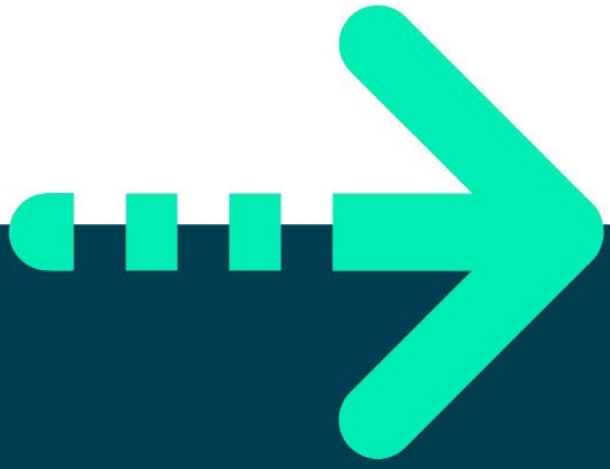
Introduction to Data Science for Data Professionals

Exercise Guide



Exercise 2: Introduction to Python and IDEs

Objective

In this guided task, you'll use Jupyter to:

- Create a Python application.
- Perform arithmetic operations.
- Input values using keyboard and display information.
- Use casting to convert strings to integers and floats.

Create a new notebook

1. Experiment with the keyboard shortcuts in Command mode. You might want to try:
   - Adding cells above and below.
   - Deleting cells
   - Changing from markdown to code.
   - Verifying, i.e., entering some text or code and running checks.

Display Hello World!

1. Type and run the following code in a cell:

```
print('Hello World!')
```

Display a message using variables

1. Create two variables to hold someone's name and age. Type:

```
username='Bob'
age=32
print(username,'is',age,'years old')
```

2. Modify your code to use the '+' character to add strings:

```
username='Bob'
age=32
print(username+'is'+age+'years old')
```

What's wrong? Look at the error. Can you correct it?

Does the output look right or are the strings stuck together? That's what + and strings does!

3. Modify the print statement by inserting three spaces in the strings as follows:

```
print(username,' is '+age+' years old')
```

Get user input

1. Write the following code to get the username and age by using the keyboard.

```
username=input('Please enter your name')
age = input('Please enter your age')
print(username,'is',age,'years old')
```

Arithmetic operations

1. A box contains 50 crayons, which have to be distributed to 6 children.
   Each child has to get the same number of whole crayons (no breaking).
   Calculate the number of crayons each child will get.

Casting variables

1. Capture user input to get the length of the first side of a rectangle.
   Use a suitable variable name such as **length**.
   You must cast (convert) the text you input to an integer type (int).

2. Input the length of the second side of the rectangle.
   Use a suitable variable name such as **width**.
   Again, cast the input text to an integer type (int).

3. Calculate and display the perimeter of the rectangle.

Extension activities (for Python pros)

1. Write a piece of code which calculates the amount of change needed to be given from an input amount in £.

   To start you off, we've given you a list of denominations of UK currency.

```
denoms = [50, 20, 10, 5, 2, 1, 0.5, 0.2, 0.1, 0.05, 0.02, 0.01]
```

Exercise 2: Introduction to Python and IDEs

Objective

In this guided task, you'll use Jupyter to:

- Create a Python application.
- Perform arithmetic operations.
- Input values using keyboard and display information.
- Use casting to convert strings to integers and floats.

Create a new notebook

1. Experiment with the keyboard shortcuts in Command mode. You might want to try:
   - Adding cells above and below.
   - Deleting cells
   - Changing from markdown to code.
   - Verifying, i.e., entering some text or code and running checks.

Display Hello World!

1. Type and run the following code in a cell:

   `print('Hello World!')`

```
print('Hello World!')
```

```
Hello World!
```

Display a message using variables

1. Create two variables to hold someone's name and age. Type:

   `username='Bob'`

   `age=32`

   `print(username,'is',age,'years old')`

```
username='Bob'
age=32
print(username,'is',age,'years old')
```

```
Bob is 32 years old
```

2. Modify your code to use the '+' character to add strings:

```
username='Bob'
age=32
print(username+'is'+age+'years old')
```

```
username='Bob'
age=32
print(username+'is'+age+'years old')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[3], line 3
      1 username='Bob'
      2 age=32
----> 3 print(username+'is'+age+'years old')

TypeError: can only concatenate str (not "int") to str
```

What's wrong? Look at the error. Can you correct it?

```
username='Bob'
age=32
print(username+'is'+str(age)+'years old')
```

```
Bobis32years old
```

Does the output look right or are the strings stuck together? That's what + and strings does!

3. Modify the print statement by inserting three spaces in the strings as follows:

```
print(username,' is '+age+' years old')
```

```
age=32
print(username+' is '+str(age)+' years old')
```

```
Bob is 32 years old
```

Get user input

1. Write the following code to get the username and age by using the keyboard.

```
username=input('Please enter your name')
age = input('Please enter your age')
print(username,'is',age,'years old')
```

```
username=input('Please enter your name')
age = input('Please enter your age')
print(username,'is',age,'years old')
```

```
Please enter your name Thomas
Please enter your age 30
Thomas is 30 years old
```

Arithmetic operations

1. A box contains 50 crayons, which have to be distributed to 6 children.
   Each child has to get the same number of whole crayons (no breaking).
   Calculate the number of crayons each child will get.

```
crayons = 50
children = 6

crayons // children
```

```
8
```

Casting variables

1. Capture user input to get the length of the first side of a rectangle.
   Use a suitable variable name such as **length**.
   You must cast (convert) the text you input to an integer type (int).

```
length = int(input("What is the length of the first side? "))
```

```
What is the length of the first side?  10
```

2. Input the length of the second side of the rectangle.
   Use a suitable variable name such as **width**.
   Again, cast the input text to an integer type (int).

```
width = int(input("What is the length of the second side? "))
```

```
What is the length of the second side?  20
```

3. Calculate and display the perimeter of the rectangle.

```
print(f'The rectangle with length {length} and width {width} has perimeter {2 * (length + width)}')
```

```
The rectangle with length 10 and width 20 has perimeter 60
```

Extension activities (for Python pros)

1. Write a piece of code which calculates the amount of change needed to be given from an input amount in £.

   To start you off, we've given you a list of denominations of UK currency.

```
denoms = [50, 20, 10, 5, 2, 1, 0.5, 0.2, 0.1, 0.05, 0.02, 0.01]
amount = float(input("What do you need change from? £"))

numbers = []
for denom in denoms:
    numbers.append(int(amount // denom))
    if amount % denom == 0:
        break
    else:
        amount = round(amount % denom, 2) # There are better types for this

for num, denom in zip(numbers, denoms):
    print(f"{num} x £{denom}")
```

```
What do you need change from? £ 33.60
0 x £50
1 x £20
1 x £10
0 x £5
1 x £2
1 x £1
1 x £0.5
0 x £0.2
1 x £0.1
```

Exercise 3: Data structures, flow control, functions, & basic types

Objective

In this guided task, you will:

- Perform various operations on lists and strings.
- Perform operations related to tuples, dictionaries, and sets.
- Use conditional if...elif control flow statements in Python.
- Work with iteration to control flow, using while statements in Python.
- Work with iteration to control flow, using for loops in Python.
- Define and call functions.
- Perform operations related to reading, processing and writing files.

Data structures

Lists

```
ages =
[12,18,33,84,45,67,12,82,95,16,10,23,43,29,40,34,30,16,44,69,70,74,38,
65,36,83,50,11,79,64,78,37,3,8,68,22,4,60,33,82,45,23,5,18,28,99,17,81
,14,88,50,19,59,7,44,93,35,72,25,63,11,69,11,76,10,60,30,14,21,82,47,6
,21,88,46,78,92,48,36,28,51]
```

1. Record the length of the ages list in a variable. Display the length.
2. Display the first element of the ages list.
3. Display the last element of the ages list.
4. Find the ages from the third to the last.
5. Show the first 10 ages backwards.
6. Spot the missing value in the list of odd numbers and insert it. *Hint: .insert()*

```
# a list of odd numbers
odd = [1,3,5,7,9,11,15,17,19,21]
```

Strings

1. Spell the greeting backwards. *Hint: [::]*

```
# a greeting
greeting = 'Hello'
```

2. Spell the name of the UK observing which letters should be capital and which lowercase.

```
UK = 'UNITED KINGDOM OF GREAT BRITAIN AND NORTHERN IRELAND'
```

3. Find the position of the first occurrence of the word 'forever'.

```
# from "Fare thee well", George Gordon Byron
byron = 'Fare thee well and if forever still forever fare thee
well'
```

Tuples

1. Create a tuple with your favourite desserts (at least 3 of them!).
2. Print the number of your favourite desserts.
3. Print the second of the desserts.
4. Print the last of the desserts.

```
# And no, this will not work. Why? Try to mend it and also try
another way
print(desserts[len(desserts)])
```

5. Print the first two desserts.
6. Check if pancakes are included in the desserts.

Let's get more sophisticated... (try these after the dictionaries exercises)

7. Write code to ask a customer for their dessert preference and respond depending on its availability. *Hint: You need ifs and elses*

8. Once a tuple is created, you cannot change its values. Tuples are immutable (unchangeable). But there is a workaround! You can convert the tuple into a list, change the list, and convert the list back into a tuple.

   Change the second dessert to something else.

Dictionaries

Consider the following dictionary:

```
books = {
  "title": "Hamlet",
  "author": "William Shakespeare",
  "language": "English"
}
```

1. Print the title value.
2. Get a list of the keys of the dictionary.
3. Get a list of the values of the dictionary.
4. Get a list of the key-value items.
5. Change the title to 'King Lear' and check that the dictionary is updated.

Selection

The general Python syntax for a simple if statement is:

```
if condition:
    indentedStatementBlock
```

1. Create an **IF statement** to see if the person's age is equal to 18 or over. Display: `You are in category A.`
2. Create an **IF statement** to see if the person's age is equal to 16 or over. Display: `You are in category B.`
3. Create an **IF statement** to see if the person is under 16 years of age. Display: `You are in category C.`
4. Go back, run the first bit of code, and enter 19 for age.

As you see, there are too many confusing messages!

Simple IF statements work fine, but not in a chain of IF statements such as these.

For a chain, we need to use an **if...elif** statement.

5. Create an **if...elif** statement to examine the age in one statement. Follow this pattern:

```
if person is 18 and over:
        display message
```

```
elif person is 16 and over:
      display message
else:
      display message
```

Note: You must start with the highest age value first.

6. Save and rerun your code using different values for age.

While Loops

The syntax of a while loop in Python programming language is:

```
while expression:
   statement(s)
```

1. Create a variable with the value 1.

   Write a while loop that starts at 1 and ends at 100.

   Within the while loop calculate and display each number and its square.

   End the loop if that square is bigger than 2000.

Investment

1. Calculate how many years it will take an initial investment of £100 to grow to a target value of £1000 if the interest rate is 10%.

   Tip: Don't start writing code until you've got a plan of action!

   Make your calculation easier to understand by defining the following variables:

   - Initial investment
   - Target value
   - Interest rate

For Loops

Average

1. Create a list of numbers using a for loop and compute their average.

Combinations

1. You are given a list of sizes and a list of garments. Display all combinations of a size and a garment (you will need more than one for loop for this!).

   ```
   sizes = [‘Large’, ‘Medium’, ‘Small’] garments = [‘Shirt’, ‘Jacket’, ‘Trousers’]
   ```

Functions

1. Write a function that takes a string as an argument and returns the number of capital letters in the string.

   Hint: `'foo'.upper() returns 'FOO'.`

2. Update the function so that when nothing is passed to the function, the argument defaults to an empty string.

   Check that the output of this updated function is 0 when nothing is passed to it.

File handling

1. Read file **`data.txt`**
2. Calculate the sum, minimum, maximum, and the average of the values in the file. *Hint: All data read in from the file will be of string type*
3. Output the results to a newly created file.
4. Close the file.

Append output file

1. Adults only! Ask a customer their age. If they are at least 18 years old, print ‘`Welcome’`.
   Otherwise, print ‘`Please come back in X years`’, where X is the remaining time until they turn 18.

   In addition to the screen output, send the output to a text file, adding every response to the file.

Extension activities (for Python pros)

Exercise 1: Sets

1. Create a set containing fruit.
2. Add another fruit to the set and print it.
3. Loop through the set of fruit and print each element.
4. Consider the following sets.

```
s1 = {'A', 'B', 'C', 'D', 'E'}
s2 = {'D', 'E', 'F', 'G'}
```

5. Find and print set S3, which contains all elements belonging to either S1 and S2.
6. Find and print set S3, which contains the elements belonging to both S1 and S2.
7. Find and print set S3, which contains the elements that are in S1 but not S2.
8. Find and print set S3, which contains the elements that are in only one of S1 and S2 but not the other.

