Introduction to Python Programming for Economics & Finance

Richard Foltyn
University of Glasgow

June 2, 2023

Contents

Preface

This document is intended as an introduction to the Python programming language and was compiled for the MRes/PhD-level course *Introduction to Python Programming for Economics & Finance* given at the University of Glasgow. The focus is on language elements and libraries that are useful for data analysis, statistics, machine learning and numerical computations in general.

The course consists of a collection of interactive notebooks hosted in a Github repository. While the content in this document and in the interactive notebooks is the same, the notebooks have the added benefit that they can be run directly in a browser, allowing you to immediately experiment with the provided code examples.

We omit numerous features of Python which are not of first-order importance to master the topics mentioned above. This includes manipulation of text data, advanced input/output, generators, decorators and object-oriented programming, among others.

The material does not assume any existing knowledge of Python or programming, but moves at a brisk pace compared to other introductory general-purpose Python programming textbooks.

1 Language and NumPy basics

In this unit we start exploring the Python language, covering the following topics:

1. Basic syntax
2. Built-in data types
3. NumPy arrays

1.1 Basic syntax

- Everything after a # character (until the end of the line) is a comment and will be ignored.
- Variable names are case sensitive.
- Whitespace characters matter (unlike in most languages)!
- Python uses indentation (usually 4 spaces) to group statements, for example loop bodies, functions, etc.
- You don't need to add a character to terminate a line, unlike in some languages.
- You can use the `print()` function to inspect almost any object.

[1]:
```
# First example

# create a variable named 'text' that stores the string 'Hello, world!'
text = 'Hello, world!'

# print contents of 'text'
print(text)
```

```
Hello, world!
```

In Jupyter notebooks and interactive command-line environments, we can also display a value by simply writing the variable name.

[2]:
```
text
```

[2]:
```
'Hello, world!'
```

Alternatively, we don't even need to create a variable but can instead directly evaluate expressions and print the result:

[3]:
```
2*3
```

[3]:
```
6
```

This does not print anything in *proper* Python script files that are run through the interpreter, though.

Calling `print()` is also useful if we want to display multiple expressions from a single notebook cell, as otherwise only the last value is shown:

[4]:
```
text = 'Hello world!'
var = 1
text        # does NOT print contents of text
var         # prints only value of var
```

[4]:

```
1
```

[5]:

```python
print(text) # print text explicitly
var         # var is shown automatically
```

```
Hello world!
```

[5]:

```
1
```

1.2 Built-in data types

Pythons is a dynamically-typed language:

- Unlike in C or Fortran, you don't need to declare a variable or its type.
- You can inspect a variable's type using the built-in `type()` function, but you rarely need to do this.

We now look at the most useful built-in data types:

Basic types

- integers (`int`)
- floating-point numbers (`float`)
- boolean (`bool`)
- strings (`str`)

Containers (or collections)

- tuples (`tuple`)
- lists (`list`)
- dictionaries (`dict`)

1.2.1 Integers and floats

Integers and floats (floating-point numbers) are the two main built-in data types to store numerical data (we ignore complex numbers in this tutorial). Floating-point is the standard way to represent real numbers on computers since these cannot store real numbers with arbitrary precision.

[6]:

```python
# Integer variables
i = 1
type(i)
```

[6]:

```
int
```

[7]:

```python
# Floating-point variables
x = 1.0
type(x)
```

[7]:

```
float
```

[8]:

```python
# A name can reference any data type:
# Previously, x was a float, now it's an integer!
x = 1
type(x)
```

[8]:

```
int
```

It is good programming practice to specify floating-point literals using a decimal point. It makes a difference in a few cases (especially when using NumPy arrays, or Python extensions such as Numba or Cython):

[9]:
```
x = 1.0         # instead of x = 1
```

A boolean (`bool`) is a special integer type that can only store two values, `True` and `False`. We create booleans by assigning one of these values to a variable:

[10]:
```
x = True
x = False
```

Boolean values are most frequently used for conditional execution, i.e., a block of code is run only when some variable is `True`. We study conditional execution in the next unit.

1.2.2 Strings

The string data type stores sequences of characters:

[11]:
```
# Strings need to be surrounded by single (') or double (") quotes!
institution = 'University of Glasgow'
institution = "University of Glasgow"
```

Strings support various operations some of which we explore in the exercises at the end of this section. For example, we can use the addition operation `+` to concatenate strings:

[12]:
```
# Define two strings
str1 = 'Python'
str2 = 'course'

# Concatenate strings using +
str1 + ' ' + str2
```

[12]:
```
'Python course'
```

An extremely useful variant of strings are the so-called *f-strings*. These allow us to dynamically insert a variable value into a string, a feature we'll use throughout this course.

