Toolkit for Finance

Week 5: Working with financial data: Introduction to Pandas

Lecturer: Dr. Yiping LIN

Overview

- Working with text files

- open function
- reading text files
- context managers
- writing text to files

- Working with financial data: Introduction to Pandas

- Series
- DataFrame
- Numpy

Note: You will need the following 5 companion code (shared on Ed code section): `lec_fileio.py`; `lec_avgs_example.py`; `lec_pd_series.py`; `lec_pd_dataframes.py`; `lec_pd_numpy.py`

1.0 Working with text files - Overview

Reading and Writing Files (I/O) - open function

The built-in function `open` accesses a file, returning a handler called a *file object*. It is important that we differentiate the *file* (the one saved on the computer) from the *file object* handler (the object in Python that lets us access the file on the computer).

- Computer files are used to store information. The computer natively sees this information as *binary data*, a sequences of zeros and ones. We use *byte-based* terms like *kilobytes*, *megabytes*, *gigabytes*, or *terabytes*, to measure the amount of information stored by a computer. Applications convert these bytes (*binary data*) into something humans can understand.
- When you use the `open` function in Python, you are opening a **line of communication** between you (the user) and these bytes stored in the computer. The interface we use to work with the file is the *file object*.

Consider the `qan_stk_prc.csv` file we created to store the Qantas stock price data downloaded from Yahoo! Finance. This file is a text file. We can either *read* text from this file or *write* to it.

- There is an important difference between reading and writing files -- when you read a file, you do not modify the data stored in your computer.

1.0 Working with text files - Overview

Intuitively, opening a file for *reading* in Python involves the following steps:

- Find the file in your computer
    - If the file does not exist, raise an exception and stop the program
- Return a file object
    - This opens a line of communication between Python and the file.
    - Importantly, the communication will only flow in one direction -- from the file stored in the computer to Python.

Once this line of communication is open, we can ask the file handler to "read" the information stored in the file and return its contents as Python objects.

- But remember that the information is stored as streams of bytes. If we want to get the "text" represented by these bytes, we first need to **decode** these bytes.
    - In other words, we want the file handler to convert bytes into `str` instances.
- the `open` function allow you to specify how the information contained in the file will be returned to you -- either as raw bytes (`bytes` objects) or as decoded text (`str` objects).

1.0 Working with text files - Overview

The following will open the `qan_stk_prc.csv` for reading as text (`str` objects):

```
fobj = open('qan_stk_prc.csv', mode='rt')
```

Think of this statements as follows:

- The function `open` will
    - Look for a file called `qan_stk_prc.csv` in the current working directory. If this file is not found, Python will raise an exception and exit.
    - Interpret the parameter `mode='rt'` as an instruction as to how we will communicate with the file. The value `rt` is a combination of two instructions:
        1. `r` means "open this file for reading" -- you will not be able to modify the contents of `qan_stk_prc.csv`.
        2. `t` means "open this file in *text mode* " -- the contents will be automatically converted from bytes to text (`str`)
    - Create and return a *file object*.
    - Create a variable called `fobj` and assign it to the file object returned by `open` .

1.0 Working with text files - Overview

So far, all we have done is to create a *file object*, or a line of communication. We have not yet read anything from the file!
By default, the `open` function will open a file for reading (the "r" in `mode`) and in text mode (the "t" in `mode`). This means that the following statements are equivalent:

```
# Explicitly setting the "read" and "text" modes
fobj = open('qan_stk_prc.csv', mode='rt')

# Using the default "text" mode
fobj = open('qan_stk_prc.csv', mode='r')

# Using the default "read" and "text" modes
fobj = open('qan_stk_prc.csv')
```

1.1 Customising the `open` mode parameter for reading, writing, and appending

The `open` function is meant to be a general function for ***reading and writing*** both ***text and binary*** files. We control this behaviour by customising the `mode` parameter. The mode parameter is a string consisting of two characters:

- The first character indicates the **access mode** -- whether we want to read or write to a file. Three options for the mode parameter are:
    - `'r'`: Open a file for reading. This is the *default* which Python will assume if it does not see a `'w'` or `'a'` in the mode parameter string.
    - `'w'`: Opening a file for writing, erasing any previous content if any.
    - `'a'`: Opening a file for writing, *appending* content to the end of the file if it exists.
- The second character indicates the **data type** -- whether we want to work with text or binary data.
    - `'t'`: Work with text data. This is the *default* which Python will assume if it does not see a `b` in the mode parameter string.
    - `'b'`: Work with binary data.

1.1 Customising the `open` mode parameter for reading, writing, and appending

We construct our mode parameter string by combining the *access mode* character with the *data type* character.

- As we saw previously, we can open a text file for reading explicitly using the mode string `'rt'`.
- While the string order doesn't matter, we can also use `'tr'`, the standard is to specify the *access mode* followed by the *data type*.
- We can also leverage the defaults by specifying the mode parameter string `r '` and `'t'`, or by omitting the mode parameter all together.

1.1 Customising the `open` mode parameter for reading, writing, and appending

Let's work through the example of writing a text file. Consider the following statement:

```
fobj = open('qan_stk_prc.csv', mode= 'wt')
```

What Python will do in this case is:

- Look for a file called `qan_stk_prc.csv` in the current working directory.
    - If this file is not found, *create it*. This will create a new empty file.
    - If this file exists, *erase any existing content first*.
- Create a file object which will allow you to:
    - Write data to this file (the `'w'` in `mode`)
    - Convert text (`str`) into bytes automatically (the `'t'` in `mode`).
- Create a variable called `fobj` which will hold this file object.

Importantly, although nothing will be written to the file yet, the contents of an existing file will be erased! We need to be very careful when opening files in *write mode*! As before, we can omit the `'t'`, because the default is to open a file in *text mode*.

1.1 Customising the `open` mode parameter for reading, writing, and appending

What happens when you open a file in *append mode* (e.g., `fobj = open('qan_stk_prc.csv', mode='at')`)?

- The steps are very similar but, if the file exists, its contents will not be erased first.
- Instead, any content written to this file will be appended to the end of this file.

For most of this course, we will only open files in *text mode*. Since this is the default, we will omit the `'t'` in the `mode` parameter. To open a file in *binary mode*, you need to add `'b'` to the `mode` parameter:

- `mode='rb'` : Opens a file for reading in binary mode
- `mode='wb'` : Opens a file for writing in binary mode
- `mode='ab'` : Opens a file for appending in binary mode

When you open a file in binary mode, the file object will return `bytes` when reading from the file, and will write `bytes` objects to the file.

1.1 Customising the `open` mode parameter for reading, writing, and appending

The table below lists the most common flags used with the `mode` parameter. For a complete description of the open function and its parameters, see the **Python documentation**.

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, erasing any existing file first
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)

Next, we will learn how to use file objects to read content from a file.

1.1 Reading text files in Python

Before continuing, you should open the companion code in PyCharm:

Companion code for this lesson:
Please open the module `lec_fileio.py` in PyCharm. This module is inside the `lectures` package. The diagram below shows the location of this module in the `toolkit` project folder:

```
toolkit/                    <-- PyCharm project folder
|   ...
|___lectures/
|   |   lec_fileio.py   <-- Companion code for this lesson
```

1.1 Reading text files in Python

The `lec_fileio.py` will import the following modules:

```
import os
import toolkit_config as cfg
```

The `lec_fileio.py` uses the `toolkit_config` module we created previously to define two constants, `SRCFILE` and `DSTFILE`.

- These constants are strings pointing to the location of an existing file (`qan_prc_2020.csv` inside the `data` folder) and the location of a new file we will create inside the `data` folder:

```
SRCFILE = os.path.join(cfg.DATADIR, 'qan_prc_2020.csv')
DSTFILE = os.path.join(cfg.DATADIR, 'new_file.txt')
```

You should already have the `qan_prc_2020.csv` file from one of the code challenge questions. If not, you can download the file shared under the Ed code section

1.1 Reading text files in Python

Open the `lec_fileio.py` module and complete the relevant sections in the code samples as shown below.

- First, open the `SRCFILE` for reading (in the default *text mode*):

```
# ------------------------------------------------------------------------
#   Opening the `SRCFILE` and reading its contents
# ------------------------------------------------------------------------
# This will open the file located at `SRCFILE` and return a handler (file
# object):
fobj = open(SRCFILE, mode='r')
```

1.1 Reading text files in Python

We can now use the `fobj` to read the contents of the file. The file object method `read` will return a string with the entire contents of the file. You can then print the first 20 characters by slicing this string:

```
# We can get the entire content of the file by calling the method `.read()`,
# without parameters:
cnts = fobj.read()

# The variable `cnts` will be a string containing the full contents of the
# file. This will print the first 20 characters:
print(cnts[:20])

# Output:
#  Date,Open,High,Low,C
```

1.1 Reading text files in Python

At this point, the line of communication between `fobj` and the file is still open.

- You can check if a "file is open" in Python by calling the file object attribute `closed`. If the file is open, this attribute will be set to `False`:

```
# Check if the file is closed
print(fobj.closed)  # --> False
```

We can close the file by calling the method `close`:

```
# Close the file
fobj.close()
print(fobj.closed)  # --> True
```

After you close a file, you will not be able to communicate with it using `fobj`.

1.1 Reading text files in Python

The method `fobj.read` is useful to quickly retrieve the contents of a (relatively small) file.

- One problem with this method is that the entire content of the file will have to be stored in memory. If the file you are reading is very large, loading the contents in full may take a long time, slow your computer significantly, or even consume all its memory.

A better approach is to read the contents of the file in smaller, sequential pieces.

- Only a single line of text would be stored in the computer memory at a time, speeding up access and limiting the amount of computer memory needed.

1.1 Reading text files in Python

File objects are actually specifically designed to read data sequentially. In fact, they are *iterable* objects.

- For text files, the file object will iterate over each line in the file. In other words, it will read one line in the file, return its content, and then move to the next line. Objects that behave like that are called *iterators*.
- You should think of an iterator as a handler that will return data sequentially, one element at a time.
    - The trick here is that iterators do not "hold" the data. All an iterator cares about is the location of the element in the data and how to move to the next element.

One way to think about the relation between the file object and the file is the following:

- The iterator (file object) starts at the beginning of the file (the data). Lets call this location "line 0". - - When we ask the iterator for data, the iterator reads the contents of "line 0", returns it to you, and moves to a new location, "line 1".
- This means that the iterator only needs to know the current line number. It does not keep track of the content previously returned to you.