Exercise 2: Functions

Write a function that takes two sequences `seq_a` and `seq_b` as arguments and returns `True` if every element in `seq_a` is also an element of `seq_b`, else `False`.

- By 'sequence' we mean a list, a tuple, or a string.
- Do the exercise without using sets and set methods.

Exercise 3

Using list comprehension syntax, simplify the loop in the following code.

```
import numpy as np

n = 100
ϵ_values = []
for i in range(n):
```

```
    e = np.random.randn()
    ϵ_values.append(e)
```

Exercise 4

Using the dictionary `christmas_sequence` print out each verse of the The Twelve Days of Christmas.

For example, the third verse will be: *On the third day of Christmas my true love sent to me three french hens, two turtle doves and a partridge in a pear tree.*

Hint: You will probably need nested loops.

```
christmas_sequence = {"first": "a partridge in a pear tree",
                      "second": "two turtle doves",
                      "third": "three french hens",
                      "fourth": "four calling birds",
                      "fith": "five gold rings",
                      "sixth": "six geese a-laying",
                      "seventh": "seven swans a-swimming",
                      "eigth": "eight maids a-milking",
                      "nineth": "nine ladies dancing",
                      "thenth": "ten lords a-leaping",
                      "eleventh": "eleven pipers piping",
                      "twelth": "twelve drummers drumming"}
```

Exercise 5

The function **`dir`** allows you to look at the methods and attributes associated with a python object. Pick one or more of the objects we have looked at.

There are two main types of methods and attributes returned. What are they? What is the value in this approach?

```
dir(1)
```

Exercise 6

As you might have realised, variables are accessible between code cells. They are in the same *namespace*. We can look at what is in that namespace using the `globals` and `locals` functions.

- What are the datatypes of the `globals` and `locals` function outputs?

- There are lots of variables listed besides the ones you have created. Many of these are specific to this interface. Can you identify their purpose?

Exercise 3: Data structures, flow control, functions, & basic types

Objective

In this guided task, you will:

- Perform various operations on lists and strings.
- Perform operations related to tuples, dictionaries, and sets.
- Use conditional if...elif control flow statements in Python.
- Work with iteration to control flow, using while statements in Python.
- Work with iteration to control flow, using for loops in Python.
- Define and call functions.
- Perform operations related to reading, processing and writing files.

Data structures

Lists

```
ages =
[12,18,33,84,45,67,12,82,95,16,10,23,43,29,40,34,30,16,44,69,70,74,38,
65,36,83,50,11,79,64,78,37,3,8,68,22,4,60,33,82,45,23,5,18,28,99,17,81
,14,88,50,19,59,7,44,93,35,72,25,63,11,69,11,76,10,60,30,14,21,82,47,6
,21,88,46,78,92,48,36,28,51]
```

1. Record the length of the ages list in a variable. Display the length.

```
ages = [12,18,33,84,45,67,12,82,95,16,10,23,43,29,40,34,30,16,44,69,70,74,38,65,36,83,50,11,79,64,78,37,3,8,
        68,22,4,60,33,82,45,23,5,18,28,99,17,81,14,88,50,19,59,7,44,93,35,72,25,63,11,69,11,76,10,60,30,14,
        21,82,47,6,21,88,46,78,92,48,36,28,51]
```

```
age_length = len(ages)
print(age_length)
```

```
81
```

2. Display the first element of the ages list.

```
print(ages[0])
```

```
12
```

3. Display the last element of the ages list.

```
print(ages[-1])
```

```
51
```

4. Find the ages from the third to the last.

```
print(ages[3:])
```

```
[84, 45, 67, 12, 82, 95, 16, 10, 23, 43, 29, 40, 34, 30, 16, 44, 69, 70, 74, 38, 65, 36, 83, 50, 11, 79, 64, 78, 3
7, 3, 8, 68, 22, 4, 60, 33, 82, 45, 23, 5, 18, 28, 99, 17, 81, 14, 88, 50, 19, 59, 7, 44, 93, 35, 72, 25, 63, 11,
69, 11, 76, 10, 60, 30, 14, 21, 82, 47, 6, 21, 88, 46, 78, 92, 48, 36, 28, 51]
```

5. Show the first 10 ages backwards.

```
print(ages[10::-1])
```

```
[10, 16, 95, 82, 12, 67, 45, 84, 33, 18, 12]
```

6. Spot the missing value in the list of odd numbers and insert it. *Hint: .insert()*

```
# a list of odd numbers
odd = [1,3,5,7,9,11,15,17,19,21]
```

```
odd.insert(6, 13)
odd
```

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

Strings

1. Spell the greeting backwards. *Hint: [::]*

```
# a greeting
greeting = 'Hello'
```

```
greeting[::-1]
```

```
'olleH'
```

2. Spell the name of the UK observing which letters should be capital and which lowercase.

```
UK = 'UNITED KINGDOM OF GREAT BRITAIN AND NORTHERN IRELAND'
```

```
UK.title()
```

```
'United Kingdom Of Great Britain And Northern Ireland'
```

3. Find the position of the first occurrence of the word ‘forever’.

```
# from "Fare thee well", George Gordon Byron
byron = 'Fare thee well and if forever still forever fare thee
well'
```

```
byron.find('forever')
```

```
22
```

Tuples

1. Create a tuple with your favourite desserts (at least 3 of them!).

```
desserts = 'sticky toffee pudding', 'affogato', 'syrup sponge'
```

2. Print the number of your favourite desserts.

```
print(len(desserts))
```

```
3
```

3. Print the second of the desserts.

```
desserts[1]
```

```
'affogato'
```

4. Print the last of the desserts.

```
# And no, this will not work. Why? Try to mend it and also try
another way
print(desserts[len(desserts)])
```

```
# And no, this will not work. Why? Try to mend it and also try another way
#print(desserts[len(desserts)])
print(desserts[len(desserts) - 1])
```

```
syrup sponge
```

```
# much simpler
desserts[-1]
```

```
'syrup sponge'
```

5. Print the first two desserts.

```
desserts[:2]
```

```
('sticky toffee pudding', 'affogato')
```

6. Check if pancakes are included in the desserts.

```
'pancakes' in desserts
```

```
False
```

Let's get more sophisticated... (try these after the dictionaries exercises)

7. Write code to ask a customer for their dessert preference and respond depending on its availability. *Hint: You need ifs and elses*

```
user_dessert = input('What dessert do you want? ')

user_dessert in desserts
```

```
What dessert do you want?  cheese
False
```

8. Once a tuple is created, you cannot change its values. Tuples are immutable (unchangeable). But there is a workaround! You can convert the tuple into a list, change the list, and convert the list back into a tuple.

   Change the second dessert to something else.

```
desserts = list(desserts)
desserts[1] = 'creme brulee'
desserts = tuple(desserts)
```

Dictionaries

Consider the following dictionary:

```
books = {
  "title": "Hamlet",
  "author": "William Shakespeare",
  "language": "English"
}
```

1. Print the title value.

```
books['title']
```

```
'Hamlet'
```

2. Get a list of the keys of the dictionary.

```
books.keys()
```

```
dict_keys(['title', 'author', 'language'])
```

3. Get a list of the values of the dictionary.

```
books.values()
```

```
dict_values(['Hamlet', 'William Shakespeare', 'English'])
```

4. Get a list of the key-value items.

```
books.items()
```

```
dict_items([('title', 'Hamlet'), ('author', 'William Shakespeare'), ('language', 'English')])
```

5. Change the title to 'King Lear' and check that the dictionary is updated.

```
books['title'] = 'King Lear'
```

```
books
```

```
{'title': 'King Lear', 'author': 'William Shakespeare', 'language': 'English'}
```

Selection

The general Python syntax for a simple if statement is:

```
if condition:
    indentedStatementBlock
```

1. Create an **IF statement** to see if the person's age is equal to 18 or over. Display: `You are in category A.`

```
age = 40

if age > 18:
    print('You are in category A')
```

```
You are in category A
```

2. Create an **IF statement** to see if the person's age is equal to 16 or over. Display: `You are in category B.`

```
age = 40

if age > 16:
    print('You are in category B')
```

```
You are in category B
```

3. Create an **IF statement** to see if the person is under 16 years of age. Display: `You are in category C.`

```
age = 40

if age < 16:
    print('You are in category C')
```

4. Go back, run the first bit of code, and enter 19 for age.

As you see, there are too many confusing messages!

Simple IF statements work fine, but not in a chain of IF statements such as these.

For a chain, we need to use an **if...elif** statement.

5. Create an **if...elif** statement to examine the age in one statement. Follow this pattern:

```
if person is 18 and over:
      display message
elif person is 16 and over:
      display message
else:
      display message
```

Note: You must start with the highest age value first.

```
age = 19

if age > 18:
    print('You are in category A')
elif age > 16:
    print('You are in category B')
else:
    print('You are in category C')
```

```
You are in category A
```

6. Save and rerun your code using different values for age.

While Loops

The syntax of a while loop in Python programming language is:

```
while expression:
    statement(s)
```

1. Create a variable i with the value 1.

   ```
   i = 1
   ```

   Write a while loop that starts at 1 and ends at 100.

   Within the while loop calculate and display each number and its square.

   End the loop if that square is bigger than 2000.

   ```
   i = 1
   while i < 100:
       square = i ** 2
       if square > 2000:
           break
       print(square)
       i+=1
   ```

Investment

1. Calculate how many years it will take an initial investment of £100 to grow to a target value of £1000 if the interest rate is 10%.

   Tip: Don't start writing code until you've got a plan of action!

   Make your calculation easier to understand by defining the following variables:

- Initial investment
- Target value
- Interest rate

```
start = float(input("Initial investment: "))
target = float(input("Target value: "))
interest = int(input("Interest rate [%]: "))/100
```

```
year = 0
current = start
while current < target:
    year += 1
    current *= (1+interest)
    print("year",year,"value",round(current,2))
```

For Loops

Average

1. Create a list of numbers using a for loop and compute their average.

```
L = [1,5,3,8,4,7,9,2]

sum = 0
for n in L:
    sum += n
average = sum / len(L)
average
```

Combinations

1. You are given a list of sizes and a list of garments. Display all combinations of a size and a garment (you will need more than one for loop for this!).

   `sizes = [‘Large’, ‘Medium’, ‘Small’] garments = [‘Shirt’, ‘Jacket’, ‘Trousers’]`

```
sizes = ['Large', 'Medium', 'Small']
garments = ['Shirt', 'Jacket', 'Trousers']

for i in sizes:
    for j in garments:
        print(i,j)
```

Functions

1. Write a function that takes a string as an argument and returns the string in uppercase.

   Hint: `'foo'.upper() returns 'FOO'.`

```
def make_upper(text):
    return text.upper()
```

2. Update the function so that when nothing is passed to the function, the argument defaults to an empty string.

   Check that the output of this updated function is 0 when nothing is passed to it.

```
def make_upper(text=""):
    return text.upper()
```

File handling

1. Read file **`data.txt`**

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

2. Calculate the sum, minimum, maximum, and the average of the values in the file. *Hint: All data read in from the file will be of string type*

```
data = open('data/data.txt', 'r')
nums = []
for num in data:
    nums.append(int(num))
print("Sum:", sum(nums))
print("Min:", min(nums))
print("Max:", max(nums))
print("Avg:", sum(nums)/len(nums))

data.close()
```

```
Sum: 25
Min: -2
Max: 11
Avg: 3.5714285714285716
```

3. Output the results to a newly created file.

```
file = open('data/new.txt', 'w')

file.write("Sum: " + str(sum(nums)))
file.write("Min: " + str(min(nums)))
file.write("Max: " + str(max(nums)))
file.write("Avg: " + str(sum(nums)/len(nums)))

file.close()
```

4. Close the file.

Append output file

1. Adults only! Ask a customer their age. If they are at least 18 years old, print '`Welcome`'.
   Otherwise, print '`Please come back in X years`', where X is the remaining time until they turn 18.

