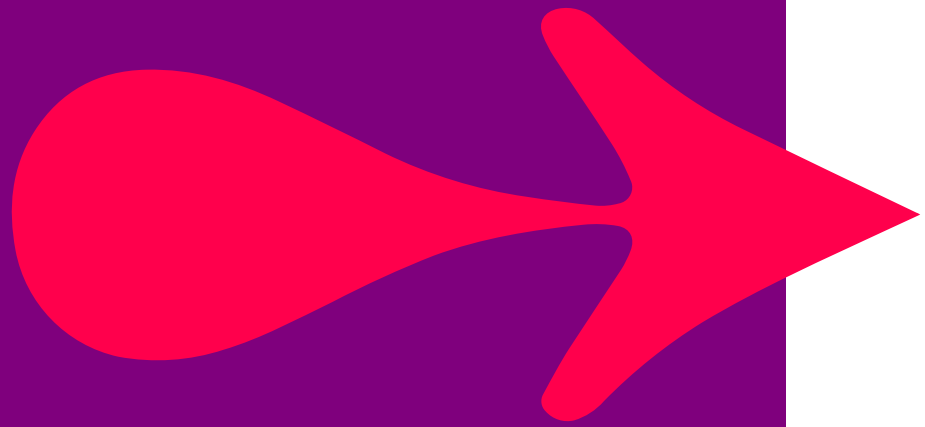
	temp	humidity	sun_hrs
2023-08-14	28.4	77.86	9.80
2023-08-15	28.0	76.43	9.13

```
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

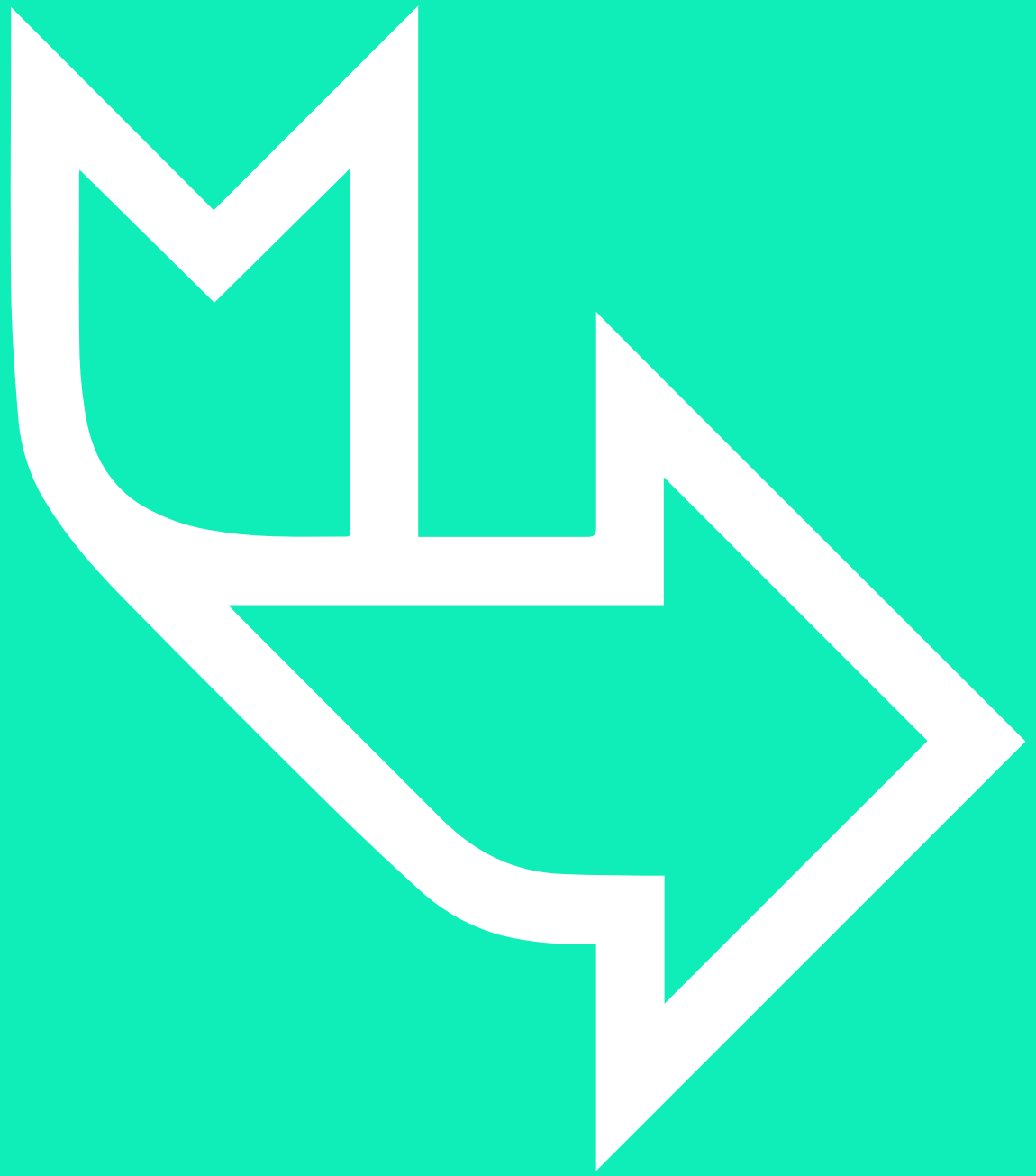
Go to **Exercise 7: Data manipulation with Pandas** in your exercise guide.



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?

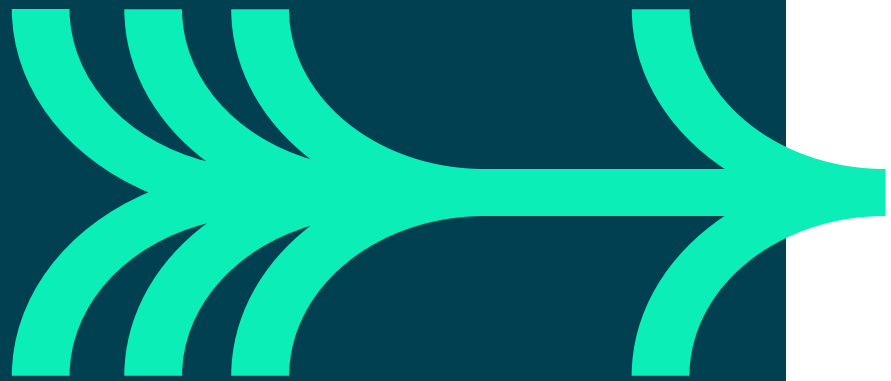
Learning objectives

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

POWERING POTENTIAL

8. Methods for Visualising Data

Methods for visualising data



Learning objectives

- Construct and tailor basic data visualisations using Matplotlib and Seaborn for both numeric and non-numeric data.
- Meaningfully visualise aggregate data using Matplotlib and Seaborn.

Expected prior knowledge

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

Matplotlib basics

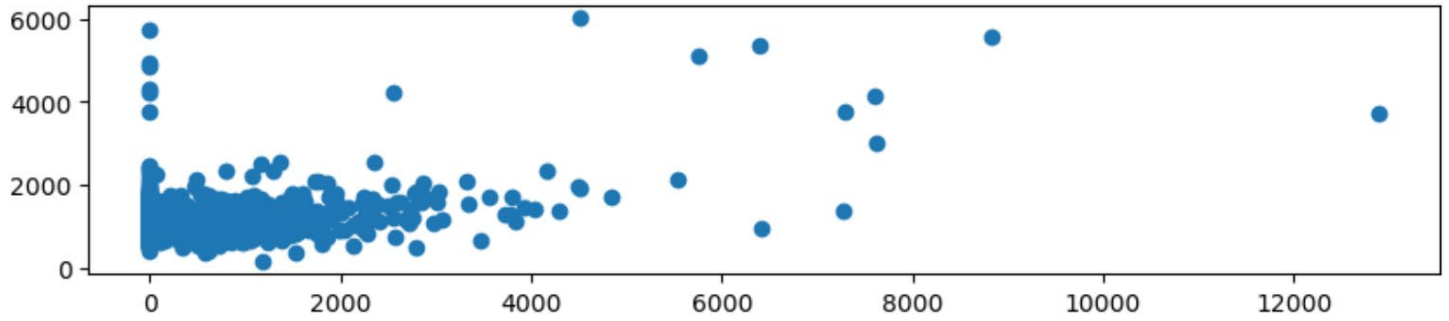
MATLAB style plotting

- Matplotlib uses MATLAB style plotting.
  - A sequence of functions ‘build’ a graph.
- The base of a graph is a figure.
  - Canvas onto which plots go.
  - Defines the size, number of plots etc.
- Added to figures are axes.
  - One or many.
- Onto axes are plotted artists (i.e., plots).
  - Can be any style of plot, e.g. scatter, bar, pie, etc.
- Alterations to the graph can be made using OO style or MATLAB style.
  - OO uses setters and getters.
  - MATLAB uses a series of functions.

Figures & axes, and artists

- Plt.subplots creates a Figure object and an Axes object, which is part of the Figure.
- Plt.scatter plots data on the Axes object.

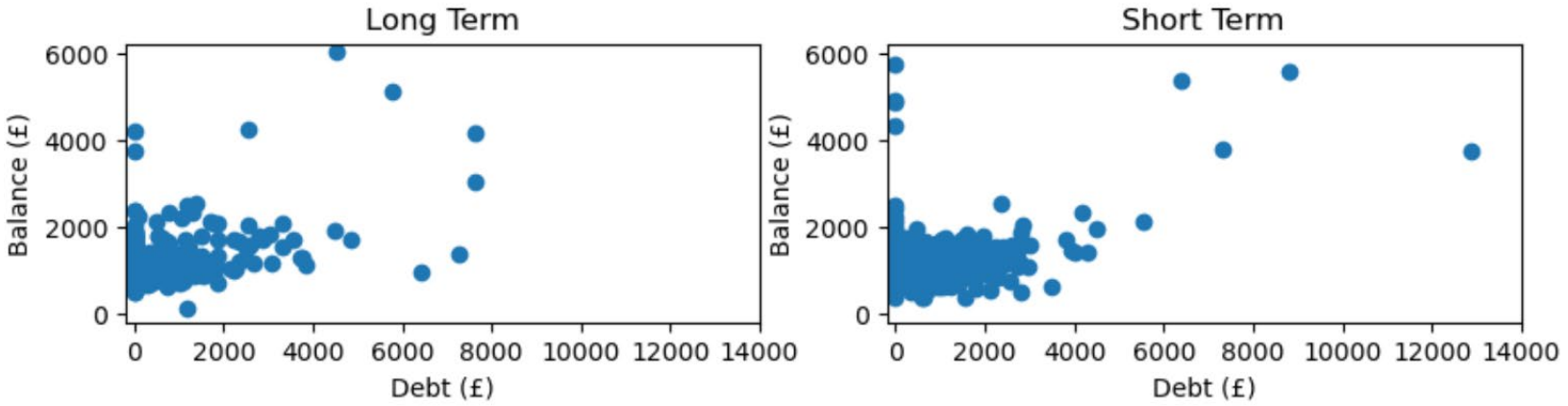
```
fig, ax = plt.subplots(figsize=(10, 2))