1.1 Reading text files in Python

You can create iterators using the built-in function `iter`. To see how that works, let's try the following code in a Python Shell:

```
>>> data = ['a', 'b', 'c'] # This is the data
>>> my_iter = iter(data)   # Creates an iterator over the object `data`
>>> next(my_iter) # Current position is 0 --> Fetch data[0] --> move to position 1
'a'
>>> next(my_iter) # Current position is 1 --> Fetch data[1] --> move to position 2
'b'
>>> next(my_iter) # Current position is 2 --> Fetch data[2] --> stop
'c'
```

- The iterator above does not hold the data in `data`. It only keeps track of its current position. The iterator starts at position 0. When you ask for the next element in the iterator, it looks up the element of `data` at position 0, returns it to you, and then sets the new position as 1.
- When we ask for the next element, the iterator will look for the element in `data` in position 1, return it, and move to the next position, 2.
- What happens when the iterator above reaches position 2 (the last position in `data`)?
    - Try it!

1.1 Reading text files in Python

Using iterators with the `next` function is very cumbersome. Fortunately, you are actually familiar with iterators in looping.

- Recall from Week 3 that a `for` loop is specified in terms of `for <variable_name> in <collection(iterable)>:` . So, we can place any iterator in the `for` loop and work through the elements one by one.
- This approach ensures that the iterator, our *file object*, stops when it reaches its end without throwing an error. This means that we would generally like to iterate over a file using:

```
for line in fobj:
  # Code to process line goes here
```

Note that we *omit* the `fobj.read()` method, and use the iterable form of `fobj` .

- As an exercise, let's see how we can use an iterable file object to read in an entire file. In other words, we want to recreate the behaviour of `fobj.read()` using the iterable `fobj` (without `.read()`).

1.1 Reading text files in Python

Lets go back to file objects and the `lec_fileio.py` module. Uncomment the statements below and run the module:

```
# ------------------------------------------------------------------------
#   Comparing different approaches to get the contents
# ------------------------------------------------------------------------
# Remember that we previously closed the file so we need to open it again
fobj = open(SRCFILE, mode='r')
# Contents using `.read`
cnts = fobj.read()
print(f"First 20 characters in cnts: '{cnts[:20]}'")

# Start with an empty string
cnts_copy = ''
# Iterate over each line of fobj
for line in fobj:
    # Add this line to the string `cnts_copy`
    cnts_copy += line

# Print the result
print(f"First 20 characters in cnts_copy: '{cnts_copy[:20]}'")

# Close the file
fobj.close()
```

Running this module will return:

```
First 20 characters in cnts: 'Date,Open,High,Low,C'
First 20 characters in cnts_copy: ''
```

What happened?

- This is a very common source of confusion when dealing with file objects -- File objects will iterate over the contents of the file *and then stop*!
- *Since the method* `read` *will iterate over the entire file first*, after we call this method the iterator reached the end of the file. This means that there is nothing to iterate over in the loop `for line in fobj...`.
- This is similar to what happens when you iterate over an empty list -- The loop does not even start. Try the following in the Python shell:

```
>>> lst = []
>>> for x in lst:
...     print("HERE")
>>>
```

Nothing will be printed...

1.1 Reading text files in Python

Go back to the `lec_fileio.py` file and **replace the `cnts = fobj.read()` statement with `cnts = ''`** . That section of your code becomes:

```
# ------------------------------------------------------------------------
#   Comparing different approaches to get the contents
# ------------------------------------------------------------------------
# Remember that we previously closed the file so we need to open it again
fobj = open(SRCFILE, mode='r')
# Contents using `.read`
# cnts = fobj.read() ## OLD
cnts = '' ##### NEW
print(f"First 20 characters in cnts: '{cnts[:20]}'")

# Start with an empty string
cnts_copy = ''
# Iterate over each line of fobj
for line in fobj:
    # Add this line to the string `cnts_copy`
    cnts_copy += line

# Print the result
print(f"First 20 characters in cnts_copy: '{cnts_copy[:20]}'")

# Close the file
fobj.close()
```

Run the module again and you will see the following output:

```
  First 20 characters in cnts: ''
  First 20 characters in cnts_copy: 'Date,Open,High,Low,C'
```

The reason we can now see the right content in `cnts_copy` is because this was the first time we iterated over the contents of the file.
To make sure this is clear, modify the next section of the module so that it reads:

```
# ------------------------------------------------------------------------
#   Reading one line at a time
# ------------------------------------------------------------------------
fobj = open(SRCFILE, mode='r')

# Read the first line
first_line = next(fobj)

# After that, the fobj iterator now points to the second line in the file
for line in fobj:
    print(f"fobj now point to : '{line}'")
    break

fobj.close()
```

The first statement will return the first line of the file and move the iterator to the next one. The loop will then start at the second line of the file. After one iteration, we will exit the loop (after `break`).
This will return:

```
fobj now point to : '2020-01-
02,7.139999866485596,7.210000038146973,7.119999885559082,7.159999847412109,6.985208034515381,4980666'
```

1.1 Context managers

You will notice that we take the time to close the file in our last example code:

```
fobj.close()    # <--- CLOSES THE FILE!!!
```

Calling the `.close()` method ends the line of communication established by the function `open`.

- Ensuring that the file is closed is *very* important. This is especially true when writing content to files
    - Because of the way Python writes data to files, some of the data will not be written if the program crashes.
- Without going into too much detail, please keep the following in mind: *Bad things can happen if you don't close your files...*

1.1 Context managers

It is surprisingly difficult to remember to close files (and even if you do, bad things can still happen when a program crashes before a file is closed). Luckily, Python offers a simple solution in the form of **context managers**.

- Context managers isolate the part of our code with our file object in a new *scope*. Exiting the context manager will exit the *scope*, which will automatically close the file object we opened. This may seem like a hassle, but it is not. You will seldom see the `open` function outside a context manager.
- In this course we will keep it simple: **You should always use context managers when opening files.**

The context manager begins by using the keyword `with`, followed by the `open` command and an alias for the object returned. The expressions inside the context managers are indented.

- Just as with a function, once we leave the indented block (the context manager scope), we exit the manager (and the scope).
- Let's see an example next

1.1 Context managers

Back to the `lec_fileio.py` module, and uncomment the necessary lines so that your module looks like this:

```
# ------------------------------------------------------------------------
#   Using context managers
# ------------------------------------------------------------------------
# Instead of fobj = open(SRCFILE, mode='r'), use a context manager:
with open(SRCFILE, mode='r') as fobj:
    cnts = fobj.read()
    # Check if the object is closed inside the manager
    print(f'Is the fobj closed inside the manager? {fobj.closed}')

# Notice that we did not close the object when using a context manager
# But after exiting the context manager, the file will automatically close
print(f'Is the fobj closed after we exit the manager? {fobj.closed}')
```

In the above, the `open` command gives us the *file object* in the variable `fobj` .

- Everything indented relative to the `with` keyword is in the scope of the context manager.
- We proceed to read the file. Our first `print` statement confirms that the file object is still open in the *context manager*.
- Once we leave the *context manager*, Python automatically closes the file object. We confirm that when we exectue the final `print` statement at the outermost scope level.

1.1 Writing text to files using Python

Opening a file for **writing** is simply a matter of setting the parameter `mode='w'` in the `open` method's `mode` parameter.

- Remember that any existing file will be erased when using write mode `w`.
- Once the file is open, we can use the *file object* method `write` to add content to the file.

The example below demonstrates writing a single line to a file.

- We first define a simple auxiliary function to display the contents of a file, line by line, which will allow us to confirm the contents of any file we write. This function is already in your `lec_fileio.py` module.
- We then use a context manager to open `DSTFILE` for writing and store the manager in `fobj`. The indented line `fobj.write('This is a line')` adds a single line to the file, the context manager exits, and the file is closed.
- We then call our `print_lines` diagnostic function to check that the file was written correctly:

1.1 Writing text to files using Python

```
# Auxiliary function to print the lines of a file
def print_lines(pth):
    """ Function to print the lines of a file
    Parameters
    ----------
    pth : str
        Location of the file
    Notes
    -----
    Each line in the file will be printed as
        line number: 'string with the line text'
    """
    with open(pth) as fobj:
        for i, line in enumerate(fobj):
            print(f"line {i}: {line}")

#  This will create the file located at `DSTFILE` and write some content to it
with open(DSTFILE, mode='w') as fobj:
    fobj.write('This is a line')

# Exiting the context manager will close the file
# We can then print its contents
print_lines(DSTFILE)
```

Running the module will produce:

```
line 0: This is a line
```

1.1 Writing text to files using Python

If you open the same file again in writing mode, the line we wrote above will be erased:

```
# If you open the same file again in writing mode, the line we wrote above
# will be erased:
with open(DSTFILE, mode='w') as fobj:
    fobj.write('This is another line')

print_lines(DSTFILE)
```

This will give:

```
  line 0: This is another line
```

1.1 Line endings when reading and writing text files

One common source of confusion is that (by default), lines are not terminated with the write statement.

- Thus, calling two write statements sequentially simply produces a single line containing the content of each write statement, for example,

```
# ------------------------------------------------------------------------
#   The write method does not add terminate the line.
# ------------------------------------------------------------------------
with open(DSTFILE, mode='w') as fobj:
    fobj.write('This is a line')
    fobj.write('This is a another line')
print_lines(DSTFILE)
```

This will produce:

```
  line 0: This is a lineThis is another line
```

1.1 Line endings when reading and writing text files

To terminate a line, you must include a newline character, written as "'\n'" in a string.

- newlines are not included after the string passed to the `write` method, we need to add them ourselves.
- So, to separate the lines in Python, we should change our code to:

```
# ------------------------------------------------------------------------
#   Notice that the write method does not add a newline character (\n). You
#   must add it yourself:
# ------------------------------------------------------------------------
with open(DSTFILE, mode='w') as fobj:
    fobj.write('This is a line\n')
    fobj.write('This is a another line')
```

1.1 Line endings when reading and writing text files

However, this brings up another area of confusion and perhaps frustration.