   In addition to the screen output, send the output to a text file, adding every response to the file.

```
# open the file for writing (appending)
a = open('age_output.txt', 'a')
```

```
age_limit = 18
age = int(input('Please enter your age: '))

if age >= 18:
    # print to the screen
    print('Welcome')
    # output into the file
    a.write('Age: ' + str(age) + ' :: ')
    a.write('Welcome\n')
else:
    # print to the screen
    print('Please come back in '+ str(age_limit-age) + ' years')
    # output into the file
    a.write('Age: ' + str(age) + ' :: ')
    a.write('Please come back in '+ str(age_limit-age) + ' years\n')
```

```
# close the file
a.close()
```

Extension activities (for Python pros)

Exercise 1: Sets

1. Create a set containing fruit.

```
fruits = {'banana', 'kiwi', 'orange'}
```

2. Add another fruit to the set and print it.

```
fruits.add('grapefruit')
print(fruits)
```

```
{'kiwi', 'banana', 'orange', 'grapefruit'}
```

3. Loop through the set of fruit and print each element.

```
for fruit in fruits:
    print(fruit)
```

```
kiwi
banana
orange
grapefruit
```

4. Consider the following sets.

```
s1 = {'A', 'B', 'C', 'D', 'E'}
s2 = {'D', 'E', 'F', 'G'}
```

5. Find and print set S3, which contains all elements belonging to either S1 and S2.

```
s3 = s1 | s2
s3
```

```
{'A', 'B', 'C', 'D', 'E', 'F', 'G'}
```

6. Find and print set S3, which contains the elements belonging to both S1 and S2.

```
s3 = s1 & s2
s3
```

```
{'D', 'E'}
```

7. Find and print set S3, which contains the elements that are in S1 but not S2.

```
s3 = s1 - s2
s3
```

```
{'A', 'B', 'C'}
```

8. Find and print set S3, which contains the elements that are in only one of S1 and S2 but not the other.

```
s3 = s1 ^ s2
s3
```

```
{'A', 'B', 'C', 'F', 'G'}
```

Exercise 2: Functions

Write a function that takes two sequences `seq_a` and `seq_b` as arguments and returns `True` if every element in `seq_a` is also an element of `seq_b`, else `False`.

- By 'sequence' we mean a list, a tuple, or a string.
- Do the exercise without using sets and set methods.

```
def seq_check(seq_a, seq_b):
    return all([seq_a_element in seq_b for seq_a_element in seq_a])
```

Exercise 3

Using list comprehension syntax, simplify the loop in the following code.

```
import numpy as np

n = 100
ϵ_values = []
for i in range(n):
    e = np.random.randn()
    ϵ_values.append(e)
```

```
ϵ_values = [np.random.randn() for _ in range(100)]
```

Exercise 4

Using the dictionary `christmas_sequence` print out each verse of the The Twelve Days of Christmas.

For example, the third verse will be: *On the third day of Christmas my true love sent to me three french hens, two turtle doves and a partridge in a pear tree*.

Hint: You will probably need nested loops.

```
christmas_sequence = {"first": "a partridge in a pear tree",
                      "second": "two turtle doves",
                      "third": "three french hens",
                      "fourth": "four calling birds",
                      "fith": "five gold rings",
                      "sixth": "six geese a-laying",
                      "seventh": "seven swans a-swimming",
                      "eigth": "eight maids a-milking",
                      "nineth": "nine ladies dancing",
                      "thenth": "ten lords a-leaping",
                      "eleventh": "eleven pipers piping",
                      "twelth": "twelve drummers drumming"}
```

Exercise 5

The function **`dir`** allows you to look at the methods and attributes associated with a python object. Pick one or more of the objects we have looked at.

There are two main types of methods and attributes returned. What are they? What is the value in this approach?

```
dir(1)
```

Exercise 6

As you might have realised, variables are accessible between code cells. They are in the same *namespace*. We can look at what is in that namespace using the `globals` and `locals` functions.

- What are the datatypes of the `globals` and `locals` function outputs?
- There are lots of variables listed besides the ones you have created. Many of these are specific to this interface. Can you identify their purpose?

Exercise 4: Mathematical and statistical programming with NumPy

```
# import necessary libraries
import numpy as np

# create data to work with
x = np.arange(10)
x
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

1. Display the first five elements.
2. Display every other element.
3. Display the elements from index 5 in reverse order.

```
# two dimensional array
y = np.array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

4. Display the first 2 rows and the first 3 columns.
5. Display the first column of y.
6. Display the first row of y.
7. Read the contents of file `cdc_1.csv`, containing heights, weights and ages, into array data. To do this, you can use the below code:

```
data = np.genfromtxt('data/cdc_1.csv', delimiter=',', skip_header=1)
```

8. Calculate the minimum, maximum, and mean height, weight, and age.

Descriptive statistics with NumPy

```
# import necessary libraries
import numpy as np
import scipy.stats
```

1. Read the contents of file `cdc_nan.csv`, containing heights, weights and ages, into array data. To do this, you can use the below code:

```
data = np.genfromtxt('data/cdc_nan.csv', delimiter=',', skip_header=1)
```

2. Separate the heights (column 0) and the weights (column 1).

Univariate

Measures of central tendency

Median

1. Calculate the median for the heights and the weights and assign the values to variables.
2. What has happened? Check if the arrays contain missing values.
3. Array weights contain three nan values. Find their positions.
4. Now calculate the median for the weights ignoring the nan values.

Mean

Calculate the mean values for heights and weights.

Measures of spread (variability)

Range

Calculate the range of the heights and the weights. Remember, range is the difference between the maximum and the minimum value.

Interquartile Range (IQR)

Find the quartiles for the heights and the weights, and the Interquartile Range (IQR).

Variance

Calculate population and sample variance for heights and weights.

Standard deviation

Calculate population and sample standard deviation for heights and weights.

Bivariate

Covariance

Find the covariance between heights and weights and comment on its direction and strength.

Correlation

Find the correlation between heights and weights and comment on its direction and strength.

Exercise 4: Mathematical and statistical programming with NumPy

```
# import necessary libraries
import numpy as np

# create data to work with
x = np.arange(10)
x
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

1. Display the first five elements.

```
x[:5]
```

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

2. Display every other element.

```
x[::2]
```

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

3. Display the elements from index 5 in reverse order.

```
x[5::-1]
```

```
array([5, 4, 3, 2, 1, 0])
```

```
# two dimensional array
y = np.array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

4. Display the first 2 rows and the first 3 columns.

```
y[:2, :3]
```

```
array([[12,  5,  2],
       [ 7,  6,  8]])
```

5. Display the first column of y.

```
y[:,0]
```

```
array([12,  7,  1])
```

6. Display the first row of y.

```
y[0,:]
```

```
array([12,  5,  2,  4])
```

7. Read the contents of file `cdc_1.csv`, containing heights, weights and ages, into array data. To do this, you can use the below code:

```
data = np.genfromtxt('data/cdc_1.csv', delimiter=',', skip_header=1)
```

8. Calculate the minimum, maximum, and mean height, weight, and age.

```
data.min(axis=0)
```

```
array([49., 78., 18.])
```

```
data.max(axis=0)
```

```
array([ 93., 500.,  94.])
```

```
data.mean(axis=0)
```

```
array([ 70.25164594, 189.32270875,  44.27306929])
```

Descriptive statistics with NumPy

```
# import necessary libraries
import numpy as np
import scipy.stats
```

1. Read the contents of file `cdc_nan.csv`, containing heights, weights and ages, into array data. To do this, you can use the below code:

```
data = np.genfromtxt('data/cdc_nan.csv', delimiter=',', skip_header=1)
```

2. Separate the heights (column 0) and the weights (column 1).

```
heights = data[:,0]
```

```
weights = data[:,1]
```

Univariate

Measures of central tendency

Median

1. Calculate the median for the heights and the weights and assign the values to variables.

```
median_heights = np.median(heights)
median_heights
```

```
70.0
```

```
median_weights = np.median(weights)
median_weights
```

```
nan
```

2. What has happened? Check if the arrays contain missing values.

```
np.isnan(heights)
```

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

```
np.sum(np.isnan(heights))
```

```
0
```

```
np.isnan(weights)
```

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

```
np.sum(np.isnan(weights))
```

```
3
```

3. Array weights contain three nan values. Find their positions.

```
np.argwhere(np.isnan(weights))
```

```
array([[ 3],
       [11],
       [17]], dtype=int64)
```

4. Now calculate the median for the weights ignoring the nan values.

```
median_weights = np.nanmedian(weights)
median_weights
```

```
185.0
```

Mean

Calculate the mean values for heights and weights.

```
mean_heights = np.mean(heights)
mean_heights
```

```
70.25164594001463
```

```
mean_weights = np.nanmean(weights)
mean_weights
```

```
189.3251097637466
```

Measures of spread (variability)

Range

Calculate the range of the heights and the weights. Remember, range is the difference between the maximum and the minimum value.

```
range_heights = np.max(heights) - np.min(heights)
range_heights
```

```
44.0
```

```
range_weights = np.max(weights) - np.min(weights)
range_weights
```

```
nan
```

```
range_weights = np.nanmax(weights) - np.nanmin(weights)
range_weights
```

```
422.0
```

Interquartile Range (IQR)

Find the quartiles for the heights and the weights, and the Interquartile Range (IQR).

```
quant_heights = np.quantile(heights, [0.25, 0.5, 0.75])
quant_heights
```

```
array([68., 70., 72.])
```

```
IQR_heights = quant_heights[2] - quant_heights[0]
IQR_heights
```

```
4.0
```

```
quant_weights = np.quantile(weights, [0.25, 0.5, 0.75])
quant_weights
```

```
array([nan, nan, nan])
```

```
quant_weights = np.nanquantile(weights, [0.25, 0.5, 0.75])
quant_weights
```

```
array([165., 185., 210.])
```

```
IQR_weights = quant_weights[2] - quant_weights[0]
IQR_weights
```

```
45.0
```

Variance

Calculate population and sample variance for heights and weights.

```
# population variance - heights
var_pop_heights = np.var(heights,ddof=0)
var_pop_heights
```

```
9.054450473032162
```

```
# sample variance - heights
var_sample_heights = np.var(heights,ddof=1)
var_sample_heights
```

```
9.055396799377588
```

```
# population variance - weights
var_pop_weights = np.var(weights,ddof=0)
var_pop_weights
```

```
nan
```

```
# population variance - weights
var_pop_weights = np.nanvar(weights,ddof=0)
var_pop_weights
```

```
1336.1876341871994
```

```
# sample variance - weights
var_sample_weights = np.nanvar(weights,ddof=1)
var_sample_weights
```

```
1336.3273297056714
```

Standard deviation

Calculate population and sample standard deviation for heights and weights.

```
# population standard variation - heights
std_pop_heights = np.std(heights,ddof=0)
std_pop_heights
```

```
3.009061394028404
```

```
# sample standard deviation - heights
std_sample_heights = np.std(heights,ddof=1)
std_sample_heights
```

```
3.009218636021249
```

```
# population standard deviation - weights
std_pop_weights = np.std(weights,ddof=0)
std_pop_weights
```

```
nan
```

```
# population standard deviation - weights
std_pop_weights = np.nanstd(weights,ddof=0)
std_pop_weights
```

```
36.55390039636262
```

```
# sample standard deviation - weights
std_sample_weights = np.nanstd(weights,ddof=1)
std_sample_weights
```

```
36.555811161916125
```

Bivariate

Covariance

Find the covariance between heights and weights and comment on its direction and strength.

```
cov_matrix = np.cov(heights,weights)
cov_matrix
```

```
array([[9.0553968,       nan],
       [      nan,       nan]])
```

```
weights_1 = weights[~np.isnan(weights)]
heights_1 = heights[~np.isnan(weights)]
```

```
cov_matrix = np.cov(heights_1,weights_1)
cov_matrix
```

```
array([[   9.05811323,   47.78471363],
       [  47.78471363, 1336.32732971]])
```

Positive covariance - height and weight both increase. We cannot judge the strength of the relationship. For that we have to calculate the correlation.

Correlation

Find the correlation between heights and weights and comment on its direction and strength.

```
corr_matrix = np.corrcoef(heights,weights)
corr_matrix
```

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

```
corr_matrix = np.corrcoef(heights_1,weights_1)
corr_matrix
```

```
array([[1.        , 0.43432386],
       [0.43432386, 1.        ]])
```

Positive correlation - height and weight both increase. Weak correlation (<0.5).

Exercise 5: Introduction to Pandas

Pandas exercise

```
# import necessary libraries
import numpy as np
import pandas as pd
```

Series

1. Create a series from the below list:

   ```
   attempts = [1, 3, 2, 3, 2, 3, 1, 1, 2, 1]
   ```

2. Find the first person's number of attempts.
3. Find the first five people's number of attempts.
4. Find the last three people's number of attempts.