plt.scatter(df['Debt'],
            df['Balance']);
```



Plotting data on axes

Editing here is being done using the OO style.

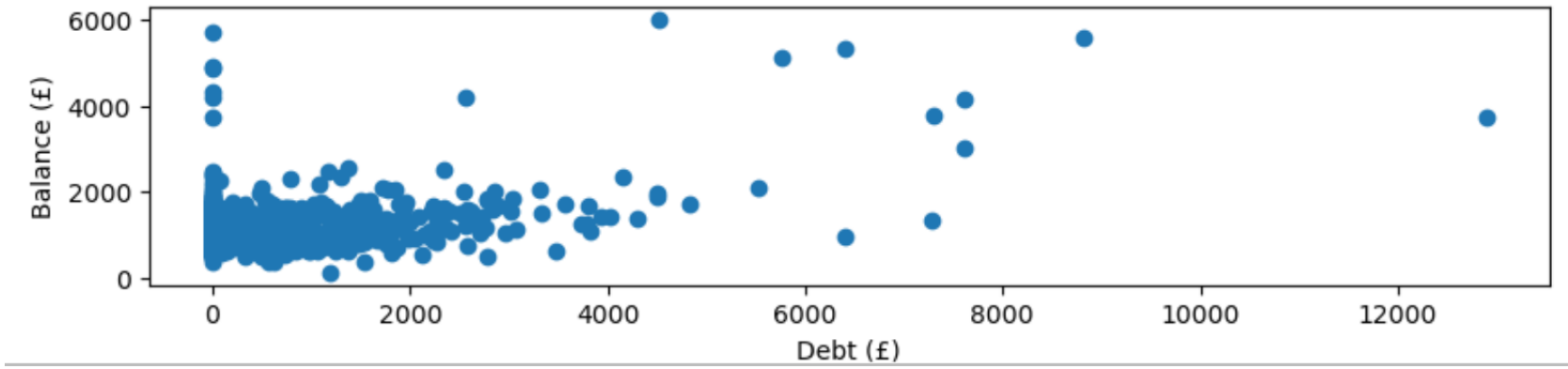
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 2))

ax1.scatter(df[df['Term'] == 'Long Term']['Debt'],
            df[df['Term'] == 'Long Term']['Balance'])
ax1.set_xlabel('Debt (£)')
ax1.set_ylabel('Balance (£)')
ax1.set_title('Long Term')
ax1.set_xlim(-200, 14_000)
ax1.set_ylim(-200, 6_200)

ax2.scatter(df[df['Term'] == 'Short Term']['Debt'],
            df[df['Term'] == 'Short Term']['Balance'])
ax2.set_xlabel('Debt (£)')
ax2.set_ylabel('Balance (£)')
ax2.set_title('Short Term')
ax2.set_xlim(-200, 14_000)
ax2.set_ylim(-200, 6_200);
```



Tailoring plots

Editing here is done using the MATLAB style.