- When we open a file for reading, the string returned will *include* newline characters. But, Python's `print()` method automatically adds a newline after the contents.
- So, printing a line that ends with a newline will result in your output having two newlines -- the one from the line and the one added automatically by the `print()` method. Let's confirm this is the case:

```
# ------------------------------------------------ ----------------------------
#   Notice that the write method does not add a ne wline character (\n). You
#   must add it yourself:
# ------------------------------------------------ ----------------------------
with open(DSTFILE, mode='w') as fobj:
    fobj.write('This is a line\n')
    fobj.write('This is a another line')
print_lines(DSTFILE)
```

Executing this code produces the following output:

```
  line 0: This is a line

  line 1: This is a another line
```

Should you wish to eliminate newlines, you can use the `str.rstrip()` method. This removes any newlines and spaces at the end of the string. Thus, we can create a new `print_lines_rstrip` function that removes all newlines:

1.1 Line endings when reading and writing text files

```
# Auxiliary function to print the lines of a file
def print_lines_rstrip(pth):
    """ Function to print the lines of a file
    Parameters
    ----------
    pth : str
        Location of the file
    Notes
    -----
    Each line in the file will be printed as
        line number: 'string with the line text'
    """
    with open(pth) as fobj:
        for i, line in enumerate(fobj):
            print(f"line {i}: '{line.rstrip()}'")  #<---New

with open(DSTFILE, mode='w') as fobj:
    fobj.write('This is a line\n')
    fobj.write('This is a another line')
print_lines_rstrip(DSTFILE) # <comment>
```

Executing this code produces the following output:

```
line 0: This is a line
line 1: This is a another line
```

In-Class Exercise 1: Create a function to find the most common word and its frequency in a text file?

- Let's mix the function, context manager, basic data structures, loops, conditional executions, statement assignment, and f strings concepts and syntax we have learnt so far to solve this problem.
- You may use the sample file (iso.txt) shared on Ed code section to test your code.
- Hint:
    - read file use open( ) function;
    - split lines using split( ) function;
    - you may also count words use get( ) function to simplify a bit.
        - The get() method returns the value of the item with the specified key.
        - Syntax: `dictionary.get(keyname, value)`
            - keyname (Required) - The keyname of the item you want to return the value from
            - value (Optional). A value to return if the specified key does not exist. Default value None

```
### A Basic Sample Solution

def freqword(filepath):
    with open(filepath) as file:
        # Count word frequency
        counts = dict()
        for line in file:
            words = line.split()
            for word in words:
                counts[word] = counts.get(word,0) + 1

        # Find the most frequent word
        maxcount = None
        maxword = None
        for word,count in counts.items():
            if maxcount is None or count > maxcount:
                maxword = word
                maxcount = count

    # Return the result
    return(f"The most frequent word is: {maxword}, and the number of times appeared is: {maxcount}")

### Call the function
freqword('iso.txt')
```

2.0 Introducing Pandas - Overview

- Pandas is a Python package providing fast, powerful and flexible open source data analysis and manipulation tool, built on top of the Python.
- It aims to be the fundamental high-level building block for doing practical, real-world data analysis in Python, and well suited for many different kinds of data (e.g. tabular data as in an SQL table or Excel spreadsheet, time series data).
- The two primary data structures in pandas: **Series (1-dimensional) and DataFrame (2-dimensional)**
    - handle the vast majority of typical use cases in finance, statistics, social science, and many areas of engineering.
    - For R users, DataFrame provides everything that R's data.frame provides and much more. pandas is built on top of NumPy and is intended to integrate well within a scientific computing environment with many other 3rd party libraries.

2.1 Organising Financial Data with Pandas - Data in tabular form

Previously, we downloaded and saved Qantas share prices in a CSV file called `qan_prc_2020.csv`. This file is available to you inside the `data` folder of the toolkit package.
When we open the `qan_prc_2020.csv` file in Excel, we are presented with a spreadsheet containing the following data:

	A	B	C	D	E	F	G
1	Date	Open	High	Low	Close	Adj Close	Volume
2	2020-01-02	7.1400	7.2100	7.1200	7.1600	6.9852	4980666
3	2020-01-03	7.2800	7.3100	7.1600	7.1900	7.0145	2763615
4	2020-01-06	7.0100	7.0300	6.9100	7.0000	6.8291	7859151
5	2020-01-07	7.2300	7.2550	7.0800	7.1000	6.9267	7589056
6	2020-01-08	7.0500	7.0800	6.7600	6.8600	6.6925	13449760
7	2020-01-09	6.9700	7.0000	6.9000	6.9500	6.7803	6173211
8	2020-01-10	6.9900	7.0600	6.9600	7.0000	6.8291	4450193
9	2020-01-13	6.9800	7.0600	6.9800	7.0200	6.8486	3842992
10	2020-01-14	7.0900	7.1500	7.0800	7.1100	6.9364	4958613
11	2020-01-15	7.1700	7.2000	7.0200	7.0400	6.8681	10472071
12	2020-01-16	7.0400	7.0700	7.0200	7.0700	6.8974	10728336
13	2020-01-17	7.1200	7.1500	7.1000	7.1500	6.9755	9619822

2.1 Organising Financial Data with Pandas - Data in tabular form

Data presented in this format is in *tabular form*. *Tabular form* data is often represented by items located in rows and columns, forming a two-dimensional matrix.

- Most commonly, we have different types of data in columns. In this case, we have data on a date (column A), the price of a stock when the market opened (column B), the highest (column C) and lowest (column D) prices the stock achieved during the day, the price of the stock at the market close (column E), a closing price that is adjusted for stock splits, dividends and other events (column F), and the total trading volume (column G) for the day.
- Rows serves to link the cells across columns. In this case, column A provides us with a date. We then know that each data item in a given row refers to the date provided in column A.
    - For example, we know that in row 7, the open, high, low, and close prices listed in columns B, C, D, and E, respectively, are all from the date listed in column A, 9 January 2020.

2.1 Why Pandas: Critiquing Excel and the Python standard library

Pandas is a package that, among other things, provides a convenient and efficient way to handle *tabular form* data. But, if Excel can handle tabular data, why do we need Pandas?

- The reason is that Pandas provides a combination of flexibility and power. But that comes at a cost, at least initially, in the form of complexity. It takes a practice to get comfortable with Pandas.

However, even at the most basic level, looking up data, Pandas offers significant advantages over Excel. We're all so used to spreadsheets programs, that we don't ever question how they work or if they can be improved.

```
- Spreadsheets force us to access data by letter and number in all situations, which is not particularly intuitive or user
friendly. You wouldn't ask someone in conversation for the opening price of a stock on 15 Jan 2020 as "What was Qantas in
column D, row 5".

- It is better to refer to data by named indexes. For example, if we want the opening price data for a specific date, we would
say: "What was Qantas's opening price on 15 January 2020?". Pandas let's us do this.
```

2.1 Why Pandas: Critiquing Excel and the Python standard library

Our strategy to begin our transition from spreadsheets to Pandas is as follows. We'll start with a very simple task, computing the average **closing** price of a stock over a time window. We'll work through this problem several ways:

- First, use Excel to compute this average. This serves as a baseline since both Excel and Pandas rely on similar elements to organise and manipulate tabular data.
- Next, we will use Python's standard library to perform the same task. This will illustrate some advantages versus Excel, but still leave room for improvement.
- After that, we will be ready to discuss Pandas and its advantages over both Excel and the Python standard library.

2.1 Calculate Qantas average price during the second week of 2020: Excel

To compute Qantas' average closing price during the second week of 2020, we start by identifying the relevant dates. The second week of January began on Monday, January 6, and ended on Friday, January 10.
To calculate the average price of Qantas in the second week of 2020, we need to perform the following steps:

- Identify the column with the dates: column `A`.
- Identify the relevant rows in column `A` corresponding to dates from '2020-01-06' to '2020-01-10': rows `4` to `8`.
- Identify the column with close prices (called 'Close' above): column `E`.
- Define the slice (or range) of relevant data items as `E4:E8`
- Apply the function `AVERAGE` to the slice `E4:E8` by entering `=average(E4:E8)` into Excel.

2.1 Calculate Qantas average price during the second week of 2020: Excel

This works, but computations like this in Excel have a few drawbacks:

- For example, what happens if the order of the columns in the underlying source data changes? What if we add data and shift the rows up or down? What if we want a different date range? What if we want to compute the average for every week?
- We can't simply copy the average cell down one row because that will give us an average from Tuesday to Monday, not Monday to Friday. Come to think of it, how do we deal with holidays and other non-trading days? What happens if we have hundreds of thousands of observations?
- Wouldn't it be better to refer to the columns by a different label (e.g, "Close") instead of `A`, `B`, ...? Wouldn't it be better to refer to the rows directly using the dates? Ultimately, we'd like to refer to both columns and rows by the information they contain, something like "Compute the average of `['Close']['2020-01-06':'2020-01-10']`"?

Next, lets see how to compute this average using Python's standard library.

2.1 Calculate Qantas average price during the second week of 2020: Python Standard Library

Before continuing, you should open the companion code for this lesson in PyCharm:

Companion code for this lesson:
Please open the module `lec_avgs_example.py` in PyCharm. This module is inside the `lectures` package. The diagram below shows the location of this module in the `toolkit` project folder:

```
toolkit/                <-- PyCharm project folder
|   ...
|___lectures/
|   |   lec_avgs_example.py   <-- Companion code for this
lesson
```

- This module contain a scaffold for the code below. As you progress through this lesson, uncomment or complete missing statements. It is crucial that you do **not** simply copy and paste the code below into the `lec_avgs_example.py` module.

2.1 Using lists...

For the sake of exposition, we will not use the entire data at this point. Instead, we will just create two lists, corresponding to the first ten observations of the columns `Date` and `Close`.
The `lec_avgs_example.py` module defines two lists, `dates` and `prices`:

- As we know, we can access elements in a Python list by their numerical position, using zero-based indexing (i.e., the first element is at `0`).

```
# ---------------------------------------------------------------------------
#   The dates and prices lists
# ---------------------------------------------------------------------------
dates = [
  '2020-01-02',
  '2020-01-03',
  '2020-01-06',
  '2020-01-07',
  '2020-01-08',
  '2020-01-09',
  '2020-01-10',
  '2020-01-13',
  '2020-01-14',
  '2020-01-15',
  ]

# Close prices
prices = [
  7.1600,
  7.1900,
  7.0000,
  7.1000,
  6.8600,
  6.9500,
  7.0000,
  7.0200,
  7.1100,
  7.0400,
  ]
```

2.1 Using lists...

To calculate the average price from '2020-01-06' to '2020-01-10':

- Using the `dates` list, we can find the indexes for the first and last date. The list method `.index(item)` takes an argument `item`. Python searches the list to find the first element in the list equal to `item` and returns its zero-based index position.
- Slice `prices` to subset the prices for those dates.
- Compute the average as the sum of all elements divided by the number of elements. Python's `sum(iterable)` can be used to add up the values in a list of floating point numbers.