DataFrames

1. Create and display a DataFrame from the below dictionary exam_data.

   ```
   exam_data = {'name': ['Anne', 'Kofi', 'Ridhwaan', 'Zara',
   'Muhammad', 'Donald', 'Isabella', 'Sarah', 'Paula', 'John'],
   'score': [12.5, 9, 16.5, None, 9, 20, 14.5, None, 8, 19],
   'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
   'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no',
   'no', 'yes']}

   labels = ['Student ' + letter for letter in ['a', 'b', 'c', 'd',
   'e', 'f', 'g', 'h', 'i', 'j']]
   ```

2. Display the first three rows of the data frame.
3. Display the name and score columns of the data frame.
4. Display the name and score columns and rows 1, 3, 4, and 8.
5. Display the rows where the number of attempts is greater than 2.
6. Display the rows where score is greater or equal than a pass rate of 10.

7. Display the rows where the score is above 15 and the student has only made one attempt.
8. Set the index of the DataFrame so it is given by labels. Hint: df.index = ...

   Return the name of student d.

Reading in data

1. Read the file `mortgage_applicants.csv,` which sits in the `data` folder, into a variable called `mortgage`.

   Hint: Your path will need to reflect the location of the file.
2. The same data sits in an excel file called `mortgage_applicants.xlsx`, in a sheet called `PrevYear`. Read that into another DataFrame called `mortgage_excel.`
3. Some weather data is held in a file called `weather_data.json`. Read it into a dataframe called `weather`.

Changing types and parsing times

1. What data type has each column in `mortgage` been read in as? Use a method to find out.
2. Convert the `ID` column so that it is instead represented as a Unicode string. *Hint: the type is denoted 'string'*
3. Convert the `day` column of the `weather` Dataframe to an appropriate type.

Retrieving rows and columns

1. Select the `Score` column of the `mortgage` DataFrame.
2. Select the `Balance` and `Income` columns.
3. Select the first row in the mortgage DataFrame.
4. What is the `Debt` of the first applicant?
5. What is the `Balance` and `Income` of the 10th to 20th applicants?
6. What are the final 5 people's `Score`?

7. Using weather, display the weather on the most recent day.

Querying DataFrames

1. Compute the monthly earnings of mortgage applicants, just using `Income.`
2. Compute the ratio of debts to assets for each mortgage applicant.
3. Display all mortgage applicants who have a `Balance` greater than £1000.
4. Display all mortgage applicants who have a `Balance` greater than £1000 and a `Debt` below £50.
5. Display all loan applicants who have an `Income` greater than 30,000 who have a 10-year loan, and those with an `Income` greater than 20,000 who have a 20-year loan (together!)

Aggregating DataFrames

1. Compute the average `Balance` of mortgage applicants depending on the `Term` of their loan.
2. Compute the average `Income` of mortgage applicants depending on whether they defaulted or not.
3. Compute the average `Debt`, `Income`, and `Balance` of mortgage applicants based on whether they defaulted or not, and the term of their loan.
4. Add a column to the DataFrame called MeanTermIncome, which contains the mean `Income` of mortgage applicants based on their `Term`.

Exercise 5: Introduction to Pandas

Pandas exercise

```
# import necessary libraries
import numpy as np
import pandas as pd
```

Series

1. Create a series from the below list:

   ```
   attempts = [1, 3, 2, 3, 2, 3, 1, 1, 2, 1]
   ```

   ```
   attempts = pd.Series(attempts)
   ```

2. Find the first person's number of attempts.

   ```
   attempts[0]
   ```

   ```
   1
   ```

3. Find the first five people's number of attempts.

   ```
   attempts[:5]
   ```

   ```
   0    1
   1    3
   2    2
   3    3
   4    2
   dtype: int64
   ```

4. Find the last three people's number of attempts.

   ```
   attempts[-3:]
   ```

   ```
   7    1
   8    2
   9    1
   dtype: int64
   ```

DataFrames

1. Create and display a DataFrame from the below dictionary exam_data.

```
exam_data = {'name': ['Anne', 'Kofi', 'Ridhwaan', 'Zara',
'Muhammad', 'Donald', 'Isabella', 'Sarah', 'Paula', 'John'],
'score': [12.5, 9, 16.5, None, 9, 20, 14.5, None, 8, 19],
'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no',
'no', 'yes']}

labels = ['Student ' + letter for letter in ['a', 'b', 'c', 'd',
'e', 'f', 'g', 'h', 'i', 'j']]
```

```
exam = pd.DataFrame(exam_data)
```

2. Display the first three rows of the data frame.

```
exam.loc[:2, :]
```

	name	score	attempts	qualify
0	Anne	12.5	1	yes
1	Kofi	9.0	3	no
2	Ridhwaan	16.5	2	yes

3. Display the name and score columns of the data frame.

```
exam[['name', 'score']]
```

	name	score
0	Anne	12.5
1	Kofi	9.0
2	Ridhwaan	16.5
3	Zara	NaN
4	Muhammad	9.0
5	Donald	20.0
6	Isabella	14.5
7	Sarah	NaN
8	Paula	8.0
9	John	19.0

4. Display the name and score columns and rows 1, 3, 4, and 8.

```
exam.loc[[1, 3, 4, 8], ['name', 'score']]
```

	name	score
1	Kofi	9.0
3	Zara	NaN
4	Muhammad	9.0
8	Paula	8.0

5. Display the rows where the number of attempts is greater than 2.

```
exam[exam['attempts'] > 2]
```

	name	score	attempts	qualify
1	Kofi	9.0	3	no
3	Zara	NaN	3	no
5	Donald	20.0	3	yes

6. Display the rows where score is greater or equal than a pass rate of 10.

```
exam[exam['score'] >= 10]
```

	name	score	attempts	qualify
0	Anne	12.5	1	yes
2	Ridhwaan	16.5	2	yes
5	Donald	20.0	3	yes
6	Isabella	14.5	1	yes
9	John	19.0	1	yes

7. Display the rows where the score is above 15 and the student has only made one attempt.

```
exam[(exam['score'] > 10) & (exam['attempts'] == 1)]
```

	name	score	attempts	qualify
0	Anne	12.5	1	yes
6	Isabella	14.5	1	yes
9	John	19.0	1	yes

8. Set the index of the DataFrame so it is given by labels. Hint: df.index = ...

   Return the name of student d.

```
exam.index = labels

exam.loc["Student d", "name"]
```

```
'Zara'
```

Reading in data

1. Read the file `mortgage_applicants.csv`, which sits in the `data` folder, into a variable called `mortgage`.

   Hint: Your path will need to reflect the location of the file.

```
mortgage = pd.read_csv('data/mortgage_applicants.csv')
mortgage
```

	Unnamed: 0	ID	Income	Term	Balance	Debt	Score	Default
0	0	567	17626	10 Years	1381	293	228.0	False
1	1	523	18959	20 Years	883	1012	187.0	False
2	2	544	20560	10 Years	684	898	86.0	False
3	3	370	21894	10 Years	748	85	NaN	False
4	4	756	24430	10 Years	1224	59	504.0	False
...	...	...	...	...	...	...	...	...
851	851	71	30191	20 Years	1319	3880	55.0	True
852	852	932	41669	20 Years	1385	32	780.0	False
853	853	39	36816	20 Years	1868	3123	366.0	True
854	854	283	42145	20 Years	1447	2498	422.0	False
855	855	847	30594	20 Years	1216	2473	179.0	True

2. The same data sits in an excel file called `mortgage_applicants.xlsx`, in a sheet called `PrevYear`. Read that into another DataFrame called `mortgage_excel`.

```
mortgage_excel = pd.read_excel('data/mortgage_applicants.xlsx', sheet_name='PrevYear')
mortgage_excel
```

	Unnamed: 0	ID	Income	Term	Balance	Debt	Score	Default
0	0	567	17626	10 Years	1381	293	228.0	False
1	1	523	18959	20 Years	883	1012	187.0	False
2	2	544	20560	10 Years	684	898	86.0	False
3	3	370	21894	10 Years	748	85	NaN	False
4	4	756	24430	10 Years	1224	59	504.0	False
...	...	...	...	...	...	...	...	...
851	851	71	30191	20 Years	1319	3880	55.0	True
852	852	932	41669	20 Years	1385	32	780.0	False
853	853	39	36816	20 Years	1868	3123	366.0	True
854	854	283	42145	20 Years	1447	2498	422.0	False
855	855	847	30594	20 Years	1216	2473	179.0	True

3. Some weather data is held in a file called `weather_data.json`. Read it into a dataframe called `weather`.

```
weather = pd.read_json("data/weather_data.json")
weather
```

	day	temp	humidity	sun_hrs
0	2023-07-15	15.68	73.18	6.40
1	2023-07-16	25.16	83.88	8.06
2	2023-07-17	13.26	80.05	4.89
3	2023-07-18	24.63	82.37	9.13
4	2023-07-19	12.78	83.10	17.10
5	2023-07-20	23.52	85.35	0.72
6	2023-07-21	17.80	85.64	5.79
7	2023-07-22	24.98	76.81	10.95
8	2023-07-23	23.48	80.86	3.77
9	2023-07-24	23.30	79.96	14.62

Changing types and parsing times

1. What data type has each column in `mortgage` been read in as? Use a method to find out.

```
mortgage.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 856 entries, 0 to 855
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Unnamed: 0  856 non-null    int64
 1   ID          856 non-null    int64
 2   Income      856 non-null    int64
 3   Term        856 non-null    object
 4   Balance     856 non-null    int64
 5   Debt        856 non-null    int64
 6   Score       836 non-null    float64
 7   Default     856 non-null    bool
dtypes: bool(1), float64(1), int64(5), object(1)
memory usage: 47.8+ KB
```

2. Convert the `ID` column so that it is instead represented as a Unicode string. *Hint: the type is denoted 'U'*

```
mortgage = mortgage.astype({'ID': 'string'})
```

```
mortgage.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 856 entries, 0 to 855
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Unnamed: 0  856 non-null    int64
 1   ID          856 non-null    string
 2   Income      856 non-null    int64
 3   Term        856 non-null    object
 4   Balance     856 non-null    int64
 5   Debt        856 non-null    int64
 6   Score       836 non-null    float64
 7   Default     856 non-null    bool
dtypes: bool(1), float64(1), int64(4), object(1), string(1)
memory usage: 47.8+ KB
```

3. Convert the `day` column of the `weather` Dataframe to an appropriate type.

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

```
<class 'pandas.core.frame.DataFrame'>
Index: 10 entries, 0 to 9
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype         
---  ------    --------------  -----         
 0   day       10 non-null     datetime64[ns]
 1   temp      10 non-null     float64       
 2   humidity  10 non-null     float64       
 3   sun_hrs   10 non-null     float64       
dtypes: datetime64[ns](1), float64(3)
memory usage: 400.0 bytes
```

Retrieving rows and columns

1. Select the `Score` column of the `mortgage` DataFrame.

```
mortgage['Score']
```

```
0      228.0
1      187.0
2       86.0
3        NaN
4      504.0
       ...
851     55.0
852    780.0
853    366.0
854    422.0
855    179.0
Name: Score, Length: 856, dtype: float64
```

2. Select the `Balance` and `Income` columns.

```
mortgage[['Balance', 'Income']]
```

	Balance	Income
0	1381	17626
1	883	18959
2	684	20560
3	748	21894
4	1224	24430
...	...	...
851	1319	30191
852	1385	41669
853	1868	36816
854	1447	42145
855	1216	30594

856 rows × 2 columns

3. Select the first row in the mortgage DataFrame.

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

```
Unnamed: 0           0
ID                 567
Income           17626
Term          10 Years
Balance           1381
Debt               293
Score            228.0
Default          False
Name: 0, dtype: object
```

4. What is the `Debt` of the first applicant?

```
mortgage.loc[0, 'Debt']
```

```
293
```

5. What is the `Balance` and `Income` of the 10th to 20th applicants?

```
mortgage.loc[10:20, ['Balance', 'Income']]
```

	Balance	Income
10	854	19166
11	946	20838
12	1316	18630
13	1160	24182
14	901	17887
15	701	23596
16	889	18223
17	991	18271
18	1214	17887
19	973	19330
20	771	24185

6. What are the final 5 people's `Score`?

```
mortgage.iloc[-5:, 6]
```

```
851     55.0
852    780.0
853    366.0
854    422.0
855    179.0
Name: Score, dtype: float64
```