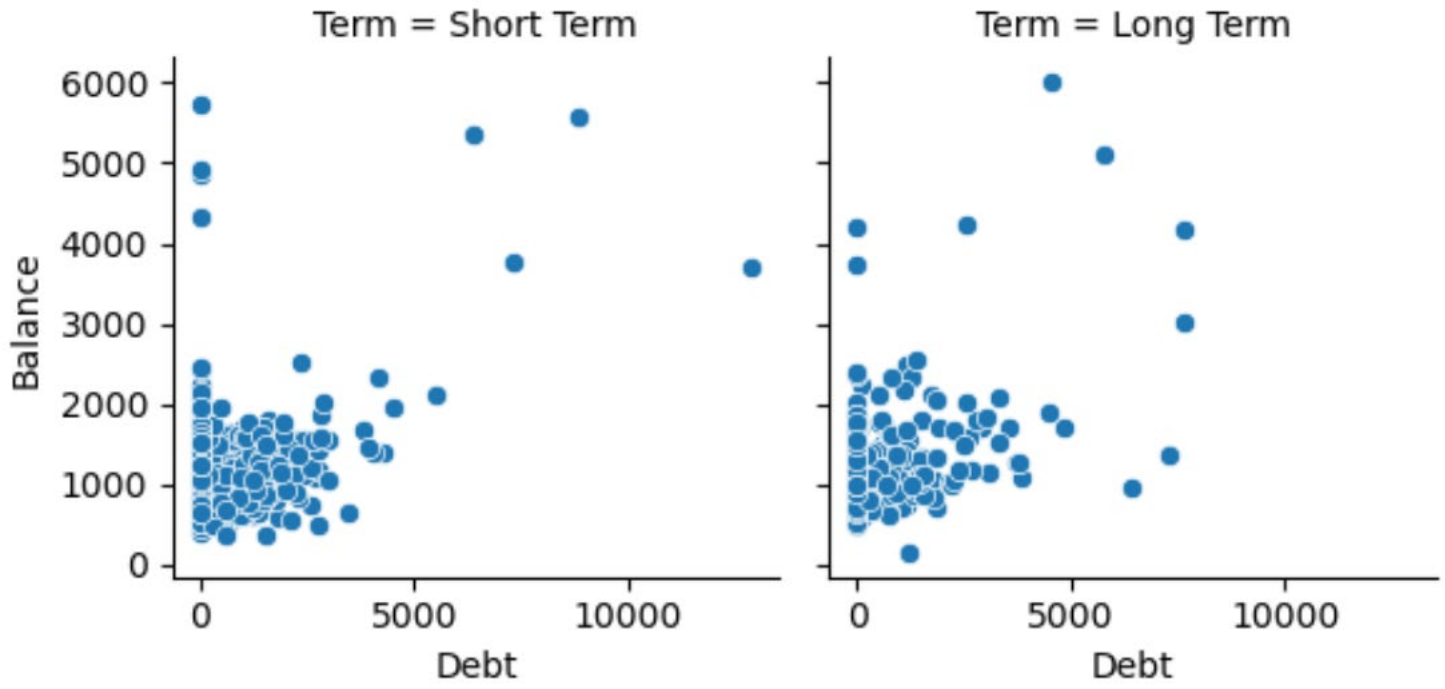
```
fig, ax = plt.subplots(figsize=(10, 2))

plt.scatter(df['Debt'],
            df['Balance'])

plt.xlabel('Debt (£)')
plt.ylabel('Balance (£)');
```



Seaborn

- Seaborn makes plotting easier.
  - It is still Matplotlib under the hood.

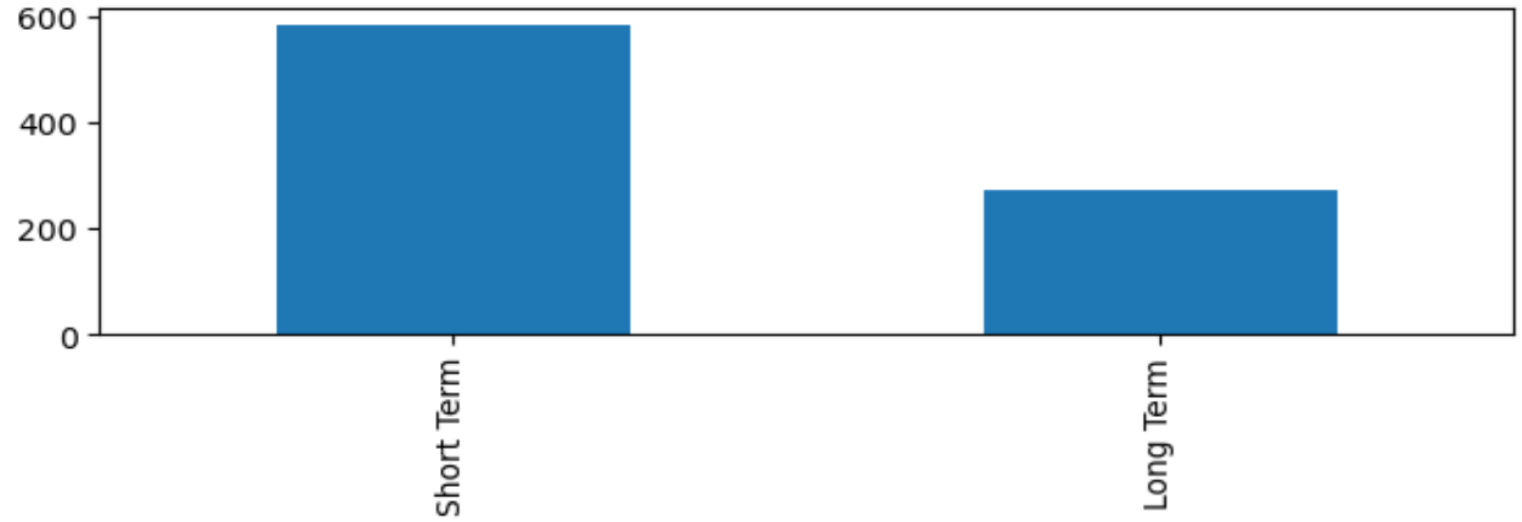
```
sns.FacetGrid(data=df,
              col='Term').map(sns.scatterplot, 'Debt', 'Balance');
```