2.1 Using lists...

You can implement the steps above by adding the following to `lec_avgs_example.py` and then running the module. Try to type each statement in the `lec_avgs_example.py` module yourself.

```
# ---------------------------------------------------------------------------
#   Indexing using lists
# ---------------------------------------------------------------------------

# The `start` variable will hold the first index in the slice and the `end`
# variable will hold the last index in the slice. Remember that the `index`
# list method will return the position of the element in the list, starting at
# 0. In this case, `start` will be set to 2 and `end` will be set to 6.

# Remember to uncomment the statements below and complete the part with '?'
start = dates.index('2020-01-06')
end = dates.index('2020-01-10')
print(start, end)

# Now, slice the `prices` list.
# Remember that slices do not include endpoints
prcs_w1 = prices[start:end+1]

# Finally, calculate the average of the prices in the slice
avgprc = sum(prcs_w1)/len(prcs_w1)
print(avgprc)
```

2.1 Using lists...

This is similar to what we did in Excel. For simple tasks, `list` objects work fine. However, when performing serious data analysis (or financial data analysis in particular), `list` objects are not ideal. There are many reasons for that, including readability, synchronisation, and performance considerations.

Readability issues

- We mentioned that it would be more convenient to select prices directly using the dates. In the example above, we first defined the `start` and `end` values. Of course, we can simply do the following instead:

```
# Select the price on '2020-01-06'
prices[dates.index('2020-01-06')]

# Select the prices for the first week of 2020
prices[dates.index('2020-01-06'):dates.index('2020-01-10')+1]
```

As you can see, readability becomes an issue. Not only is `prices[dates.index('2020-01-06'):dates.index('2020-01-10')+1]` difficult to read, but it would also be very easy to forget to include the "+1" to ensure the last day is captured!

2.1 Using lists...

Synchronisation issues

- With lists, the index of each date in the `dates` list has to match the index of the price for that date in `prices`. However, the date list instance and the price list instance have no intrinsic relationship. The date list could be sorted chronologically, while the price list is sorted in ascending values, for example.
- The responsibility to make sure that the order of the elements in each list remain synchronised falls to you, as the programmer. You might accidentally drop an observation from the middle of the date list without doing so for the price list. Were this to happen, the indexing would be incorrect.

Performance issues

- Python's lists are designed to handle arbitrary collections of objects. Given this general purpose, lists are not optimised to handle data where all the objects are numbers, for example.
- Python's *NumPy* library provides significant performance enhancements for the numerical computations common in finance. *NumPy* is integrated into *Pandas*, meaning that a program employing Pandas remains reasonably speedy even when it handles a large amount of data.

2.1 Using dictionaries...

Wouldn't it be nice if we could combine both `prices` and `dates` into a single array-type object?

- Something would allow us to do easily select prices by dates without the cumbersome syntax? At the same time, it would be a significant improvement if we could ensure that when we change one of the lists (adding values, changing the order of elements), the link between dates and prices is maintained.

One way to get around the readability and synchronisation issues is to use a dictionary instead of two lists.

- dictionaries is essentially a collection of key-value pairs. You can think of each `key` as a label which uniquely identifies the object stored under it (the `value`).

2.1 Using dictionaries...

We will create a dictionary that associates each date in `dates` to the corresponding price in `prices`. In other words, we want to create a "map" between the elements of `dates` and `prices`. That will allow us to fetch prices by their dates (but not the other way around). The table below summarises the relationship:

Keys	Values
'2020-01-02'	7.1600
'2020-01-03'	7.1900
'2020-01-06'	7.0000
'2020-01-07'	7.1000
'2020-01-08'	6.8600
'2020-01-09'	6.9500
'2020-01-10'	7.0000
'2020-01-13'	7.0200
'2020-01-14'	7.1100
'2020-01-15'	7.0400

2.1 Using dictionaries...

In the `prices` list, we identified each element by it's position in the list. This is non-intuitive and error prone. In a dictionary, we instead identify each price by its label (key) instead:

```
prc_dic = {
  '2020-01-02': 7.1600,
  '2020-01-03': 7.1900,
  '2020-01-06': 7.0000,
  '2020-01-07': 7.1000,
  '2020-01-08': 6.8600,
  '2020-01-09': 6.9500,
  '2020-01-10': 7.0000,
  '2020-01-13': 7.0200,
  '2020-01-14': 7.1100,
  '2020-01-15': 7.0400,
  }
```

This dictionary makes it easy to query prices by date. We do not have to worry about keeping the elements in `prices` consistent with the elements in `dates`. This is because the map "hard-wires" the link in a single object.

2.1 Using dictionaries...

While dictionaries improve the ergonomics of querying, we cannot slice dictionaries very easily. We'd like to compute prices using something analogous to using Lists. Ideally, our code would look like the following:

```
# ---------------------------------------------------------------------------
#   Indexing using dictionaries
# ---------------------------------------------------------------------------
prc_dic = {
  '2020-01-02': 7.1600,
  '2020-01-03': 7.1900,
  '2020-01-06': 7.0000,
  '2020-01-07': 7.1000,
  '2020-01-08': 6.8600,
  '2020-01-09': 6.9500,
  '2020-01-10': 7.0000,
  '2020-01-13': 7.0200,
  '2020-01-14': 7.1100,
  '2020-01-15': 7.0400,
  }

# Get the price on '2020-01-13', in this case, 7.02
x = prc_dic['2020-01-13']
print(f'The price on 2020-01-13 is {x}')

# Try the following... it will not work because we cannot slice dictionaries
prc_dic['2020-01-02':'2020-01-13']
```

2.1 Using dictionaries...

However, because we cannot slice dictionaries, our code to compute the average prices would be ugly. One option is to make a list of the prices during the second week (`prices_wk2`) and then computing the average of the prices in the list:

```
prices_wk2 = [ prc_dic['2020-01-06'], prc_dic['2020-01-07'],
               prc_dic['2020-01-08'], prc_dic['2020-01-09'],
               prc_dic['2020-01-10']
             ]
avgprc = sum(prices_wk2)/len(prices_wk2)
```

- While the dictionary has improved the ease with which we access data, the lack of an ability to slice the data easily has led to some ugly code.
- Alternatively, we could loop over the keys and values in the dictionary to pull out the required elements. But, this would force us to examine all the elements in a dictionary regardless if we needed them, leading to poor performance with large datasets.

Another option to compute the average is to use an ordering of the keys in the dictionary.

- For example, we could loop over the dictionary and find the key with the first date, `2020-01-06` . If the keys are chronological, then we know that the next four keys define the dates through to `2020-01-10` .

While we (as humans) think of the order of a dictionary as the order of the keys -- e.g., dates are chronological. Python dictionaries do not maintain key ordering. We *strongly recommend* that you avoid thinking of dictionaries as ordered within Python.

2.2 Pandas Series, Indexes, DataFrames - Overview

The previous section examined issues with using Excel and the Python standard library for performing a simple task: computing a weekly average stock price.

- Excel, while simple, forces us to work with data in a somewhat non-intuitive way due to the forced use of letter indexing for columns and numeric indexing for rows.
- The Python standard library can make some improvements.
    - *Lists* let us identify lookup index values, much like an `xlookup` in Excel. We can then use these indices to perform calculations provided we keep all our lists synchronised and remember to add one to ending slice index.
    - *Dictionaries* improve on retrieving a single data item at the cost of making calculations employing multiple items very difficult.

2.2 Pandas Series, Indexes, DataFrames - Overview

Pandas fixes the issues listed above.

- It lets us look up naturally using labels, keeps our "columns" synchronised, and permits easy slicing for complex computations. Moreover, Pandas let us combine data more easily and more flexibly than either Excel or the Python standard library.
- And, all these features come with a relatively high performance engine, ensuring that we are able to process relatively large amounts of data easily. The **Pandas documents** provide a nice summary of the advantages of Pandas.

Next, we will introduce the main objects used in Pandas: Series, and Indexes, and DataFrames.

- Perhaps the biggest hurdle is to understand how its three main objects relate to each other and why we need them. So, it is important that you become comfortable with these objects, because understanding the logical and hierarchical structure of information in Pandas is the key to getting the most benefit from it.
- Below, we first formalise the *canonical perspective* for tabular data, which provides a foundation for the Pandas objects.

2.2 A canonical perspective

In Finance, we often focus on types of data before considering how individual data observations are organised. This corresponds to viewing the *columns* (the data types) as being the main building blocks of our dataset. Our **canonical perspective** is helpful for understanding Pandas:
A *spreadsheet* is a *collection of columns*:

- Each *column* contains a specific type of data. Columns do not mix data.
- The *column* lists individual data elements (which, in Excel, are called *cells*).
- The *column index* is a label that identify each column uniquely. In Excel, these index labels are letters (e.g., A, B, C...) and fixed. While the first row often contains a descriptive name for the data contained in the column, it is not generally possible to refer to cells by this name.
    - e.g., in the data above, column (B) row (1) is "open". However, we cannot get the first data item using `open`. Instead, we have to use the column label `B`. Unlike Excel, Pandas will let us access data using the column name.

Rows link the data across *columns*:

- The *row index* is also a set of labels separate from the column indices. The *row indices* allow us to identify elements in *any* column. In Excel, the row index label is just a number and fixed.
- The first column often contains information about the row, such as the date of observation for stock prices. However, In Excel, we must use the row number (1,2,3..) to refer such observations. Again, Pandas let us access data in a more intuitive way.

2.2 Excel from the canonical perspective

Let's see how Excel fits within the canonical perspective outlined above using our sample table of Qantas stock price data:

	A	B	C	D	E	F	G
1	Date	Open	High	Low	Close	Adj Close	Volume
2	2020-01-02	7.1400	7.2100	7.1200	7.1600	6.9852	4980666
3	2020-01-03	7.2800	7.3100	7.1600	7.1900	7.0145	2763615
4	2020-01-06	7.0100	7.0300	6.9100	7.0000	6.8291	7859151
5	2020-01-07	7.2300	7.2550	7.0800	7.1000	6.9267	7589056
6	2020-01-08	7.0500	7.0800	6.7600	6.8600	6.6925	13449760
7	2020-01-09	6.9700	7.0000	6.9000	6.9500	6.7803	6173211
8	2020-01-10	6.9900	7.0600	6.9600	7.0000	6.8291	4450193
9	2020-01-13	6.9800	7.0600	6.9800	7.0200	6.8486	3842992
10	2020-01-14	7.0900	7.1500	7.0800	7.1100	6.9364	4958613
11	2020-01-15	7.1700	7.2000	7.0200	7.0400	6.8681	10472071
12	2020-01-16	7.0400	7.0700	7.0200	7.0700	6.8974	10728336
13	2020-01-17	7.1200	7.1500	7.1000	7.1500	6.9755	9619822

Excel trains us to look up data in a specific way:

1. We first pick a column index (letter). This corresponds to a list (*series*) containing all the observations for that column. Excel is consistent with the canonical framework; the spreadsheet as a collection of columns indexed by letter.