7. Using weather, display the weather on the most recent day.

```
weather[-1:]
```

	day	temp	humidity	sun_hrs
9	2023-07-24	23.3	79.96	14.62

Querying DataFrames

1. Compute the monthly earnings of mortgage applicants, just using `Income`.

```
mortgage['Income'] / 12
```

```
0      1468.833333
1      1579.916667
2      1713.333333
3      1824.500000
4      2035.833333
          ...
851    2515.916667
852    3472.416667
853    3068.000000
854    3512.083333
855    2549.500000
Name: Income, Length: 856, dtype: float64
```

2. Compute the ratio of debts to assets for each mortgage applicant.

```
mortgage['Debt'] / mortgage['Balance']
```

```
0      0.212165
1      1.146093
2      1.312865
3      0.113636
4      0.048203
          ...
851    2.941622
852    0.023105
853    1.671842
854    1.726330
855    2.033717
Length: 856, dtype: float64
```

3. Display all mortgage applicants who have a `Balance` greater than £1000.

```
mortgage[mortgage['Balance'] > 1000]
```

	Unnamed: 0	ID	Income	Term	Balance	Debt	Score	Default
0	0	567	17626	10 Years	1381	293	228.0	False
4	4	756	24430	10 Years	1224	59	504.0	False
5	5	929	22995	20 Years	1678	1329	384.0	False
6	6	373	21124	10 Years	1135	115	560.0	False
7	7	818	24644	10 Years	1634	105	309.0	False
...	...	...	...	...	...	...	...	...
851	851	71	30191	20 Years	1319	3880	55.0	True
852	852	932	41669	20 Years	1385	32	780.0	False
853	853	39	36816	20 Years	1868	3123	366.0	True
854	854	283	42145	20 Years	1447	2498	422.0	False
855	855	847	30594	20 Years	1216	2473	179.0	True

543 rows × 8 columns

4. Display all mortgage applicants who have a `Balance` greater than £1000 and a `Debt` below £50.

```
mortgage[(mortgage['Balance'] > 1000) & (mortgage['Debt'] < 50)]
```

	Unnamed: 0	ID	Income	Term	Balance	Debt	Score	Default
12	12	327	18630	20 Years	1316	37	430.0	False
54	54	608	25267	10 Years	1308	2	470.0	False
56	56	67	20528	10 Years	1253	40	511.0	False
59	59	316	16142	10 Years	1251	12	281.0	False
66	66	472	19400	10 Years	1292	24	235.0	False
...	...	...	...	...	...	...	...	...
841	841	575	36132	10 Years	1596	1	992.0	False
842	842	324	23597	20 Years	1419	6	465.0	False
847	847	96	26553	20 Years	1805	26	491.0	False
850	850	627	27676	10 Years	1598	39	445.0	False
852	852	932	41669	20 Years	1385	32	780.0	False

5. Display all loan applicants who have an `Income` greater than 30,000 who have a 10-year loan, and those with an `Income` greater than 20,000 who have a 20-year loan (together!)

```
mortgage[((mortgage['Income'] > 30_000) & (mortgage['Term'] == '10 Years'))
        |((mortgage['Income'] > 20_000) & (mortgage['Term'] == '20 Years'))]
```

	Unnamed: 0	ID	Income	Term	Balance	Debt	Score	Default
5	5	929	22995	20 Years	1678	1329	384.0	False
8	8	284	27138	20 Years	840	1877	251.0	False
22	22	616	22869	20 Years	1150	803	431.0	True
32	32	740	22259	20 Years	985	1906	152.0	True
50	50	789	23557	20 Years	819	79	350.0	False
...	...	...	...	...	...	...	...	...
851	851	71	30191	20 Years	1319	3880	55.0	True
852	852	932	41669	20 Years	1385	32	780.0	False
853	853	39	36816	20 Years	1868	3123	366.0	True
854	854	283	42145	20 Years	1447	2498	422.0	False
855	855	847	30594	20 Years	1216	2473	179.0	True

403 rows × 8 columns

Aggregating DataFrames

1. Compute the average `Balance` of mortgage applicants depending on the `Term` of their loan.

```
mortgage[['Balance', 'Term']].groupby('Term').mean()
```

	Balance
Term	
10 Years	1155.763699
20 Years	1334.095588

2. Compute the average `Income` of mortgage applicants depending on whether they defaulted or not.

```
mortgage[['Income', 'Default']].groupby('Default').mean()
```

	Income
Default	
False	30315.666667
True	26438.409091

3. Compute the average `Debt`, `Income`, and `Balance` of mortgage applicants based on whether they defaulted or not, and the term of their loan.

```
mortgage[['Debt', 'Income', 'Balance', 'Term', 'Default']].groupby(['Default', 'Term']).mean()
```

		Debt	Income	Balance
Default	**Term**			
False	**10 Years**	546.523719	28228.009488	1161.590133
	20 Years	631.228216	34880.792531	1344.688797
True	**10 Years**	1805.561404	23848.526316	1101.894737
	20 Years	1904.935484	31200.451613	1251.741935

4. Add a column to the DataFrame called MeanTermIncome, which contains the mean `Income` of mortgage applicants based on their `Term`.

```
mortgage['MeanTermIncome'] = mortgage[['Income', 'Term']].groupby('Term').transform(np.mean).round(2)
mortgage
```

	Unnamed: 0	ID	Income	Term	Balance	Debt	Score	Default	MeanTermIncome
0	0	567	17626	10 Years	1381	293	228.0	False	27800.56
1	1	523	18959	20 Years	883	1012	187.0	False	34461.34
2	2	544	20560	10 Years	684	898	86.0	False	27800.56
3	3	370	21894	10 Years	748	85	NaN	False	27800.56
4	4	756	24430	10 Years	1224	59	504.0	False	27800.56
...	...	...	...	...	...	...	...	...	...
851	851	71	30191	20 Years	1319	3880	55.0	True	34461.34
852	852	932	41669	20 Years	1385	32	780.0	False	34461.34
853	853	39	36816	20 Years	1868	3123	366.0	True	34461.34
854	854	283	42145	20 Years	1447	2498	422.0	False	34461.34
855	855	847	30594	20 Years	1216	2473	179.0	True	34461.34

856 rows × 9 columns

Exercise 6: Data cleaning with Pandas

Load in the dataset `renfe_trains.csv`.

Initial data inspection

1. Inspect the columns of the DataFrame. Specifically, consider the type of each column and whether it seems reasonable. If not, investigate why.
2. It seems like we have some bad values in the `price` column with the value 'price'.
   You can see them by using the method `.value_counts()`.

   Inspect the specific rows where this is the case.
3. It looks like some sort of error has meant the column names have been fed into the data in intervals. Let's drop these rows as they are clearly an accident.
4. We can now represent `price` using the appropriate type. Convert it to the appropriate data type.

Missing values

1. Identify whether there are missing values in the DataFrame.
2. Which columns are they in?
3. Inspect some rows which contain them.
4. Drop all rows which have missing `vehicle_class` **and** `price` **and** `fare`. Hint: how='all'
5. Run the below code. What does it suggest about ticket price with respect to `vehicle_class` and `fare`?

   ```
   df[['vehicle_class', 'fare', 'price']].groupby(['vehicle_class', 'fare']).mean()
   ```
6. Fill the remaining missing price values with the mean of all the prices.
   - In the extension, you can try to tackle this more appropriately (and trickily!).
7. Check you have gotten rid of all `NaN` values in df.

Deduplication

1. Use `duplicated` to see whether the dataset contains any duplicated rows.
2. As the dataset constitutes ticket price search results, there's a good chance duplication has come about due to the data collection method. E.g., there are many tickets available on each train.

   We would typically investigate this further, but in order to see the functionality pandas offers, just remove the duplicate rows.

Data transformation

Now that we have a dataset without the above errors, we want to add some meaningful columns.

1. First, let's bin `price` into three groups and add that as column to the DataFrame called `price_bin`.

   The bins should be `[(0, 150], (150, 300], (300, 450]]`.
2. Compute the amount of tax that would need to be paid if vat is at 20% and prices aren't inclusive of it.

Text data manipulation

1. Capitalise the word `renfe` in the `company` column.
2. Generate a statistical summary of trips whose `vehicle_class` contains `Turista`.
3. Display all rows whose `fare` ends with a `+`.

Extension activity

1. As it appears that `price` depends upon `vehicle_class` and `fare`, we choose to replace missing `price` values with the average for their `vehicle_class` and `fare` category. Write some code which does this.

Exercise 6: Data cleaning with Pandas

Load in the dataset `renfe_trains.csv`.

```
import pandas as pd
import numpy as np

df = pd.read_csv('data/renfe_trains.csv')
```

Initial data inspection

1. Inspect the columns of the DataFrame. Specifically, consider the type of each column and whether it seems reasonable. If not, investigate why.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 85948 entries, 0 to 85947
Data columns (total 8 columns):
 #   Column         Non-Null Count  Dtype 
---  ------         --------------  ----- 
 0   company        85948 non-null  object
 1   origin         85948 non-null  object
 2   destination    85948 non-null  object
 3   departure      85948 non-null  object
 4   arrival        85948 non-null  object
 5   vehicle_class  77116 non-null  object
 6   price          72769 non-null  object
 7   fare           77116 non-null  object
dtypes: object(8)
memory usage: 5.2+ MB
```

2. It seems like we have some bad values in the `price` column with the value 'price'.
   You can see them by using the method `.value_counts()`.

```
df['price'].value_counts()
```

```
price
price    3875
76.3     3794
85.1     3615
107.7    2845
53.4     2719
         ...
98.01       1
98.2        1
69.05       1
19.75       1
61.15       1
Name: count, Length: 389, dtype: int64
```

Inspect the specific rows where this is the case.

```
df[df['price']=='price']
```

	company	origin	destination	departure	arrival	vehicle_class	price	fare
69	company	origin	destination	departure	arrival	vehicle_class	price	fare
146	company	origin	destination	departure	arrival	vehicle_class	price	fare
209	company	origin	destination	departure	arrival	vehicle_class	price	fare
287	company	origin	destination	departure	arrival	vehicle_class	price	fare
347	company	origin	destination	departure	arrival	vehicle_class	price	fare
...	...	...	...	...	...	...	...	...

3. It looks like some sort of error has meant the column names have been fed into the data in intervals. Let's drop these rows as they are clearly an accident.

```
df = df[df['price'] != 'price']
```

```
df['price'].value_counts()
```

```
price
76.3     3794
85.1     3615
107.7    2845
53.4     2719
60.3     2544
         ...
49.67       1
166.6       1
69.39       1
55.85       1
61.15       1
Name: count, Length: 388, dtype: int64
```

4. We can now represent `price` using the appropriate type. Convert it to the appropriate data type.

```
df['price'] = df['price'].astype(np.float32)
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 82073 entries, 0 to 85947
Data columns (total 8 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   company        82073 non-null  object
 1   origin         82073 non-null  object
 2   destination    82073 non-null  object
 3   departure      82073 non-null  object
 4   arrival        82073 non-null  object
 5   vehicle_class  73241 non-null  object
 6   price          68894 non-null  float32
 7   fare           73241 non-null  object
dtypes: float32(1), object(7)
memory usage: 5.3+ MB
```

Missing values

1. Identify whether there are missing values in the DataFrame.

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

```
company          False
origin           False
destination      False
departure        False
arrival          False
vehicle_class     True
price             True
fare              True
dtype: bool
```

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

```
company              0
origin               0
destination          0
departure            0
arrival              0
vehicle_class     8832
price            13179
fare              8832
dtype: int64
```

2. Which columns are they in?
   vehicle_class, price, fare

3. Inspect some rows which contain them.

```
df[df['price'].isna()]
```

	company	origin	destination	departure	arrival	vehicle_class	price	fare
11	renfe	MADRID	BARCELONA	2019-05-03 18:30:00	2019-05-03 21:20:00	Preferente	NaN	Promo
15	renfe	MADRID	BARCELONA	2019-04-23 07:30:00	2019-04-23 10:40:00	Turista	NaN	Promo
33	renfe	MADRID	SEVILLA	2019-04-21 21:25:00	2019-04-22 00:10:00	NaN	NaN	NaN
52	renfe	MADRID	SEVILLA	2019-04-17 09:45:00	2019-04-17 12:27:00	Turista	NaN	Flexible
65	renfe	MADRID	SEVILLA	2019-05-03 13:30:00	2019-05-03 16:05:00	Turista	NaN	Promo
...	...	...	...	...	...	...	...	...

4. Drop all rows which have missing `vehicle_class` **and** `price` **and** `fare`. Hint: how='all'

```
df.dropna(subset = ['vehicle_class', 'price', 'fare'],
          how='all',
          inplace=True
         )
```

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

```
company              0
origin               0
destination          0
departure            0
arrival              0
vehicle_class        0
price             4347
fare                 0
dtype: int64
```

5. Run the below code. What does it suggest about ticket price with respect to `vehicle_class` and `fare`?