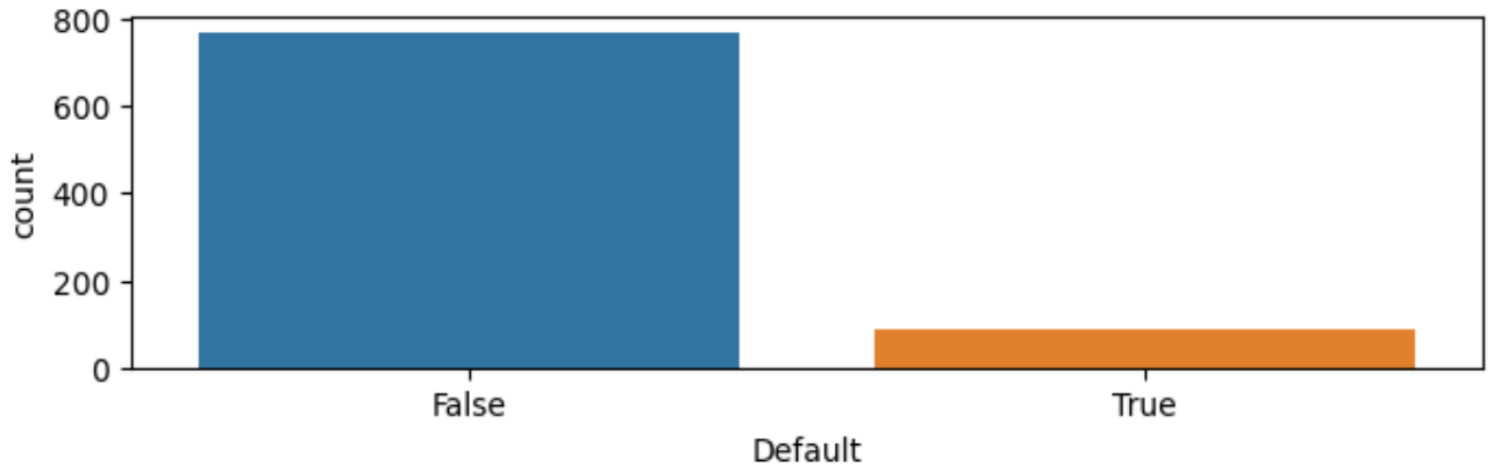
Univariate plots

Bar charts

```
fig, ax = plt.subplots(figsize=(8, 2))

df['Term'].value_counts().plot(kind='bar', ax=ax);
```

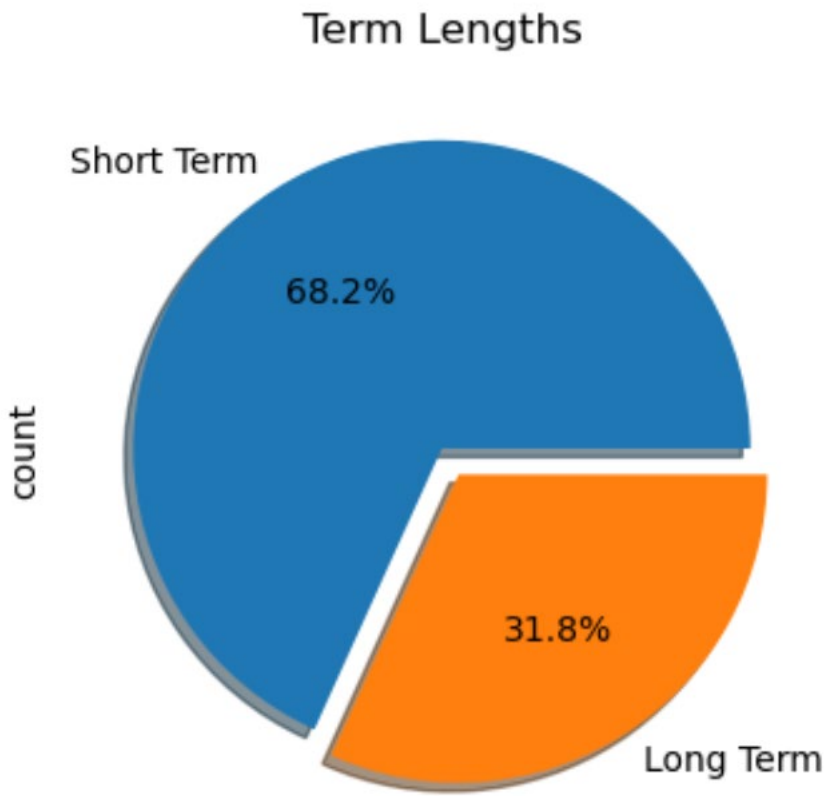



```
fig, ax = plt.subplots(figsize=(8, 2))

sns.countplot(data=df,
              x='Default',
              ax=ax);
```



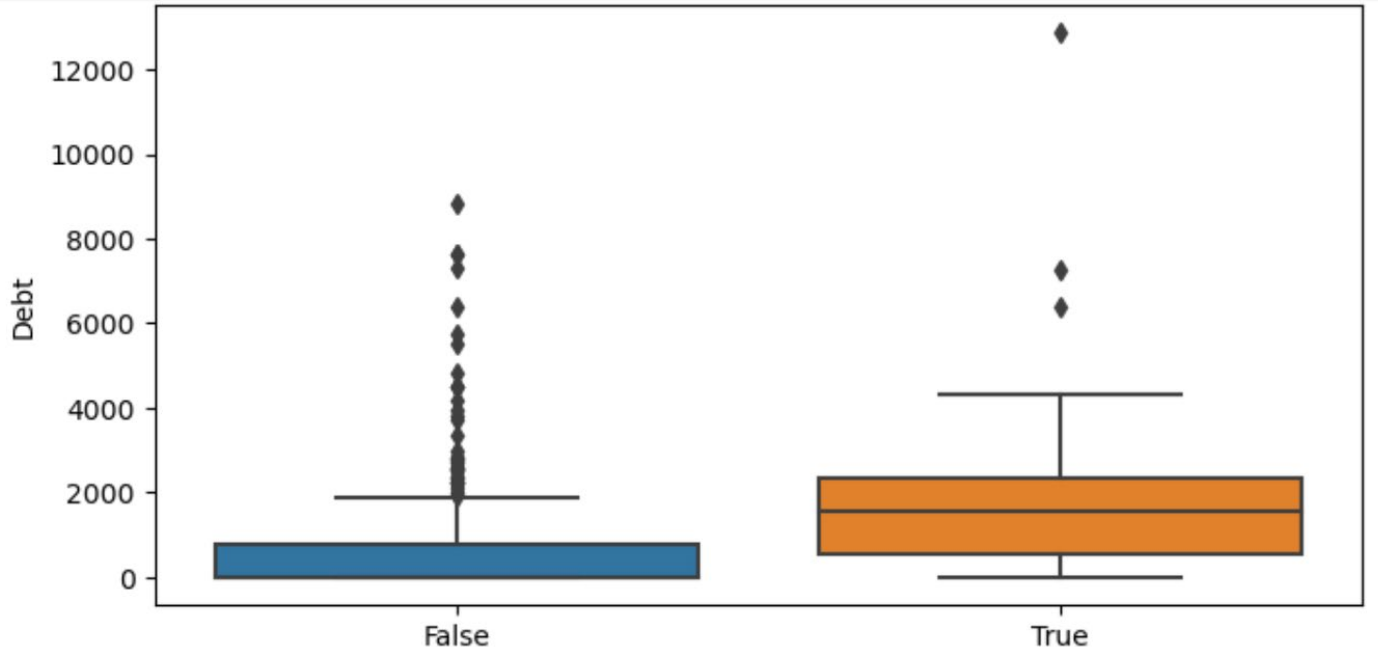
Pie charts

```
df.value_counts('Term').plot.pie(y='Term',
                                 title="Term Lengths",
                                 legend=False,
                                 autopct='%1.1f%%',
                                 explode=(0, 0.1),
                                 shadow=True,
                                 startangle=0);
```