2. Then, we pick a row index (number) to get the particular observation. Importantly, all the columns share the same indexing method. If row 5 has the date 2020 January 15, then all the columns have data for 2020 January 15 in row 5. Excel row indexing is also consistent with the canonical framework.

2.2 Pandas: Series and Indices

Pandas, among other things, allows you to efficiently work with data in tabular format. In Pandas, tabular data fits our canonical representation:

- A pandas *data frame* is a collection of *series*
    - The *column index* maps each column label to its corresponding series.
- Each *series* represents an one-dimensional array of data elements
    - The *row index* (called simply *index* in Pandas) maps row labels to their corresponding data elements

To bring Pandas into our project, we need to ensure the package is installed. Then, we need to import the Pandas package into our Python source code. The universally accepted shortcut for pandas is `pd`:

```
import pandas as pd
```

Now, we start with one important data structures in Pandas, the `Series`.

- A `Series` is like a column in Excel. Excel columns contain data and are indexed by row. Similarly, a Pandas `Series` contains data, but permits more sophisticated indexing than Excel.
- More formally, a `Series` is a one-dimensional array-like object containing *values*, where each *value* is associated with a data label, called its *index*.

2.2 Pandas: Series and Indices

Before continuing, you should open the companion code for this lesson in PyCharm:

Companion code for this lesson:
Please open the module `lec_pd_series.py` in PyCharm. This module is inside the `lectures` package. The diagram below shows the location of this module in the `toolkit` project folder:

```
toolkit/                    <-- PyCharm project folder
|   ...
|___lectures/
|   |   lec_pd_series.py    <-- Companion code for this
lesson
```

As always, resist the temptation to copy the code directly into the module. Instead, type everything in by hand. Follow along by completing the missing code as you progress through the lecture.

2.2 Pandas: Series and Indices

This module starts by importing the Pandas module (as `pd`) and defining two lists:

```
""" lec_pd_series.py

Companion codes for the lecture on pandas Series
"""

import pandas as pd

# ----------------------------------------------------------------------------
#   The dates and prices lists
# ----------------------------------------------------------------------------
dates = [
  '2020-01-02',
  '2020-01-03',
  '2020-01-06',
  '2020-01-07',
  '2020-01-08',
  '2020-01-09',
  '2020-01-10',
  '2020-01-13',
  '2020-01-14',
  '2020-01-15',
  ]

prices = [
  7.1600,
  7.1900,
  7.0000,
  7.1000,
  6.8600,
  6.9500,
  7.0000,
  7.0200,
  7.1100,
  7.0400,
  ]
```

2.2 Constructing Series objects

To understand how to build `Series` objects, lets go back to example of computing the average weekly stock price of Qantas using lists.

- We had two *separate* lists, `prices` and `dates`. By maintaining these two lists, we were able to access prices by their dates. **A Series allows us to *combine* these two lists into a single `Series` object.** This combination into a single object eliminates the need to constantly check that two separate lists are synchronised.

To create a series in Pandas, we use the `Series` constructor.

- This is part of the Pandas module, which we've imported using the alias `pd`. The constructor is therefore `pd.Series`.
- The constructor takes several parameters, but we'll focus on two, `data` and `index` for this introduction. Full documentation is available **here**.

2.2 Constructing Series objects

In the `lec_pd_series.py` module, we have:

```
# ----------------------------------------------------------------------------
#   Create a Series instance
# ----------------------------------------------------------------------------
# Create a series object
ser = pd.Series(data=prices, index=dates)
print(ser)

# Output:
#  2020-01-02    7.16
#  2020-01-03    7.19
#  2020-01-06    7.00
#  2020-01-07    7.10
#  2020-01-08    6.86
#  2020-01-09    6.95
#  2020-01-10    7.00
#  2020-01-13    7.02
#  2020-01-14    7.11
#  2020-01-15    7.04
#  dtype: float64
```

2.2 Constructing Series objects

We mentioned that a `Series` is a *one-dimensional* array-like object. But why the output of `print` shows two columns?
The first column in the output above is the *index* of this series. A Pandas `Series` object ensures that our *index* and *data* are always synchronised. But, that doesn't mean that we can't view them separately.

- The first column above (the date index) *is not part of the data* -- in the same way that the row labels `1`, `2`, ... in an Excel spreadsheet do not represent data.
- But the index *is part of the `Series` object*! The second column shows the array of *values* of this series object. Each row in the index (first column in the print output) is mapped to a corresponding value.

2.2 Constructing Series objects

As mentioned earlier, we should think of `Series` like a column in Excel. Excel columns have data and are indexable. This gives us insight on the two required parameters, `data` and `index`, to the `Series` constructor method:

- `data`: An object, generally a List, that contains our main data of interest.
- `index`: A list, or list-like object, containing the row index labels for our data.

If you are using a list for your data, it is straightforward to specify two separate lists for the `data` and `index` parameters.
If you are using a dictionary for your data, you have two options:

- To use the dictionary keys as the index and the values as the data, simply provide the dictionary to the `data` argument and specify `index=None`.
- If, however, you wish to replace the keys with a different indexing method, provide a new list of indices as the `index` parameter.

2.2 Constructing Series objects

Combining the data and the index into a single object allows us to select individual data elements by their index label directly. We can compare the two methods by adding the following to `lec_pd_series.py`:

```
# This is the series we created above
ser = pd.Series(data=prices, index=dates)

# Select Qantas price on '2020-01-02' ($7.16) using ...

# ... the `prices` list
prc0 = prices[dates.index('2020-01-02')]
print(prc0)

# ... the `ser` series
prc1 = ser['2020-01-02']
print(prc1)
```

As you can see,

- the `list` approach demonstrated in `prc0` actually involves two steps. We look up the zero-based location of the date in the list using `dates.index(...)` and then pull out the price information at that location.
- The `Series` object, on the other hand, provides direct access: we simply request the data using the date itself as in `ser['2020-01-02']`.

2.2 Selecting elements: Introduction

Pandas provides a number of ways to select elements from a series. We will talk more about that later. For now, we will stick with using `[]`.
You can use `[]` to select slices, similar to how we did it with the lists above. For example, to select the closing prices for the second week of January, 2020:

```
# ----------------------------------------------------------------------
#   Slicing series
# ----------------------------------------------------------------------
# Unlike dictionaries, you can slice a series
prcs = ser['2020-01-06':'2020-01-10']

print(prcs)

# Output:
#  2020-01-06    7.00
#  2020-01-07    7.10
#  2020-01-08    6.86
#  2020-01-09    6.95
#  2020-01-10    7.00
#  dtype: float64
```

Let's compare that with the same slice from a list:

```
# Using the lists:
prices[dates.index('2020-01-06'):dates.index('2020-01-10')+1]
```

The comparison highlights three benefits of a Series over a list:

- First, the Series is much easier to read `ser['2020-01-06':'2020-01-10']` than `prices[dates.index('2020-01-06'):dates.index('2020-01-10')+1]`.
- Second, when using index labels, Series slices **include** the endpoints, while List slices do not. Thus, we can just specify the start and end observations.
- Third, and this is not obvious from the output, but the internal operations of Series makes them relatively more efficient data structures when manipulating large amounts of data.

2.2 Series attributes

`Series` is a powerful and flexible data structure. This is, in part, because they do *not* store the data and index into simple Python lists. Instead, `Series` use the values in the `dates` and `prices` list to create specialised indexed containers.
We can think of `Series` as one-dimensional array-like objects containing:

1. An array (or series) of values, and
2. An index which associates data labels to each value

2.2 Series attributes

Lets take a closer look at each of these components. You can access these underlying components in a `Series` using attributes. Recall from Week 1 that attributes are "raw data" inside Python objects, which we can access using dot-syntax without parentheses. The `array` attribute gives us the data in the Series:

```
# ----------------------------------------------------------------------------
#   Accessing the underlying array
# ----------------------------------------------------------------------------

# This is the series we created above
ser = pd.Series(data=prices, index=dates)

# Use `.array` to get the underlying data array
ary = ser.array
print(ary)

# Output:
#  <PandasArray>
#  [7.16, 7.19, 7.0, 7.1, 6.86, 6.95, 7.0, 7.02, 7.11, 7.04]
#  Length: 10, dtype: float64

# Like any instance, you can get its type (i.e., the class used to create the instance)
print(type(ser.array))

# Output:
#  <class 'pandas.core.arrays.numpy_.PandasArray'>
#
```

2.2 Series attributes

`Series` does not store values in a python `list` or `tuple`, which is why `ser.array` does not return a `list` object. Instead, it returns a `PandasArray` object. We will talk more about these arrays later. For now, just think of it as a specialised version of a tuple -- that is, it contains data in a specific order and is immutable. What about the index? Predictably, we can access the index using the `index` attribute (again, no parentheses):

```
# Create a series object
ser = pd.Series(data=prices, index=dates)

# Use the `index` attribute to get the index from a series
the_index = ser.index
print(the_index)

# Output:
# Index(['2020-01-02', '2020-01-03', '2020-01-06', '2020-01-07', '2020-01-08',
#        '2020-01-09', '2020-01-10', '2020-01-13', '2020-01-14', '2020-01-15'],
#       dtype='object')


# Like any instance, you can get its type (i.e., the class used to create the
# instance).
print('The type of `the_index` is', type(the_index))

# Output:
# The type of `the_index` is <class 'pandas.core.indexes.base.Index'>
```

2.2 Series attributes

The index is an instance of `pandas.core.indexes.base.Index`, but what does that mean?