```
df[['vehicle_class', 'fare', 'price']].groupby(['vehicle_class',
'fare']).mean()
```

Vehicle class and fare appear to affect average ticket price.

6. Fill the remaining missing price values with the mean of all the prices.
    - In the extension, you can try to tackle this more appropriately (and trickily!).

```
df.fillna({'price': df['price'].mean()},
          inplace=True)
```

7. Check you have gotten rid of all `NaN` values in df.

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

```
company          0
origin           0
destination      0
departure        0
arrival          0
vehicle_class    0
price            0
fare             0
dtype: int64
```

Deduplication

1. Use `duplicated` to see whether the dataset contains any duplicated rows.

```
df[df.duplicated()]
```

	company	origin	destination	departure	arrival	vehicle_class	price	fare
39	renfe	MADRID	BARCELONA	2019-04-30 07:00:00	2019-04-30 09:30:00	Turista Plus	94.550003	Promo
71	renfe	MADRID	SEVILLA	2019-05-18 09:00:00	2019-05-18 11:38:00	Turista	76.300003	Flexible
83	renfe	MADRID	BARCELONA	2019-05-27 17:00:00	2019-05-27 19:30:00	Turista	88.949997	Promo
132	renfe	MADRID	BARCELONA	2019-05-10 08:30:00	2019-05-10 11:15:00	Turista	85.099998	Promo
174	renfe	MADRID	BARCELONA	2019-05-13 14:00:00	2019-05-13 16:30:00	Turista	68.650002	Promo
...	...	...	...	...	...	...	...	...

2. As the dataset constitutes ticket price search results, there's a good chance duplication has come about due to the data collection method. E.g., there are many tickets available on each train.

    We would typically investigate this further, but in order to see the functionality pandas offers, just remove the duplicate rows.

```
df.drop_duplicates(inplace=True)
```

Data transformation

Now that we have a dataset without the above errors, we want to add some meaningful columns.

1. First, let's bin `price` into three groups and add that as column to the DataFrame called `price_bin`.

   The bins should be `[(0, 150], (150, 300], (300, 450]]`.

```
df['price_bin'] = pd.cut(x = df['price'],
       bins = [0, 150, 300, 450])
```

2. Compute the amount of tax that would need to be paid if vat is at 20% and prices aren't inclusive of it.

```
df['price'] * 0.20
```

```
0        13.880000
1         8.710000
2        17.020000
3        21.539999
4        21.539999
           ...
85940    10.140000
85941    10.140000
85943    11.050000
85944    11.110000
85946    19.340000
Name: price, Length: 29724, dtype: float32
```

Text data manipulation

1. Capitalise the word `renfe` in the `company` column.

```
df['company'] = df['company'].str.upper()
```

2. Generate a statistical summary of trips whose `vehicle_class` contains `Turista`.

```
df[df['vehicle_class'].str.contains('Turista')].describe()
```

	price
count	26171.000000
mean	70.803299
std	22.684448
min	16.750000
25%	53.400002
50%	71.493797
75%	85.099998
max	206.800003

3. Display all rows whose `fare` ends with a `+`.

```
df[df['fare'].str.match(r'.*\+$')]
```

	company	origin	destination	departure	arrival	vehicle_class	price	fare	price_bin
118	RENFE	MADRID	SEVILLA	2019-05-04 12:00:00	2019-05-04 14:32:00	Turista con enlace	71.493797	Promo +	(0, 150]
139	RENFE	MADRID	SEVILLA	2019-04-26 13:10:00	2019-04-26 20:51:00	Turista con enlace	34.349998	Promo +	(0, 150]
394	RENFE	MADRID	SEVILLA	2019-04-15 13:10:00	2019-04-15 20:51:00	Turista con enlace	34.349998	Promo +	(0, 150]
804	RENFE	MADRID	SEVILLA	2019-04-14 13:00:00	2019-04-14 15:40:00	Turista con enlace	71.493797	Promo +	(0, 150]
818	RENFE	MADRID	SEVILLA	2019-04-26 13:10:00	2019-04-26 20:51:00	Turista con enlace	71.493797	Promo +	(0, 150]
...	...	...	...	...	...	...	...	...	...

Extension activity

1. As it appears that `price` depends upon `vehicle_class` and `fare`, we choose to replace missing `price` values with the average for their `vehicle_class` and `fare` category. Write some code which does this.

```
df['price'] = df[['vehicle_class', 'fare', 'price']].groupby(
                ['vehicle_class', 'fare'])['price'].transform(
                lambda p : p.fillna(p.mean()))
df
```

	company	origin	destination	departure	arrival	vehicle_class	price	fare	price_bin
0	RENFE	MADRID	SEVILLA	2019-05-07 19:00:00	2019-05-07 21:38:00	Preferente	69.400002	Promo	(0, 150]
1	RENFE	MADRID	SEVILLA	2019-05-07 21:25:00	2019-05-08 00:10:00	Turista	43.549999	Promo	(0, 150]
2	RENFE	MADRID	BARCELONA	2019-05-11 08:30:00	2019-05-11 11:15:00	Turista	85.099998	Promo	(0, 150]
3	RENFE	MADRID	BARCELONA	2019-04-18 14:30:00	2019-04-18 17:21:00	Turista	107.699997	Flexible	(0, 150]
4	RENFE	MADRID	BARCELONA	2019-04-28 16:30:00	2019-04-28 19:15:00	Turista	107.699997	Flexible	(0, 150]
...	...	...	...	...	...	...	...	...	...

Exercise 7: Data manipulation with Pandas

Load in the dataset `renfe_trains_cleaned.csv.`

Pivot tables

1. Use a pivot table to explore how `price` differs with respect to the type of `fare` for each destination.

Working with time series

1. Convert `departure` & `arrival` to a more appropriate datatype.
2. Calculate the duration of each train journey and add it as a column called `duration.`
3. Make `departure` the index of the DataFrame.
4. Sort the index low to high (earlier to later). This will make slicing possible later.
5. Select all journeys which `departed` on `07/05/19.`
6. Select all journeys which `departed` on `07/05/19` to `11/05/19.`
7. Add one year to each date in the index of the DataFrame (but do not save it!).
8. Create a subset of the DataFrame called `madrid_to_barca` which contains only journeys with `origin` as `MADRID` and `destination` as `BARCELONA.`
9. Select only those tickets in `madrid_to_barca` which are in the `Promo` category for `fare` and `Turista` for `vehicle_class.` Update `madrid_to_barca` to only contain these.
10. Compute a seven day rolling average for price for the `madrid_to_barca` DataFrame. Add it as a column called `rolling`.
11. (To try after completing the combining tables section) Plot the rolling average vs. the actual values of `price.`

Combining tables

1. Read in the `fare_conditions.csv file`. It contains the conditions for the type of ticket that has been purchased.
2. Add the fare conditions to the original `df` DataFrame.

Exercise 7: Data manipulation with Pandas

Load in the dataset `renfe_trains_cleaned.csv`.

Pivot tables

1. Use a pivot table to explore how `price` differs with respect to the type of `fare` for each destination.

```
df.pivot_table(values=['price'],
               index='fare',
               columns='destination').round()
```

	price		
destination	BARCELONA	PONFERRADA	SEVILLA
fare			
Adulto ida	45.0	NaN	51.0
Básica	46.0	NaN	NaN
COD.PROMOCIONAL	68.0	NaN	NaN
Doble Familiar-Flexible	NaN	85.0	NaN
Flexible	113.0	56.0	76.0
Individual-Flexible	NaN	98.0	NaN
Mesa	200.0	NaN	NaN
Promo	79.0	35.0	55.0
Promo +	79.0	41.0	50.0
YOVOY	64.0	33.0	49.0

Working with time series

1. Convert `departure` & `arrival` to a more appropriate datatype.

```
df[['departure', 'arrival']] = df[['departure', 'arrival']].apply(lambda time: pd.to_datetime(time, format='%Y-%m-%d %H:%M:%S'))
```

2. Calculate the duration of each train journey and add it as a column called `duration`.

```
df['duration'] = df['arrival'] - df['departure']
```

3. Make `departure` the index of the DataFrame.

```
df.set_index('departure', inplace=True)
```

4. Sort the index low to high (earlier to later). This will make slicing possible later.

```
df.sort_index(inplace=True)
```

5. Select all journeys which `departed` on `07/05/19.`

```
df.loc['2019-05', :]
```

departure	company	origin	destination	arrival	vehicle_class	price	fare	price_bin	duration
2019-05-01 06:30:00	RENFE	MADRID	BARCELONA	2019-05-01 09:20:00	Turista	107.7000	Flexible	(0, 150]	0 days 02:50:00
2019-05-01 07:00:00	RENFE	MADRID	SEVILLA	2019-05-01 09:55:00	Turista	62.2000	Flexible	(0, 150]	0 days 02:55:00
2019-05-01 07:00:00	RENFE	MADRID	SEVILLA	2019-05-01 09:55:00	Turista	71.4938	Flexible	(0, 150]	0 days 02:55:00
2019-05-01 07:00:00	RENFE	MADRID	BARCELONA	2019-05-01 09:30:00	Turista	100.4000	Promo	(0, 150]	0 days 02:30:00
2019-05-01 07:00:00	RENFE	MADRID	SEVILLA	2019-05-01 09:55:00	Turista Plus	71.4938	Flexible	(0, 150]	0 days 02:55:00
...	...	...	...	...	...	...	...	...	...

6. Select all journeys which `departed` on `07/05/19` to `11/05/19.`

```
df.loc['2019-05-07':'2019-05-11', :]
```

departure	company	origin	destination	arrival	vehicle_class	price	fare	price_bin	duration
2019-05-07 06:10:00	RENFE	MADRID	BARCELONA	2019-05-07 08:40:00	Turista Plus	49.1500	Promo	(0, 150]	0 days 02:30:00
2019-05-07 06:10:00	RENFE	MADRID	BARCELONA	2019-05-07 08:40:00	Turista	58.1500	Promo	(0, 150]	0 days 02:30:00
2019-05-07 06:10:00	RENFE	MADRID	BARCELONA	2019-05-07 08:40:00	Turista Plus	80.1500	Promo	(0, 150]	0 days 02:30:00
2019-05-07 06:20:00	RENFE	MADRID	SEVILLA	2019-05-07 09:16:00	Turista	43.5500	Promo	(0, 150]	0 days 02:56:00
2019-05-07 06:20:00	RENFE	MADRID	BARCELONA	2019-05-07 14:25:00	Turista	71.4938	Promo	(0, 150]	0 days 08:05:00
...	...	...	...	...	...	...	...	...	...

7. Add one year to each date in the index of the DataFrame (but do not save it!).

```
df.index + pd.tseries.offsets.Day(365)
```

```
DatetimeIndex(['2020-04-11 07:15:00', '2020-04-11 09:00:00',
               '2020-04-11 11:05:00', '2020-04-11 12:00:00',
               '2020-04-11 12:30:00', '2020-04-11 14:30:00',
               '2020-04-11 16:30:00', '2020-04-11 18:00:00',
               '2020-04-11 19:30:00', '2020-04-11 20:00:00',
               ...
               '2021-12-01 17:30:00', '2021-12-02 07:15:00',
               '2021-12-02 10:18:00', '2021-12-02 12:30:00',
               '2021-12-02 14:30:00', '2021-12-02 16:00:00',
               '2021-12-03 09:30:00', '2021-12-03 18:30:00',
               '2021-12-03 20:30:00', '2021-12-04 06:30:00'],
              dtype='datetime64[ns]', name='departure', length=29724, freq=None)
```

8. Create a subset of the DataFrame called `madrid_to_barca` which contains only journeys with `origin` as `MADRID` and `destination` as `BARCELONA.`

```
madrid_to_barca = df[(df['origin'] == 'MADRID') & (df['destination'] == 'BARCELONA')]
```

9. Select only those tickets in `madrid_to_barca` which are in the `Promo` category for `fare` and `Turista` for `vehicle_class.` Update `madrid_to_barca` to only contain these.