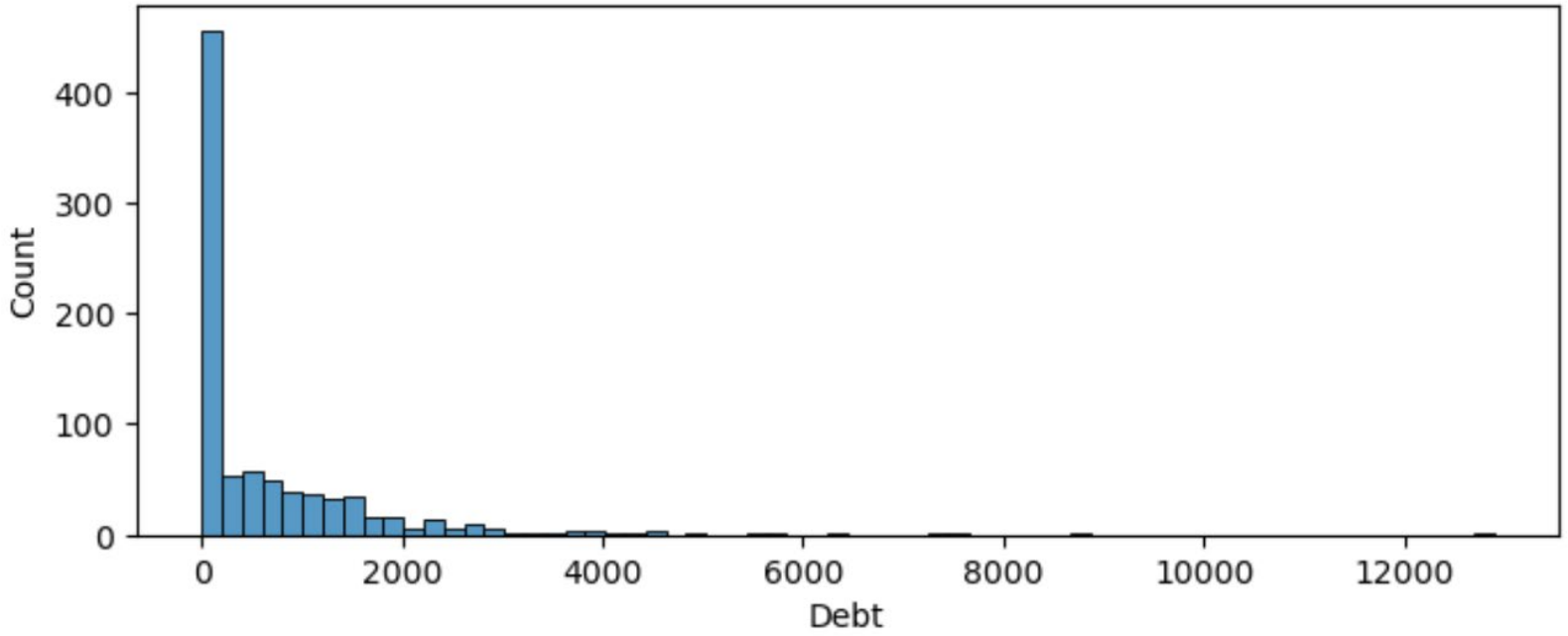
Box and whisker plots

```
fig, ax = plt.subplots(figsize=(8, 4))
sns.boxplot(data=df,
            x = "Default",
            y = "Debt",
            ax = ax)
```



Histograms

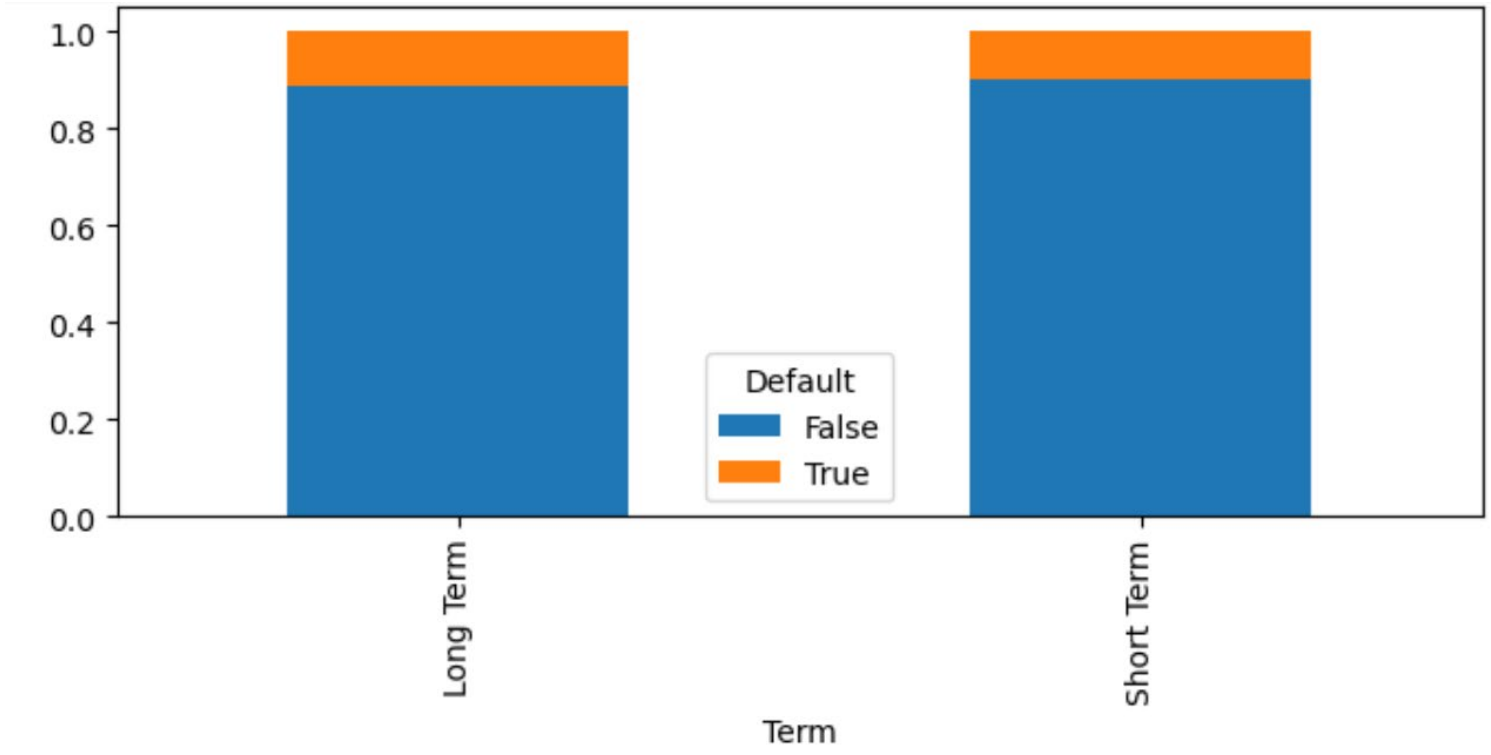
```
fig, ax = plt.subplots(figsize=(8, 3))
sns.histplot(data=df,
             x = "Debt",
             ax = ax);
```