- In a Pandas series, each data value is associated with a label. `Index` are objects that, among other things, map each data label to a data value, similar to a dictionary.
- The index is part of the `Series` object. Pandas permits us to change the index labels without creating a new `Series` instance. In other words, `Series` are mutable.
    - For example, suppose we want to change the data labels (i.e., the index) of the series `ser`:

2.2 Series attributes

```
# ------------------------------------------------------------------------
#   Changing the index by assignment
# ------------------------------------------------------------------------
# The old index is:
#
# Index(['2020-01-02', '2020-01-03', '2020-01-06', '2020-01-07', '2020-01-08',
#    '2020-01-09', '2020-01-10', '2020-01-13', '2020-01-14', '2020-01-15'],
#   dtype='object')

# Replace the existing index with another with different values
ser.index = [0, 1, 2, 3, -4, 5, 6, 7, 8, 1000] # Note the -4 and 1000
print(ser.index)

# The new index is:
# Int64Index([0, 1, 2, 3, -4, 5, 6, 7, 8, 1000], dtype='int64')

# Lets see how the series looks like
print(ser)

# Output:
#  0       7.16
#  1       7.19
#  2       7.00
#  3       7.10
# -4       6.86
#  5       6.95
#  6       7.00
#  7       7.02
#  8       7.11
#  1000    7.04
# dtype: float64
```

2.2 Series attributes

The example above highlights an important aspect of indexes, namely that they can be changed by assignment. In this case, we assigned `ser.index` to a list object. What actually happened is that pandas created a new instance of `Index` based on the values in the list and then assigned it to the series.
Note that while the new index consists of integers only, these integers do *not* refer to the position of the element in the underlying array.

- For example, `ser[1000]` will return the element associated with the label "1000", not the 1000^th^ element of the series (which in this case, does not exist).

Also, observe that the index labels do not have to be in any order. In the example above, the fourth element in the index (which corresponds to the fourth element in the array) is `-4`, not `4`.
To be clear, `ser[-4]` will **not** return the fourth element from the end of the series. Instead, it corresponds to the element associated with the index label `-4` (in this case, 6.86).

- This is different than what you get from `prices[-4]`, which will return the fourth element from the end of the **list** `prices`:

2.2 Series attributes

```
# ------------------------------------------------------------------------
#   Selecting obs using the new index
# ------------------------------------------------------------------------

# This will return 7.04
x = ser[1000]
print(x)


# Compare the following cases:
# 1. This will return the element associated with the index label -4 (or 6.86)
print(ser[-4])

# 2. This will return the fourth element from the end of the **list** `prices` (or 7.00)
print(prices[-4])
```

2.2 Series attributes

You may have noticed that the new index is not an `Index` object, but an `Int64Index`.

- Pandas has many different types of index objects, which depend on the type of the labels we provide. In the example above, all the labels were integers. Pandas automatically creates a specialised index object that is performance optimised for integer labels.

In addition, both the `Index` and the `Int64Index` instances have an attribute called `dtype`.

- This specifies the data type of the labels we provided. In general, Pandas will choose the right data type (dtype) for you (but you can change that). At this point, you do not need to worry about these different types of `Index` objects. For most of our purposes, they behave exactly the same. We will discuss data types later.

2.2 Summary so far:

We have introduced two important objects provided by Pandas

1. `Series` : one-dimensional array-like objects containing:
    A. A sequence of values
    B. An `Index` object
2. `Index` : an object that stores the data labels and associate each label to a specific data value

Next, we will discuss another type of data structure that is fundamental to Pandas, the `DataFrame` .

2.2 Pandas DataFrames

A `DataFrame` in Pandas is analogous to a spreadsheet in Excel. We can think of a `DataFrame` as collection of `Series` and *two* indexes that allow us to index the data:

- `column` *index* : Associates each `Series` with a column label. We access this index through the dataframe attribute `columns`.
- *row index* : This index identifies elements in *any* `Series` inside a dataframe. Here is one way to think about it: `DataFrames` combine several `Series`. These `Series` all have the same row labels (i.e., the same `index`). Just like in a `Series`, we can access this common index using the attribute `index`.

You can create a dataframe using the `DataFrame` constructor: `pd.DataFrame(arguments)`.

- Like `Series`, `DataFrame` accepts many different kinds of input arguments. We will demonstrate several ways to build `DataFrames` in the notes.
- In fact, you may build `DataFrames` using combinations of arrays, lists (and other iterable objects), dictionaries, and other `DataFrames`. The **Pandas documentation** provides many examples to construct `DataFrames`.

2.2 Constructing DataFrames

Before continuing, you should open the companion code for this lesson in PyCharm:

Companion code for this lesson:
Please open the module `lec_pd_dataframes.py` in PyCharm. This module is inside the `lectures` package. The diagram below shows the location of this module in the `toolkit` project folder:

```
toolkit/                        <-- PyCharm project folder
|   ...
|___lectures/
|   |   lec_pd_dataframes.py    <-- Companion code for this
lesson
```

2.2 Constructing DataFrames

Let's start with a basic way to create a `DataFrame`.

- In this case, we will first create individual series and then combine them in a data frame. When you open the module `lec_pd_dataframes.py`, you will notice that we start by creating two series.
- The first is called `prc_ser` and contains the first ten (closing) prices for Qantas in 2020.
- The second series, `bday_ser`, is just a counter for each trading day during the same period. In other words, `bday_ser` takes the value of `1` on January 2, `2` on January 3, etc...

Note: the indexes in each series have exactly the same labels, constructed from `dates`.

2.2 Constructing DataFrames

```
""" lec_pd_dataframes.py

Companion codes for the lecture on Dataframes
"""

import pandas as pd

# ------------------------------------------------------------------------
#   The dates, close prices and Business (trading) day counter lists
# ------------------------------------------------------------------------


# ------------------------------------------------------------------------
#   Create two series
# ------------------------------------------------------------------------

# Series with prices
prc_ser = pd.Series(data=prices, index=dates)

# Series with trading day
bday_ser = pd.Series(data=bday, index=dates)
```

2.2 Constructing DataFrames

We can think of a data frame as **a *labelled* collection of series objects**. In other words, we can access each series (column) in a data frame by its column label. This works just like as in a `Dictionary`, in which every key is associated with a value.

- In fact, you can create a data frame by passing a dictionary of series to the constructor. When doing so, the dictionary key used in the `pd.DataFrame` constructor becomes the index key for the column in the data frame. Add the following to `lec_pd_dataframes.py`:

```
# ----------------------------------------------------------------------
#   Create a dataframe
# ----------------------------------------------------------------------
# Using the series we created above...

# Data Frame with close and Bday columns
df = pd.DataFrame({'Close': prc_ser, 'Bday': bday_ser})

print(df)
# Output:
#             Close  Bday
# 2020-01-02   7.16     1
# 2020-01-03   7.19     2
# 2020-01-06   7.00     3
# 2020-01-07   7.10     4
# 2020-01-08   6.86     5
# 2020-01-09   6.95     6
# 2020-01-10   7.00     7
# 2020-01-13   7.02     8
# 2020-01-14   7.11     9
# 2020-01-15   7.04    10
```

In the above code, we pass a `Dictionary` to the DataFrame constructor. The dictionary isn't assigned a separate variable, it's just inside the parentheses for the constructor as `{'Close': prc_ser, 'Bday': bday_ser}`.
Analysing the output above, a DataFrame looks just like a spreadsheet in Excel. We repeat some key differences just to reinforce their importance:

- In Excel, columns are always labelled `A`, `B`,... In a `DataFrame`, column labels can be anything we want. In the data frame above, these labels are the keys of the dictionary of series: `Close` and `Bday`.
- In Excel, we identify rows by `1`, `2`,... In a `DataFrame`, we identify the position of the data element in each series using a common index. In other words, all series in the data frame have the same index. In the DataFrame above, these labels are the dates.
- The column and row labels are *not* part of the data. The first data element in the data frame above is 7.16. It can be identified by referring to the column index `Close` and the row index `2020-01-02`.

2.2 Constructing DataFrames

> *REMARK: You may be wondering: what happens if the series we pass to the* `DataFrame` *constructor have different indexes? Pandas will try to combine these indexes, if possible. When that happens, the resulting series will be different than the original ones. Here is an example:*
> *~~~python other_dates = ['2020-01-06', '2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10', '2020-01-13', '2020-01-14', '2020-01-15', '2020-01-16', '2020-01-17',] other_bday = [3, 4, 5, 6, 7, 8, 9, 10, 11, 12,]*

2.2 Constructing DataFrames

```
new_bday_ser = pd.Series(data=other_bday, index=other_dates)
new_df = pd.DataFrame({'Close': prc_ser, 'Bday': new_bday_ser})
print(df)
# Returns:
#             Close  Bday
# 2020-01-02   7.16   NaN
# 2020-01-03   7.19   NaN
# 2020-01-06   7.00   3.0
# 2020-01-07   7.10   4.0
# 2020-01-08   6.86   5.0
# 2020-01-09   6.95   6.0
# 2020-01-10   7.00   7.0
# 2020-01-13   7.02   8.0
# 2020-01-14   7.11   9.0
# 2020-01-15   7.04  10.0
# 2020-01-16    NaN  11.0
# 2020-01-17    NaN  12.0
```

We will talk more about this later

2.2 The `column` and `index` attributes

Remember that `Series` objects have only one index. In contrast, `DataFrame` objects have two.

- The column index can be accessed through the `columns` attribute.
- The common row index for all series in a data frame can be accessed through `index`. Lets take a closer look at each of these objects.

We start with the column index:

```
# ------------------------------------------------------------------------
#   Accessing the indexes in a dataframe
# ------------------------------------------------------------------------
# The attribute `columns` returns the column index
print(df.columns)

# Output:
# Index(['Close', 'Bday'], dtype='object')

print(type(df.columns))

# Output: <class 'pandas.core.indexes.base.Index'>
```

So `df.columns` points to an `Index` object. This is the same type of object we saw above. However, this index is not used to identify individual values in a series. Instead, we use it to identify individual `Series` in a `DataFrame`.

2.2 The `column` and `index` attributes

For example, we can access the closing price series inside the `DataFrame` as follows:

```
col0 = df['Close']
print(col0)
# Output:
#   2020-01-02    7.16
#   2020-01-03    7.19
#   2020-01-06    7.00
#   2020-01-07    7.10
#   2020-01-08    6.86
#   2020-01-09    6.95
#   2020-01-10    7.00
#   2020-01-13    7.02
#   2020-01-14    7.11
#   2020-01-15    7.04
#   Name: Close, dtype: float64

print(type(col0))
# Output:
#   <class 'pandas.core.series.Series'>
```

2.2 The `column` and `index` attributes

As expected, `df['Close']` refers to a `Series`. Just like any series, you can access its index directly:

```
print(col0.index)
# Out:
# Index(['2020-01-02', '2020-01-03', '2020-01-06', '2020-01-07', '2020-01-08',
#        '2020-01-09', '2020-01-10', '2020-01-13', '2020-01-14', '2020-01-15'],
#       dtype='object')

print(type(col0.index))
# <class 'pandas.core.indexes.base.Index'>
```

It is important to remember that all series in a `DataFrame` have the same row index. In fact, we can access it directly from the `DataFrame` itself:

```
print(df.index)
# Out:
# Index(['2020-01-02', '2020-01-03', '2020-01-06', '2020-01-07', '2020-01-08',
#        '2020-01-09', '2020-01-10', '2020-01-13', '2020-01-14', '2020-01-15'],
#       dtype='object')

print(type(df.index))
# <class 'pandas.core.indexes.base.Index'>
```

2.2 The `column` and `index` attributes

Similarly to series indexes, the indices in a `DataFrame` are mutable, and we can change both `df.columns` and `df.index` in place by assignment.