```
madrid_to_barca = df[(df['fare'] == 'Promo') & (df['vehicle_class'] == 'Turista')]
```

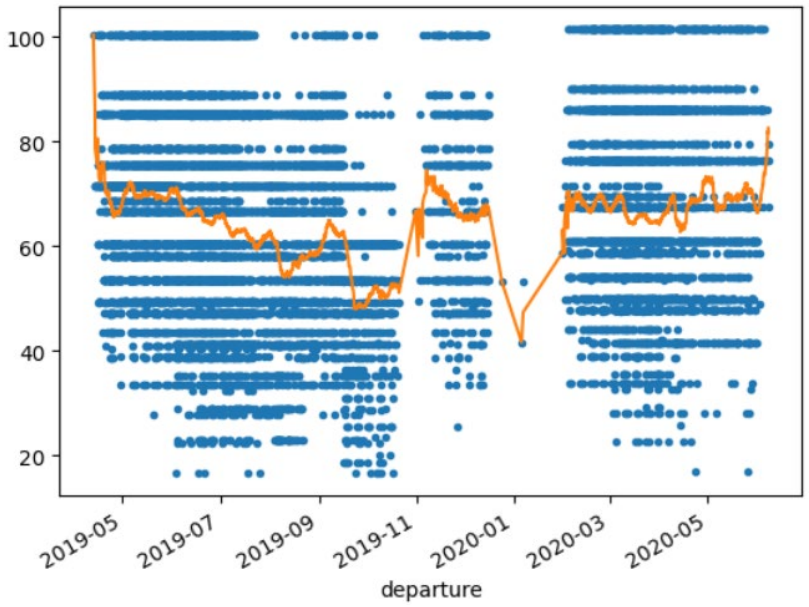
10. Compute a seven day rolling average for price for the `madrid_to_barca` DataFrame. Add it as a column called `rolling`.

```
madrid_to_barca.loc[:, 'rolling'] = madrid_to_barca['price'].rolling(window='7D').mean().round(2)
```

11. (To try after completing the combining tables section) Plot the rolling average vs. the actual values of `price.`

```
madrid_to_barca['price'].plot(style='.')
madrid_to_barca['rolling'].plot(kind='line')
```

```
<Axes: xlabel='departure'>
```



Combining tables

1. Read in the `fare_conditions.csv file`. It contains the conditions for the type of ticket that has been purchased.

```
conditions = pd.read_csv('data/fare_conditions.csv')
```

2. Add the fare conditions to the original `df` DataFrame.

```
df.merge(conditions, on='fare')
```

	company	origin	destination	arrival	vehicle_class	price	fare	price_bin	duration	Conditions
0	RENFE	MADRID	BARCELONA	2019-04-12 16:37:00	Turista	43.2500	Adulto ida	(0, 150]	0 days 09:22:00	Valid for 1 adult to return within 1 day
1	RENFE	MADRID	BARCELONA	2019-04-15 16:37:00	Turista	43.2500	Adulto ida	(0, 150]	0 days 09:22:00	Valid for 1 adult to return within 1 day
2	RENFE	MADRID	BARCELONA	2019-04-16 16:37:00	Turista	71.4938	Adulto ida	(0, 150]	0 days 09:22:00	Valid for 1 adult to return within 1 day
3	RENFE	MADRID	BARCELONA	2019-04-16 16:37:00	Turista	43.2500	Adulto ida	(0, 150]	0 days 09:22:00	Valid for 1 adult to return within 1 day
4	RENFE	MADRID	BARCELONA	2019-04-17 16:37:00	Turista	43.2500	Adulto ida	(0, 150]	0 days 09:22:00	Valid for 1 adult to return within 1 day
...	...	...	...	...	...	...	...	...	...	...
29719	RENFE	MADRID	SEVILLA	2020-09-28 18:30:00	Turista	51.6500	YOVOY	(0, 150]	0 days 02:30:00	Under 25 railcard required
29720	RENFE	MADRID	SEVILLA	2020-09-28 21:38:00	Turista	51.6500	YOVOY	(0, 150]	0 days 02:38:00	Under 25 railcard required
29721	RENFE	MADRID	SEVILLA	2020-09-29 11:38:00	Turista	51.6500	YOVOY	(0, 150]	0 days 02:38:00	Under 25 railcard required
29722	RENFE	MADRID	SEVILLA	2020-09-29 14:32:00	Turista	51.6500	YOVOY	(0, 150]	0 days 02:32:00	Under 25 railcard required
29723	RENFE	MADRID	BARCELONA	2020-09-29 23:40:00	Turista	72.9500	YOVOY	(0, 150]	0 days 03:10:00	Under 25 railcard required

Exercise 8: Methods for visualising data

1. Load in the dataset `train_viz.csv`.
2. Create a scatter plot of `duration` vs. `price`. Segment by `destination`.
3. Create a barplot of `fare counts`.
4. Create a box & whisker plot of `price` with respect to `destination`.
5. Create a histogram of `duration`.
6. Compare this to a histogram of `price`.
7. Create a violin plot of `price` with respect to `destination`.
8. Create a heatmap of the correlation between columns.
9. Create a proportional stacked bar plot of `vehicle_class` by `destination`.

Exercise 8: Methods for visualising data

1. Load in the dataset `train_viz.csv`.

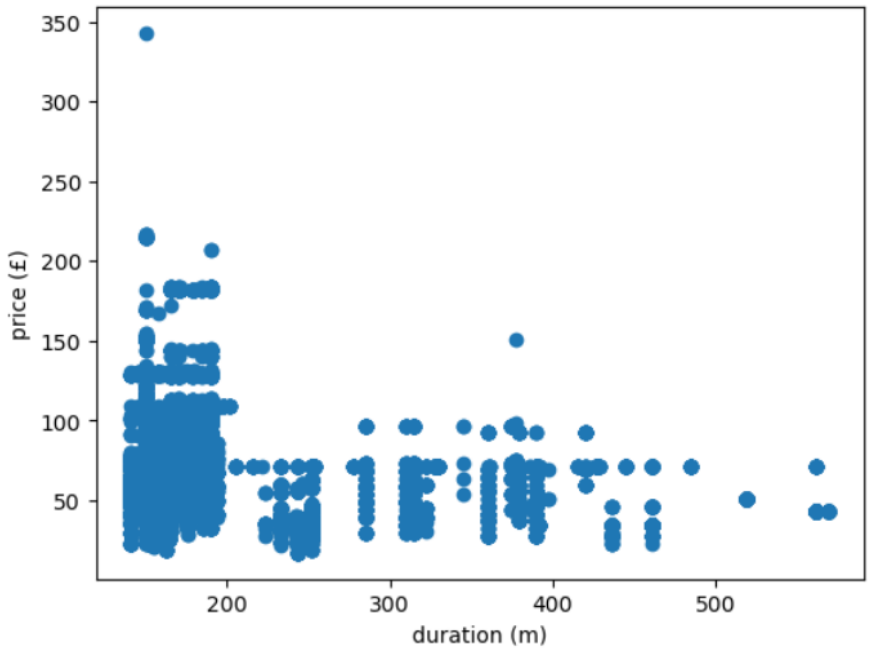
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv('data/train_viz.csv')
```

2. Create a scatter plot of `duration` vs. `price`. Segment by `destination`.

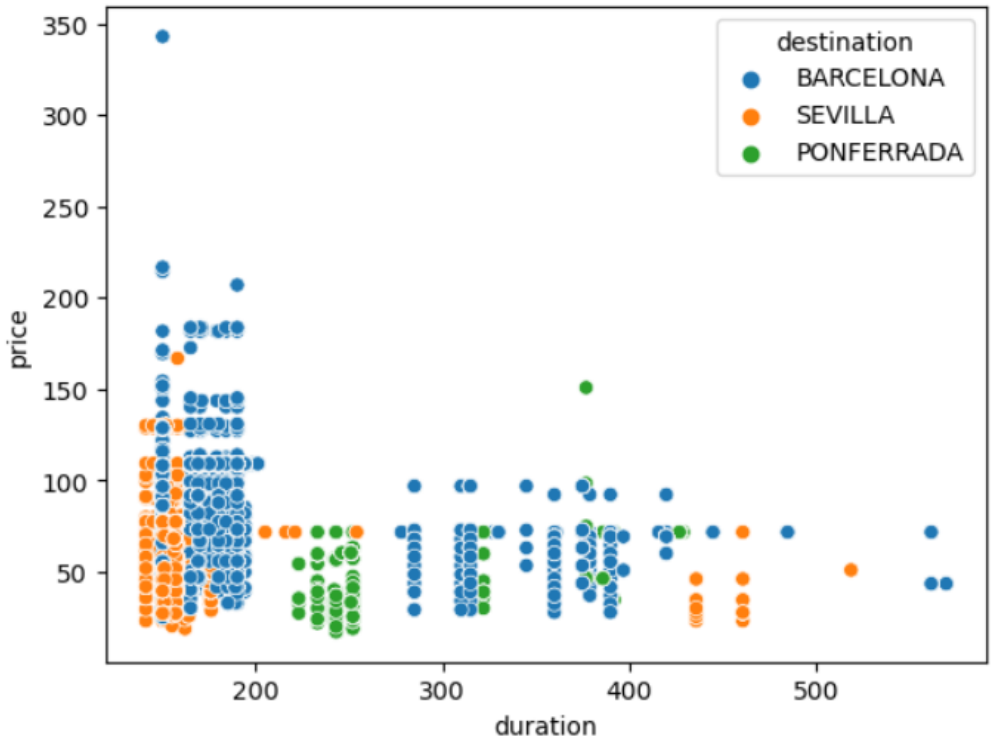
```
plt.scatter(x=df['duration'],
            y=df['price'])
plt.xlabel('duration (m)')
plt.ylabel('price (£)')
```

```
Text(0, 0.5, 'price (£)')
```



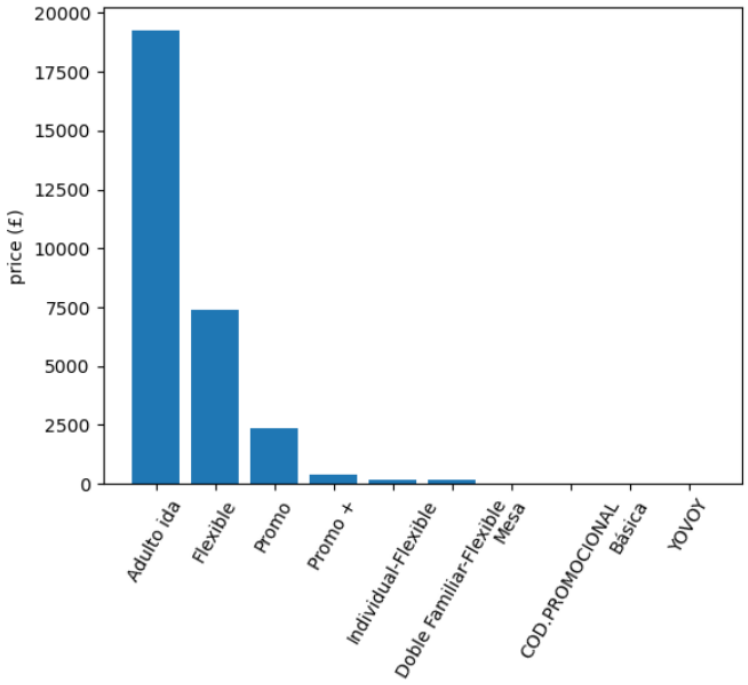
```
sns.scatterplot(data=df,
                x='duration',
                y='price',
                hue='destination')
```

```
<Axes: xlabel='duration', ylabel='price'>
```

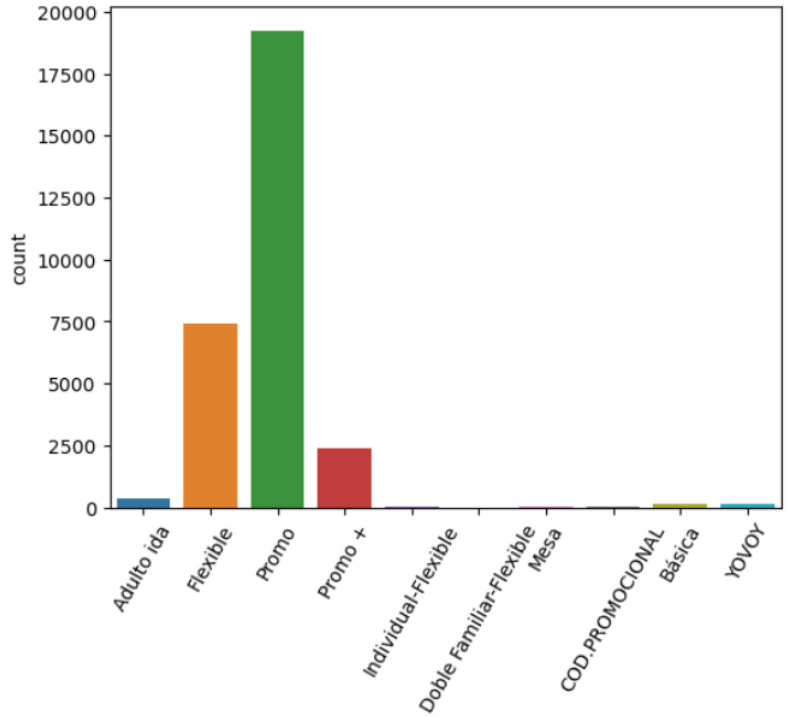


3. Create a barplot of `fare counts`.

```
plt.bar(x=df['fare'].unique(),
        height=df['fare'].value_counts())
plt.ylabel('price (£)')
plt.xticks(rotation=60);
```