Bivariate plots

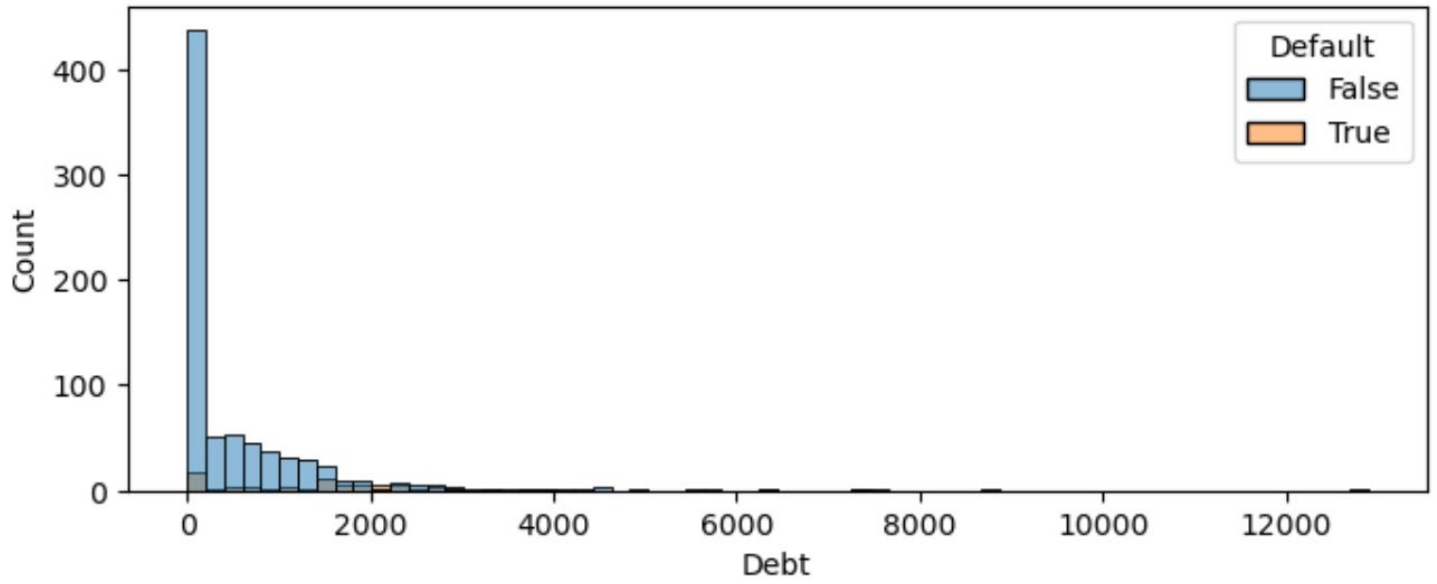
Bar charts

```
fig, ax = plt.subplots(figsize=(8, 3))
table = pd.crosstab(df['Term'], df['Default'])
table.div(table.sum(1).astype(float), axis=0).plot(kind='bar', stacked=True, ax=ax);
```



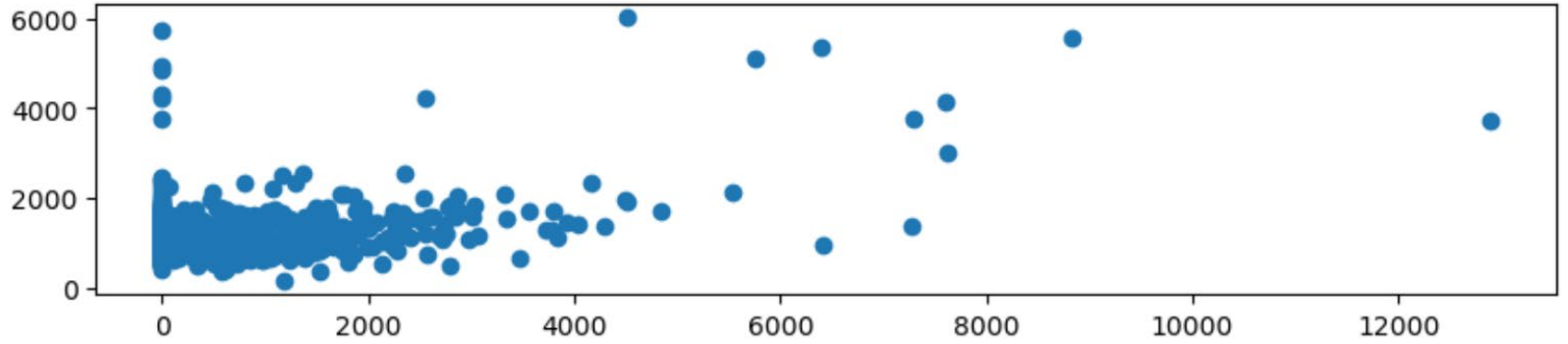
Histograms

```
fig, ax = plt.subplots(figsize=(8, 3))
sns.histplot(data=df,
             x="Debt",
             hue='Default',
             ax= ax);
```



Scatterplots

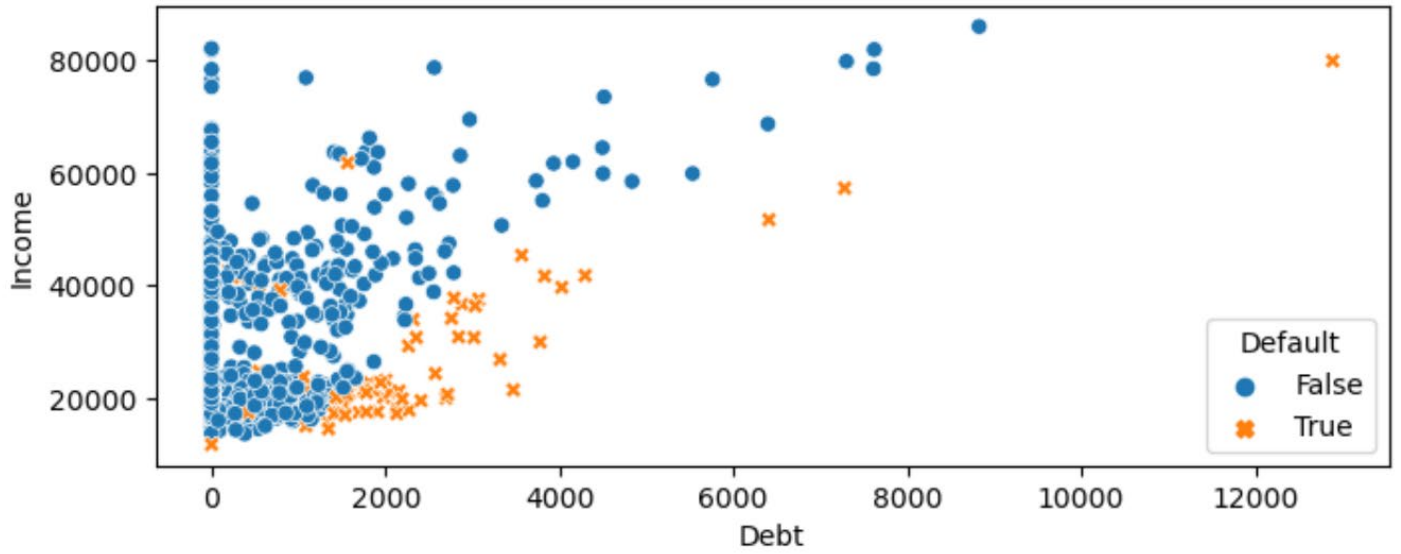
```
fig, ax = plt.subplots(figsize=(10, 2))

plt.scatter(df['Debt'],
            df['Balance']);
```