- For example, you can set the columns and index labels to match the way the data would be represented in an Excel spreadsheet (i.e., letters for columns and numbers for rows). All we have to do is set new labels.

```
# ------------------------------------------------------------------------
#   Modifying columns and indexes
# ------------------------------------------------------------------------
# Modify the columns and indexes
df.columns = ['A', 'B']
df.index = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(df)

# Then revert back

df.columns = ['Close', 'Bday']
df.index = [
  '2020-01-02',
  '2020-01-03',
  '2020-01-06',
  '2020-01-07',
  '2020-01-08',
  '2020-01-09',
  '2020-01-10',
  '2020-01-13',
  '2020-01-14',
  '2020-01-15',
  ]
print(df)
```

2.2 Sorting Series and Dataframes

It is very good practice to ensure that your `Series` and `DataFrame` objects are sorted by the index. For a `DataFrame`, this means that the index (for rows) is sorted, not that the columns are sorted.

- For example, if your data is based on dates, like our Qantas stock data, your index should be sorted chronologically. If you are using an integer identifier for rows, the integers should be sorted in ascending order.

Sorting is necessary to ensure that we get the correct results when we select data. Selecting data is the foundation of all analysis. Hence, we need to have confidence that every time we select data, it is done correctly.

- We'll see the importance of this when we get to the "Accessing Data in Pandas: Indexing" lesson later.

2.2 Sorting Series and Dataframes

Given the importance of sorting, you may wonder why Pandas doesn't automatically sort all indices.

- Constantly sorting data can be time consuming. Hence, if you were to add data one row at a time, Pandas would constantly have to either check if your data was sorted or search for the correct location for the row on insertion.
- Instead, Pandas adds data to the end of a Series or DataFrame by default, allowing it to work quickly. The disadvantage of this is that you must remember to sort your data when you are ready to actually perform analysis.

Thus, we recommend the following approach.

- You should first assemble your Series and/or DataFrames. This includes all operations that add or remove rows, and relabel rows or columns, for example.
- Once you are done, sort your Series and DataFrame before doing your analysis.

2.2 Checking if a Series or DataFrame is Sorted

To check if a `Series` is sorted, you can use the `is_monotonic_increasing` attribute.

- This is `True` if the index is sorted in asecneding order and `False` if not. `Series` also includes an `is_monotonic_decreasing` attribute that allows us to see if the index is sorted in descending order, returning `True` when it is.

You may see references online to the `is_monotonic` attribute. This is equivalent to `is_monotonic_increasing`. However, the longer `is_monotonic_increasing` is recommended due to the fact that it is more explicit.

Go back to the `lec_pd_dataframes.py` lecture and uncomment the following statements:

```
# ------------------------------------------------------------------------
#   Sorting
# ------------------------------------------------------------------------
# Create a series with an unsorted index
new_ser = pd.Series(data=[1,3,2], index=['a', 'c', 'b'])

# This will return 'False'
print(new_ser.is_monotonic_increasing)
```

When working with a `DataFrame`, you need to first access the `Index` object attribute, `.index`. Then, you may simply use `is_monotonic_increasing` or `is_monotonic_decreasing` as you see fit. Thus, to check if a DataFrame `df` is sorted in increasing order, use `df.index.is_monotonic_increasing`.

2.2 Sorting a Series or DataFrame

Series have a method called `sort_index`, which will return a **copy** of the series with sorted indexes. The original Series remains unchanged! Only the *copy* is sorted. Thus, you will need to assign the new `Series` copy from `sort_index` to a new variable to use the sorted `Series`. Don't forget this!
You can also slice a series.

```
# Sort the series
sorted_ser = new_ser.sort_index()
print(new_ser)
print(sorted_ser)

# This slicing will only return the first two rows (not the entire series as before)
x = sorted_ser['a':'b'] # --> only first two rows
print(x)
# Out:
# a    1
# b    2
# dtype: int64

# `sorted_ser` is sorted so the following will return the intersection between
# the slice and the row labels
x = sorted_ser['b':'z']
print(x)
# Out:
# b    2
# c    3
# dtype: int64
```

2.2 Sorting a Series or DataFrame

Should you wish to sort the series itself, use the optional argument `sort_index(inplace = True)`. This does not create a new series. In fact, the return of the function call is `None`. Thus, you should **not** assign the result of the function call to a new variable, because using any new variable will attempt to use the returned `None`.

```
# Create a series with an unsorted index
ser_sort_inplace = pd.Series(data=[1,3,2], index=['a', 'c', 'b'])

# Sort the series. Note that we are not assigning this function call
# to a new variable.
ser_sort_inplace.sort_index(inplace = True)
print(ser_sort_inplace)
```

The default behaviour of `sort_index` is to set `inplace` to `False` when it is omitted from the function call. You can explicitly set `inplace = False` in the function call, making `sort_index` return a copy instead of sorted the `Series` in place.

2.2 Sorting a Series or DataFrame

Sorting `DataFrames` is also done via a `sort_index` method. Its behaviour is identical to `Series.sort_index`.

- By default, `DataFrame.sort_index` will return a **copy** of the DataFrame sorted by index. Like with a `Series`, you will need to assign the new DataFrame to a new variable to use the sorted DataFrame. You can modify the original DataFrame itself by setting `inplace = True` in the function call.
- We will demonstrate examples of sorting DataFrames in practice questions and later in the notes when we get to indexing.

2.3 Pandas and NumPy Arrays - Intro

Pandas relies on a very popular Python package called *NumPy*. NumPy stands for *Numerical Python* and was specifically designed for scientific computing. It features data containers and data types which are much better suited for scientific analysis than those available in the standard Python library. One nice feature of Pandas is that it provides a convenient interface to this powerful library.

> *REMARK: You do not have to install the NumPy package yourself. It is automatically installed as part of the Pandas installation process.*

The generally accepted way to import the NumPy is to assign it the `np` alias via

```
import numpy as np
```

Because Pandas is built on top of NumPy, it is important that we discuss a few features of the NumPy library that are particularly relevant for Pandas users. These features include:

- NumPy array object (`numpy.ndarray`)
- Specialised data types
- `NaN` (not-a-number) values

2.3 NumPy Arrays

A `numpy.ndarray` (which stands for *N-dimensional array*) is a homogeneous array with a predetermined number of items.

- By homogeneous, we mean that all elements in the array have the same data type. These two features, `fixed number of elements` and `common data type`, are part of what makes these arrays well suited for scientific analysis.

> *REMARK: Similar to a tuple, once you create a NumPy array, you cannot change its size (i.e., number of elements in the array). You can, however, reshape the array without the need to create a new one, as long as the number of items remain the same. For example, you can reshape a NumPy array containing 10 elements in a single dimension into an array containing 5 elements in two dimensions. For this course, we do not have to worry about this too much. The ability to reshape the array without creating a new one has performance implications when dealing with large amounts of data or when executing tasks that uses an array many times.*

2.3 NumPy Data Types

While a NumPy array must be homogeneous (it can only have one data type), NumPy itself is designed to accommodate a wide variety of data types.

- So, for example, NumPy has an array designed to handle only integers and a different array designed to handle only floating point numbers.

The data types permitted by NumPy are specific to the NumPy package and are not available in the standard Python library.

- For example, consider the `int` data type, which represents integers in the standard Python library. In NumPy, integers can be represented by different "integer" data types, depending on whether they are signed (may be positive and negative) or unsigned (only positive) and their maximum possible value.
- The flexibility to tailor the data type to your needs helps save memory, which can also help algorithms perform well when processing large amounts of data. Small (signed) integers can be stored as `numpy.int8` types (from -128 to 127), which consume less memory than `numpy.int16` data types (which is suitable for integers from -32,768 to 32,767). In turn, `numpy.int16` consume less memory than `numpy.int32` or `numpy.int64`, which are required to store larger integers.

In the vast majority of cases, you don't have to worry about picking up the right data type because NumPy and Pandas will do that for you.

2.3 NaN values

NumPy defines a special data type called `np.nan`, representing data that is "not a number".

- We will most often encounter this data type when dealing with missing data.
- Remember that NumPy arrays must be homogeneous, all data is of the same data type. The `np.nan` object is represented as a float, not a `None` type. In most cases, this is exactly what you need. However, in some cases representing "not a number" as a float can be problematic, notably when we want to include missing numbers in an array of integers.
    - Pandas has a solution to this problem, which we will discuss next.

2.3 Pandas Arrays, Dtypes, and NaN

Now that we've covered the basics of NumPy arrays, let's see NumPy arrays are integrated into Pandas.

Companion code for this lesson:
Please open the module `lec_pd_numpy.py` in PyCharm. This module is inside the `lectures` package. The diagram below shows the location of this module in the `toolkit` project folder:

```
toolkit/                 <-- PyCharm project folder
|   ...
|___lectures/
|   |   lec_pd_numpy.py   <-- Companion code for this lesson
```

2.3 Pandas Arrays, Dtypes, and NaN

The module `lec_pd_numpy` will define the same dataframe we used in our previous example:

```
""" lec_pd_numpy.py

Companion codes for the lecture on Numpy
"""

import numpy as np
import pandas as pd

# ---------------------------------------------------------------------------
#   The dates, prices, and bday lists
# ---------------------------------------------------------------------------


# ---------------------------------------------------------------------------
#   Create two series
# ---------------------------------------------------------------------------

# Series with prices
prc_ser = pd.Series(data=prices, index=dates)

# Series with trading day
bday_ser = pd.Series(data=bday, index=dates)


# ---------------------------------------------------------------------------
#   Create a dataframe
# ---------------------------------------------------------------------------
# Data Frame with close and Bday columns
df = pd.DataFrame({'Close': prc_ser, 'Bday': bday_ser})

print(df)

# Out:
#              Close  Bday
#  2020-01-02   7.16     1
#  2020-01-03   7.19     2
#  2020-01-06   7.00     3
#  2020-01-07   7.10     4
#  2020-01-08   6.86     5
```

```
#  2020-01-09   6.95     6
#  2020-01-10   7.00     7
#  2020-01-13   7.02     8
#  2020-01-14   7.11     9
#  2020-01-15   7.04    10
```

2.3 Pandas Arrays, Dtypes, and NaN

`DataFrames` have a convenient method called `info`, which will print a concise summary of the object:

```
df.info()

# Returns something like this:
#  <class 'pandas.core.frame.DataFrame'>
#  Index: 10 entries, 2020-01-02 to 2020-01-15
#  Data columns (total 2 columns):
#   #   Column  Non-Null Count  Dtype
#  ---  ------  --------------  -----
#   0   Close   10 non-null     float64
#   1   Bday    10 non-null     int64
#  dtypes: float64(1), int64(1)
#  memory usage: 240.0+ bytes
```

2.3 Pandas Arrays, Dtypes, and NaN

Remember that a `DataFrame` is a collection of `Series` objects.