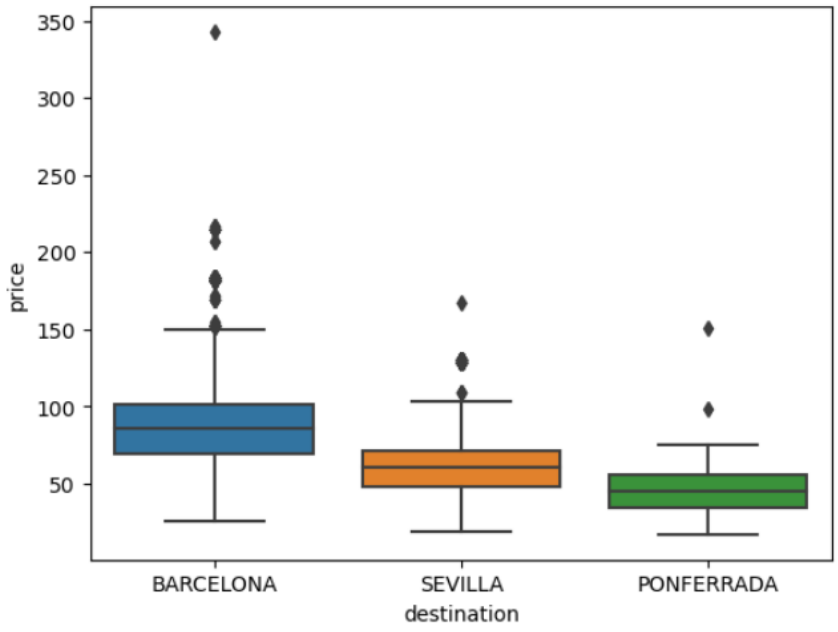

```
sns.countplot(data=df,
              x='fare')
plt.xticks(rotation=60);
```



4. Create a box & whisker plot of `price` with respect to `destination`.

```
sns.boxplot(data=df,
            x='destination',
            y='price')
```

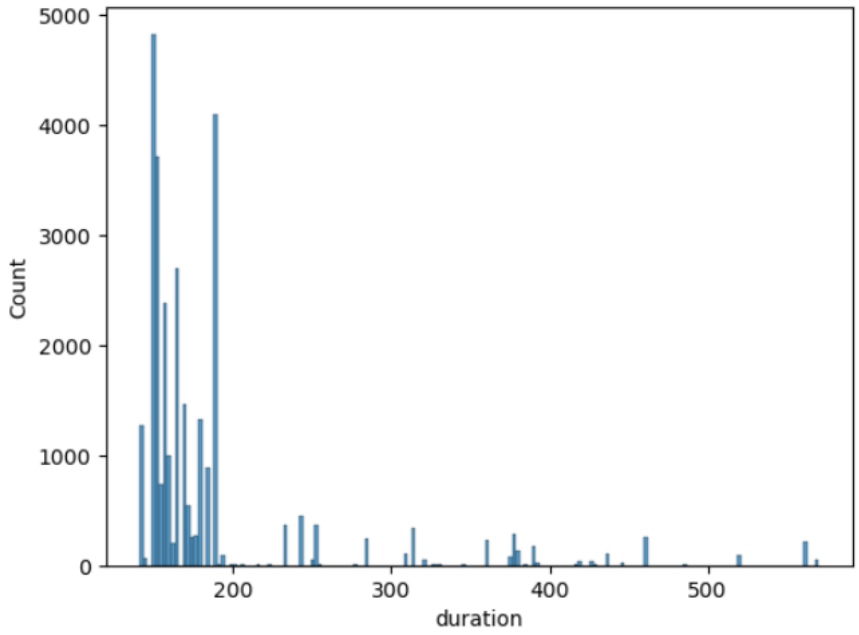
```
<Axes: xlabel='destination', ylabel='price'>
```



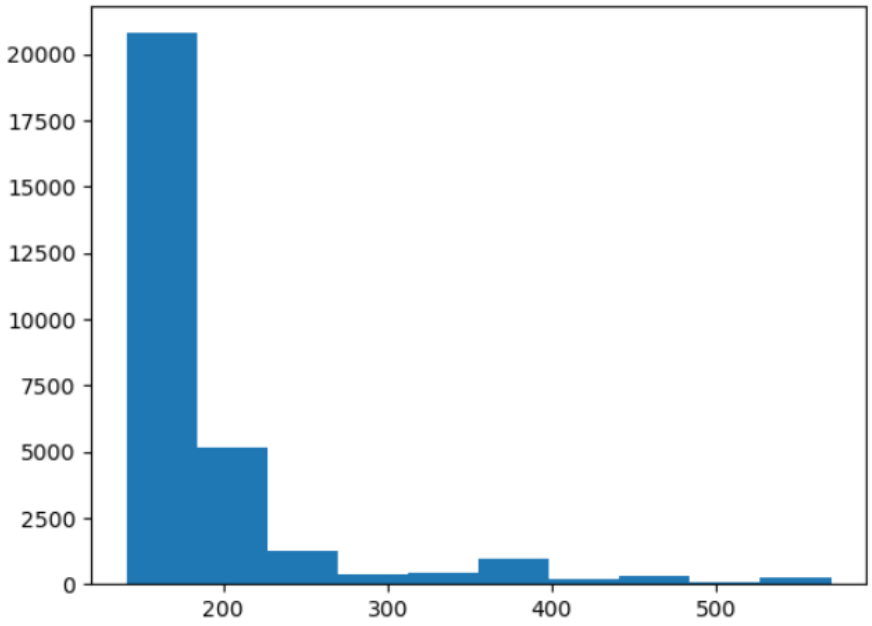
5. Create a histogram of `duration`.

```
sns.histplot(data=df,
             x='duration')
```

```
<Axes: xlabel='duration', ylabel='Count'>
```



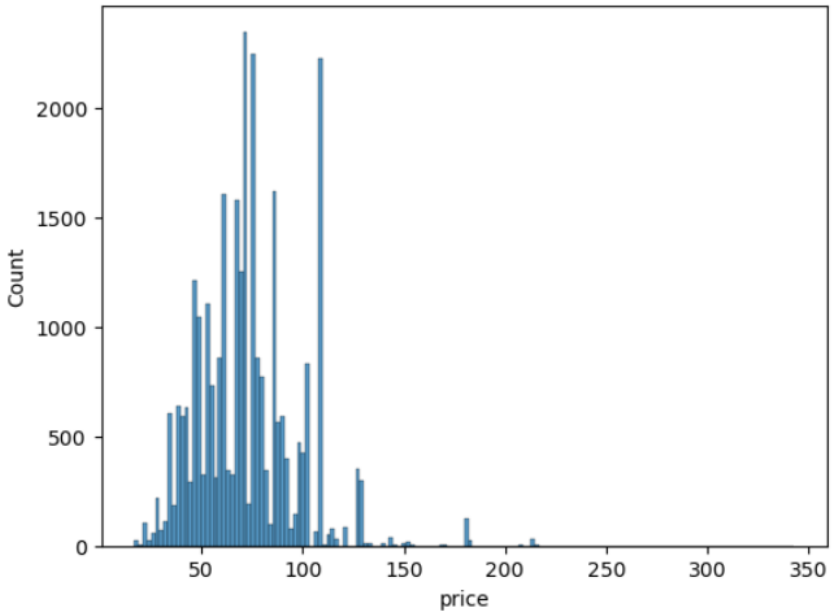

```
plt.hist(df['duration']);
```



6. Compare this to a histogram of `price`.

```
sns.histplot(data=df,
             x='price')
```

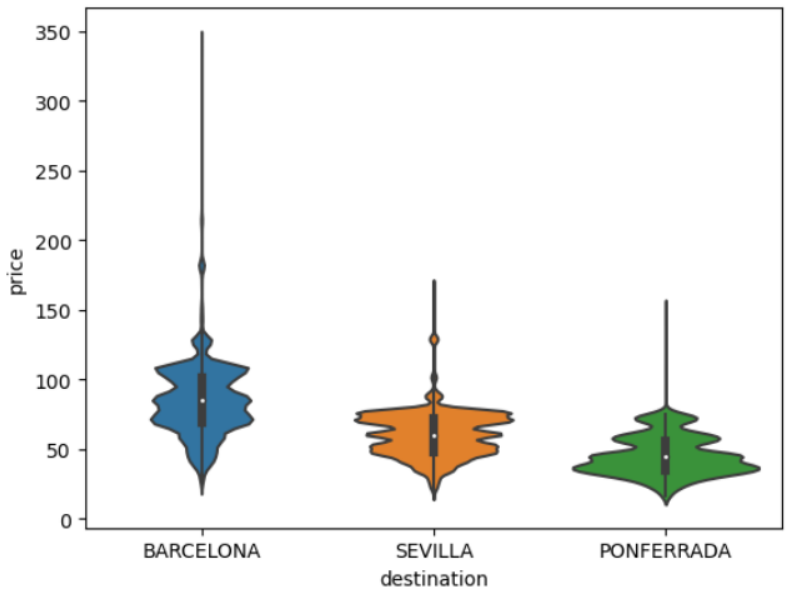
```
<Axes: xlabel='price', ylabel='Count'>
```



7. Create a violin plot of `price` with respect to `destination`.

```
sns.violinplot(data=df,
               x='destination',
               y='price')
```

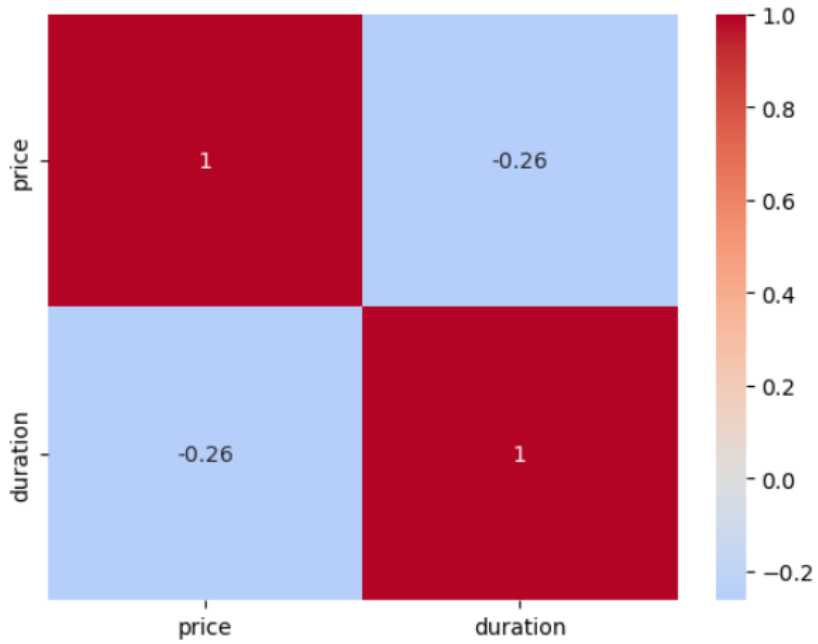
```
<Axes: xlabel='destination', ylabel='price'>
```



8. Create a heatmap of the correlation between columns.

```
sns.heatmap(df.corr(numeric_only=True),
            cmap='coolwarm',
            annot=True,
            center=0.0)
```

```
<Axes: >
```



9. Create a proportional stacked bar plot of `vehicle_class` by `destination`.

```
fig, ax = plt.subplots(figsize=(8, 5))
table = pd.crosstab(df['destination'], df['vehicle_class'])
table.div(table.sum(1).astype(float), axis=0).plot(kind='bar', stacked=True, ax=ax);
```