Seaborn: Making plotting easier

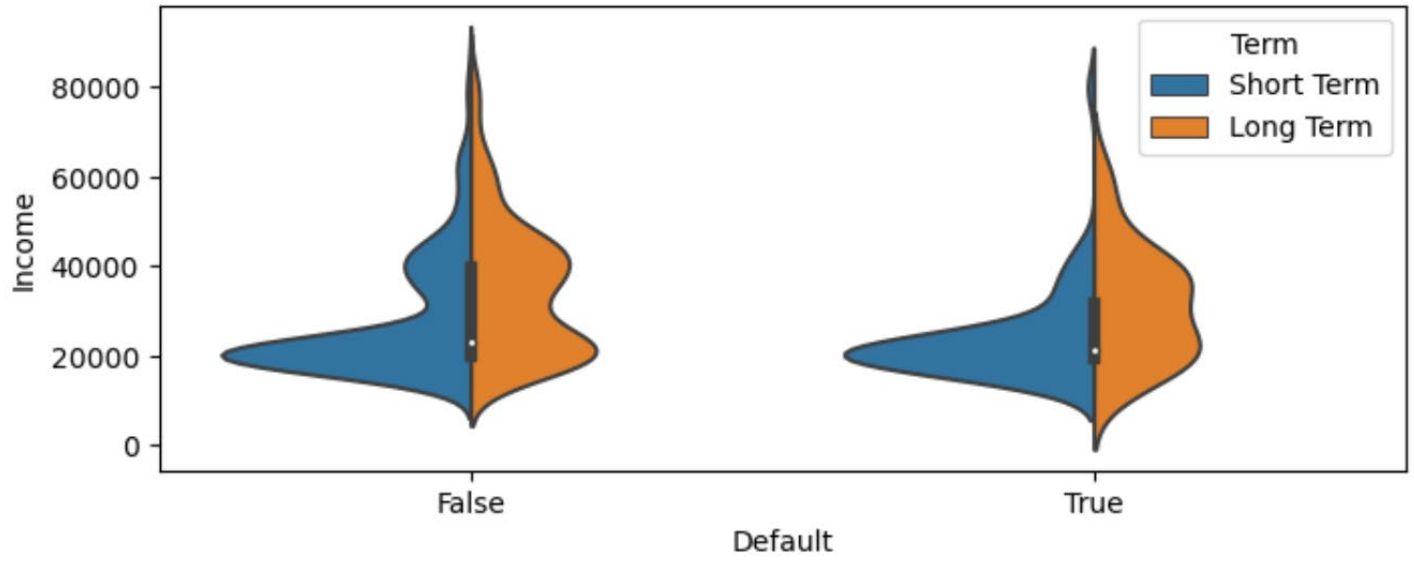
Scatterplots

```
fig, ax = plt.subplots(figsize=(8, 3))
sns.scatterplot(data=df,
                x="Debt",
                y='Income',
                hue="Default",
                style='Default',
                ax=ax);
```



Violin plots

```
fig, ax = plt.subplots(figsize=(8, 3))
sns.violinplot(data=df,
               x="Default",
               y='Income',
               hue="Term",
               split=True,
               ax=ax);
```