- As we can see, the dataframe above contains two `Series` (columns). Both series share the same index (which is also the index of the `DataFrame`). Each series, however, contains its own data, stored in a different array-like container. These containers are `PandasArray` objects, which are extensions of NumPy arrays. For most purposes, you can think of them as `numpy.ndarray` objects.

The data types for each series are provided under the `Dtype` column.

- In our example, Pandas has correctly inferred that the closing prices are floating point numbers and the business (trading) days are integers; "Close" is stored as a `np.float64` data type, whereas "Bday" is stored as `np.int64` data type.

2.3 Pandas Arrays, Dtypes, and NaN

In general, Pandas will use the data type from `Numpy` for the individual columns in a DataFrame. As a result of this decision, Pandas will generally also use the corresponding NumPy array for each data type.
When you create a series or a data frame, Pandas figures out the most efficient data type for you. If the data elements cannot be "cast" to one of these high performing data types, the data type will be set to `object`.

- You can think of it as a generic data type that is accepts data of mixed types. For example, trying to construct a column that has both numeric and textual data will result in a Pandas column with type *object*. The `object` type will also be used for any series containing strings.

> *object serves as a catch-all data type. Always check your DataFrames to ensure that the data type matches your expectation. Generally, you should view an object dtype as a sign of an inconsistency in any numerical data and try to resolve any underlying data issues before proceeding with your analysis.*

2.3 Pandas Arrays, Dtypes, and NaN

Pandas supports the following NumPy data types (*dtype*). The **source table** provides additional details:

Data Type	Description
bool_	Boolean (true or false)
int8	Byte (integers between –128 to 127)
int16	Integer (integers between –32768 to 32767)
int32	Integer (integers between –2147483648 to 2147483647)
int64	Integer (integers between –9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (positive integers between 0 to 255)
uint16	Unsigned integer (positive integers between 0 to 65535)
uint32	Unsigned integer (positive integers between 0 to 4294967295)
uint64	Unsigned integer (positive integers between 0 to 18446744073709551615)
float_	Shorthand for float64
float16	Half precision float: sign bit, 5-bit exponent, 10-bit mantissa
float32	Single precision float: sign bit, 8-bit exponent, 23-bit mantissa
float64	Double precision float: sign bit, 11-bit exponent, 52-bit mantissa
complex_	Shorthand for complex128
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128	Complex number, represented by two 64-bit floats (real and imaginary components)

2.3 Pandas Arrays, Dtypes, and NaN

For the floating point and complex numbers, you should view higher bit counts for the exponent and mantissa as supporting greater accuracy.

- The more bits that are used to represent a number, the more closely it is stored by the computer and the more accurate the resulting calculations.
- Most Finance analysis should work with the lower end of accuracy, such as *float16*, because we are often interested in prices down to cents or parts of cents.

Please note that we have omitted a few types that, in general, should not be used. These include types that may change depending on what kind of computer you are using. We believe it is better to be explicit so as to ensure that the calculations proceed as you wish.

Recall that the attribute `.array` will return the contents of a Pandas `Series` as a `PandasArray`. In the example below, you can see that Pandas will print out some information about the array for you. If you ask for the array type, Python will show you where in the Pandas library the `PandasArray` class is defined.

- Go back to the `lec_pd_numpy.py` file, uncomment the following statements, and run the module again:

2.3 Pandas Arrays, Dtypes, and NaN

```
# Get the series containing "Close" prices
ser = df['Close']

# Get the underlying data array
print(ser.array)

# Returns:
#  <PandasArray>
#  [7.16, 7.19, 7.0, 7.1, 6.86, 6.95, 7.0, 7.02, 7.11, 7.04]
#  Length: 10, dtype: float64

print(type(ser.array))

# Output:
#  <class 'pandas.core.arrays.numpy_.PandasArray'>
```

You can convert the data inside a series to a NumPy array using the attribute `.values`:

```
print(ser.values)
# Out:
#  array([7.16, 7.19, 7.  , 7.1 , 6.86, 6.95, 7., 7.02, 7.11, 7.04])

print(type(ser.values))  # --> <class 'numpy.ndarray'>
```

2.3 Working with missing data in Pandas

Now lets see how Pandas deals with missing data. Remember that the Numpy library defines a special "not a number" float object, `np.nan`.
Suppose we want to add a new row to the `df` data frame. We will create an empty row for the date '3000-01-01'. We can do using the `.loc` method.

- We will discuss this method in the next lesson. For now, just note that it will create a new row in the data frame.

To make sure we do not modify the original `df` dataframe, we will first create a copy called `df_nan`, and then add a new empty row. Open the `lec_pd_numpy.py` module and navigate to the following section:

2.3 Working with missing data in Pandas

```
# ------------------------------------------------ ----------------------------
#   Working with missing data
# ------------------------------------------------ ----------------------------

# Create a copy
df_nan = df.copy()

# Add an emtpy row to the `df_nan` dataframe
df_nan.loc['3000-01-01'] = [np.nan, np.nan]
print(df_nan)

# Returns:
#             Close  Bday
# 2020-01-02   7.16   1.0
# 2020-01-03   7.19   2.0
# 2020-01-06   7.00   3.0
# 2020-01-07   7.10   4.0
# 2020-01-08   6.86   5.0
# 2020-01-09   6.95   6.0
# 2020-01-10   7.00   7.0
# 2020-01-13   7.02   8.0
# 2020-01-14   7.11   9.0
# 2020-01-15   7.04  10.0
# 3000-01-01    NaN   NaN
```

2.3 Working with missing data in Pandas

This seems to do the trick. However, remember that the series `Bday` consists of integers, whereas `Close` contains floats. Will these data types be preserved in the new dataframe?

- Unfortunately no. This is because when we add the float `np.nan` to the end of the `Bday` series, Pandas will automatically re-cast the series to `float`. To see that do:

```
print("\nThis is the `df` dataframe:")
print(df.info())

# Returns:
#   This is the `df` dataframe:
#   <class 'pandas.core.frame.DataFrame'>
#   Index: 10 entries, 2020-01-02 to 2020-01-15
#   Data columns (total 2 columns):
#    #   Column  Non-Null Count  Dtype
#   ---  ------  --------------  -----
#    0   Close   10 non-null     float64
#    1   Bday    10 non-null     int64
#   dtypes: float64(1), int64(1)


print("\nThis is the `df_nan` dataframe:")
print(df_nan.info())

# Returns:
#   This is the `df_nan` dataframe:
#   <class 'pandas.core.frame.DataFrame'>
#   Index: 11 entries, 2020-01-02 to 3000-01-01
#   Data columns (total 2 columns):
#    #   Column  Non-Null Count  Dtype
#   ---  ------  --------------  -----
#    0   Close   10 non-null     float64
#    1   Bday    10 non-null     float64
#   dtypes: float64(2)
```

2.3 Working with missing data in Pandas

Again, because `np.nan` is a float, it forces an array of integers to become floating point. This may become a problem as we don't want integers represented by floating point numbers which make equality tests unreliable. Fortunately, Pandas allows us to preserve the original data type while supporting missing or non-numerical values. The best way is to let Pandas perform any necessary conversion for us automatically using the series or dataframe method `convert_dtypes`:

```
# Convert dtypes
df_new = df_nan.convert_dtypes()
print(df_new.info())

# Returns
#   Data columns (total 2 columns):
#    #   Column  Non-Null Count  Dtype
#   ---  ------  --------------  -----
#    0   Close   10 non-null     float64
#    1   Bday    10 non-null     Int64
#   dtypes: Int64(1), float64(1)

print(df_new.loc['3000-01-01'])
# Returns:
#   Close    <NA>
#   Bday     <NA>
```

As you can see, Pandas converted the Numpy `np.nan` in the `Bday` series to an integer version of "Not a Number", represented by "" in the series above. This `<NA>` is Pandas's own version of a "Not a Number", `pandas.NA`. Developers are still working on this feature in Pandas. It works exactly as `np.nan`, except that it can be used with `float`, `integer`, and `object` data types.

2.3 The df.info() method revisited

The `info` method is available to any Pandas data frame. When called, it produces a table with information about the data frame.
In general, `df.info()` will generate a table similar to the one below:

```
<class 'pandas.core.frame.DataFrame'>
<Index Type>: <nobs>, <first label> to <last label>
Data  columns (total of <ncols> columns):
#            Column      Non-Null Count        Dtype
---          --------    --------------        -----
<col pos>    <col name>  <not null> non-null   <col dtype>
...
```

where

- `<Index type>`: The type of the index (e.g., `RangeIndex`, `DatetimeIndex`)
- `<nobs>:` The number of observations (index labels)
- `<first label>`: The index label of the first observation
- `<last label>`: The index label of the last observation
- `<col pos>`: The position of the column in the data frame (starting at 0)
- `<col name>`: The column label
- `<not null>`: The number of valid observations (not null)
- `<col dtype>`: The data type of the column/series

2.3 The df.info() method revisited

Examples:

```
import pandas as pd

data = {
      'col_a': [1, 2, 3],
      'col_b': [10.0, None, 13.0],
      }

# First a data frame without a user-defined index
df0 = pd.DataFrame(data)
print('\nprint(df0) -->')
print(df0)
print('\ndf0.info() --> ')
df0.info()

# A data frame with a strings as index labels
idx = ['2020-01-01', '2020-01-02', '2020-01-03']
df1 = pd.DataFrame(data, index=idx)
print('\nprint(df1) -->')
print(df1)
print('\ndf1.info() --> ')
df1.info()

# A data frame with a datetime objs as index labels
idx_dt = pd.to_datetime(idx)
df2 = pd.DataFrame(data, index=idx_dt)
print('\nprint(df2) -->')
print(df2)
print('\ndf2.info() --> ')
df2.info()
```