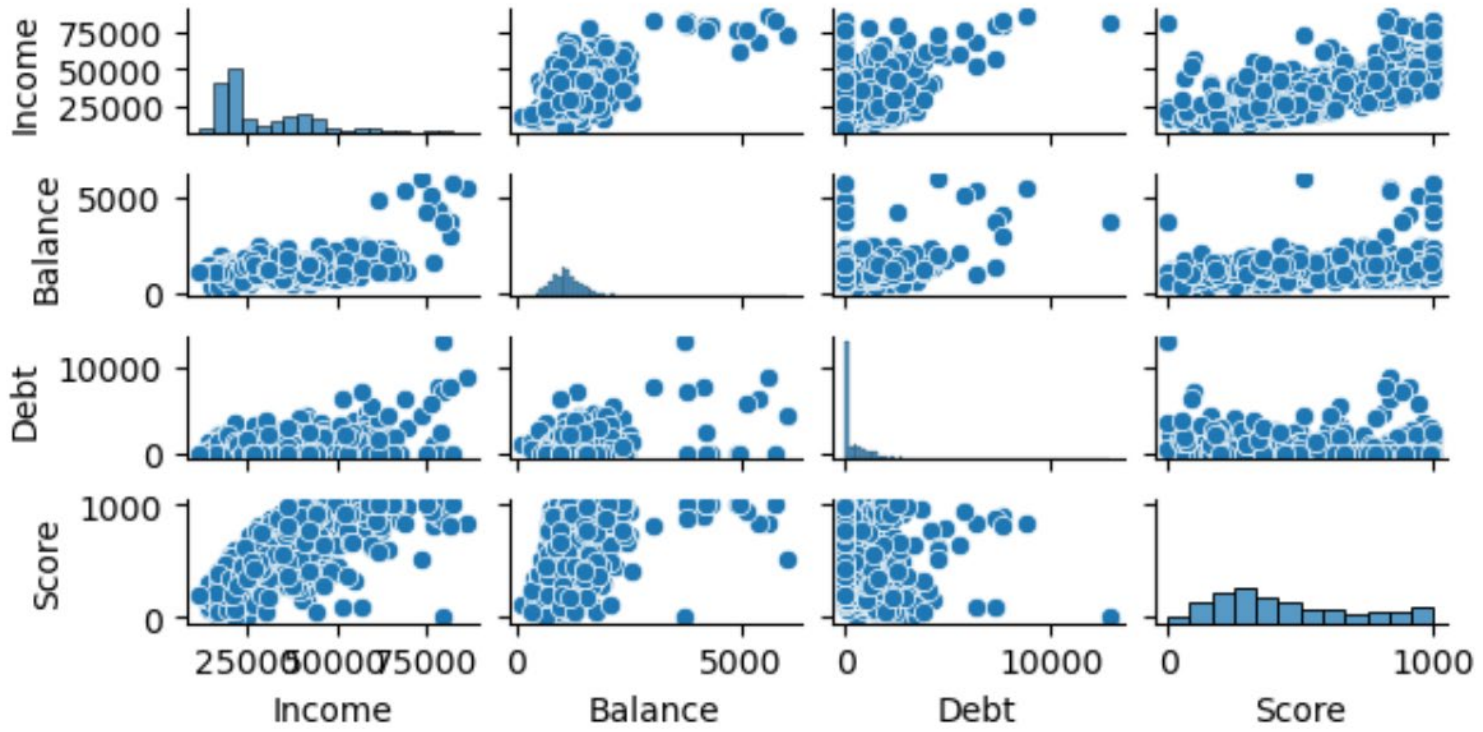


Heatmaps

```
fig, ax = plt.subplots(figsize=(8, 3))
sns.heatmap(data=df[['Income', 'Balance', 'Debt', 'Score', 'Default']].corr(method='spearman').round(2),
            center=0.0,
            vmin=-1,
            annot=True,
            cmap='coolwarm');
```

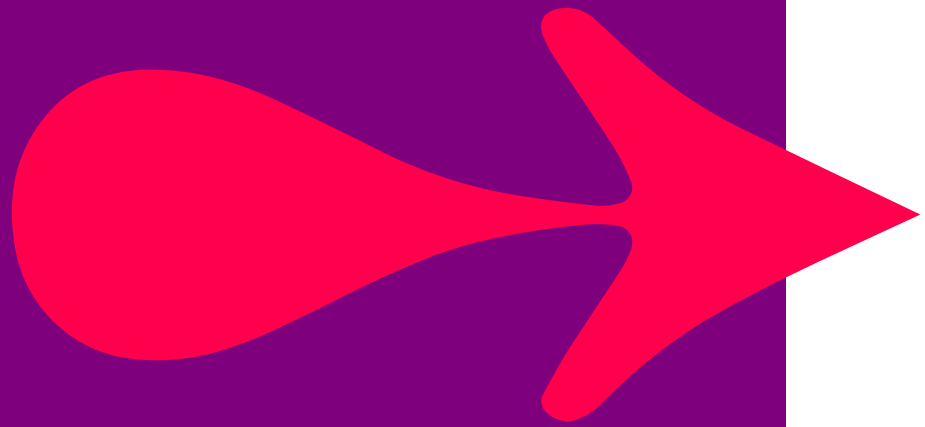
Pairplots

```
sns.pairplot(data=df[['Income', 'Balance', 'Debt', 'Score']], height=0.8, aspect=2)
```



Exercise

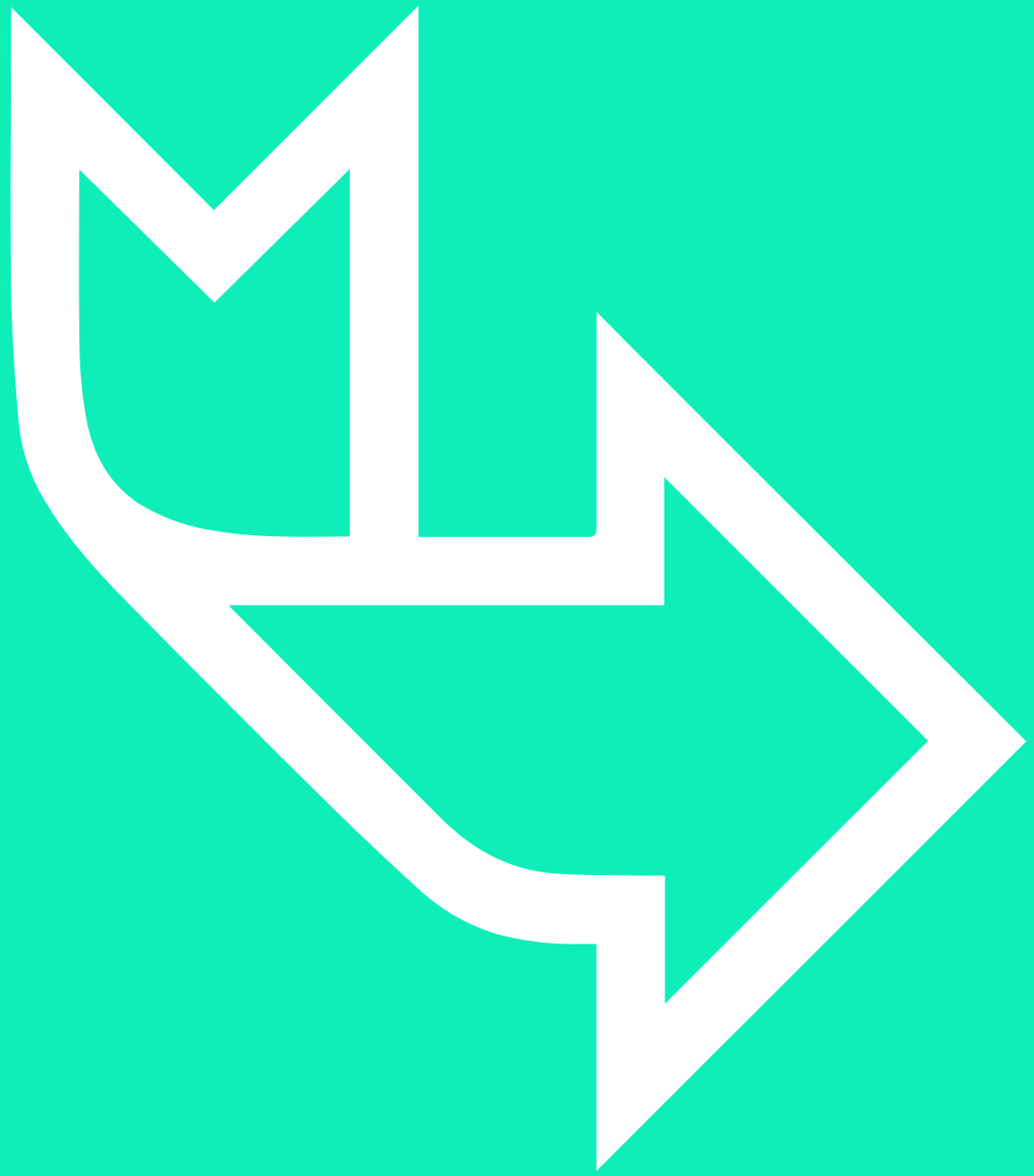
Go to **Exercise 8: Methods for visualising data** in your exercise guide.



Learning check

Think about your answers to these questions:

- What visualisations can we use to understand non-numeric data?
- What visualisations can we use to understand numeric data?
- What is the difference between Matplotlib and Seaborn?



How did you get on?

Learning objectives

- Construct and tailor basic data visualisations using Matplotlib and Seaborn for both numeric and non-numeric data.
- Meaningfully visualise aggregate data using Matplotlib and Seaborn.