[13]:
```
# Approximate value of pi
pi = 3.1415

# Use f-strings to embed the value of the variable version inside the string
s = f'Pi is approximately equal to {pi}'
s
```

[13]:
```
'Pi is approximately equal to 3.1415'
```

1.2.3 Tuples

Tuples represent a collection of several items which can have different data types. They are created whenever several items are separated by commas. The parentheses are optional:

```
(item1, item2, ...)
```

[14]:
```
# A tuple containing a string, an integer and a float
items = ('foo', 1, 1.0)
items
```

```
[14]: ('foo', 1, 1.0)
```

The parenthesis are optional, but improve readability:

```
[15]: items = 'foo', 1, 1.0        # equivalent way to create a tuple
      items
```

```
[15]: ('foo', 1, 1.0)
```

We use brackets [ ] to access an element in a tuple (or any other container object)

```
[16]: first = items[0]            # variable first now contains 'foo'
      first
```

```
[16]: 'foo'
```

Python indices are 0-based, so 0 references the first element, 1 the second element, etc.

```
[17]: second = items[1]           # second element
      second
```

```
[17]: 1
```

Tuples and any other Python collection support automatic unpacking if we want to extract multiple (or all) values at once:

```
[18]: first, second, third = items

      # Print first element
      first
```

```
[18]: 'foo'
```

If we are not interested in extracting all items, we can collect any remaining items using a * as follows:

```
[19]: first, *rest  = items

      # Rest contains a list of all remaining items
      rest
```

```
[19]: [1, 1.0]
```

Tuples are immutable, which means that the items stored in the tuple cannot be changed!

```
[20]: # This raises an error!
      items = 'foo', 1, 1.0
      items[0] = 123
```

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

1.2.4 Lists

Lists are like tuples, except that they can be modified. We create lists using brackets,

```
[item1, item2, ...]
```

```
[21]: # Create list which contains a string, an integer and a float
      lst = ['foo', 1, 1.0]
      lst
```

```
[21]: ['foo', 1, 1.0]
```

Accessing list items works the same way as with tuples

```
[22]: lst[0]                    # print first item
```

```
[22]: 'foo'
```

Lists items can be modified:

```
[23]: lst[0] = 'bar'            # first element is now 'bar'
      lst
```

```
[23]: ['bar', 1, 1.0]
```

Lists are full-fledged objects that support various operations, for example

```
[24]: lst.insert(0, 'abc')      # insert element at position 0
      lst.append(2.0)           # append element at the end
      del lst[3]                # delete the 4th element
      lst
```

```
[24]: ['abc', 'bar', 1, 2.0]
```

The built-in function `len()` returns the number of elements in a list (and any other container object)

```
[25]: len(lst)
```

```
[25]: 4
```

1.2.5 Dictionaries

Dictionaries are container objects that map keys to values.

- Both keys and values can be (almost any) Python objects, even though usually we use strings as keys.
- Dictionaries are created using curly braces: `{key1: value1, key2: value2, ...}`.

For example, to create a dictionary with three items we write

```
[26]: dct = {
          'institution': 'University of Glasgow',
          'course': 'Python course',
          'year': 2023
      }
      dct
```

```
[26]: {'institution': 'University of Glasgow',
       'course': 'Python course',
       'year': 2023}
```

Specific values are accessed using the syntax `dict[key]`:

```
[27]: dct['institution']
```

[27]:
```
'University of Glasgow'
```

We can use the same syntax to either modify an existing key or add a new key-value pair:

[28]:
```
dct['course'] = 'Introduction to Python'        # modify value of existing key
dct['city'] = 'Glasgow'                         # add new key-value pair
dct
```

[28]:
```
{'institution': 'University of Glasgow',
 'course': 'Introduction to Python',
 'year': 2023,
 'city': 'Glasgow'}
```

Moreover, we can use the methods `keys()` and `values()` to get the collection of a dictionary's keys and values:

[29]:
```
dct.keys()
```

[29]:
```
dict_keys(['institution', 'course', 'year', 'city'])
```

[30]:
```
dct.values()
```

[30]:
```
dict_values(['University of Glasgow', 'Introduction to Python', 2023,
'Glasgow'])
```

When we try to retrieve a key that is not in the dictionary, this will produce an error:

[31]:
```
dct['country']
```

```
KeyError: 'country'
```

One way to get around this is to use the `get()` method which accepts a default value that will be returned whenever a key is not present:

[32]:
```
dct.get('country', 'Scotland')      # return 'Scotland' if 'country' is
                                    # not a valid key
```

[32]:
```
'Scotland'
```

1.3 NumPy arrays

NumPy is a library that allows us to efficiently store and access (mainly) numerical data and apply numerical operations similar to those available in Matlab.

- NumPy is not part of the core Python project.
- Python itself has an array type, but there is really no reason to use it. Use NumPy!
- NumPy types and functions are not built-in, we must first import them to make them visible. We do this using the `import` statement.

The convention is to make NumPy functionality available using the `np` namespace:

[33]:
```
import numpy as np
```

1.3.1 Creating arrays

NumPy offers a multitude of functions to create arrays.

[34]:
```
# Create a 1-dimensional array with 10 elements, initialise values to 0.
# We need to prefix the NumPy function zeros() with 'np'
arr = np.zeros(10)
arr
```

[34]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

[35]:
```
arr1 = np.ones(5)       # vector of five ones
arr1
```

[35]:
```
array([1., 1., 1., 1., 1.])
```

We can also create sequences of integers using the `np.arange()` function:

[36]:
```
arr2 = np.arange(5)     # vector [0,1,2,3,4]
arr2
```

[36]:
```
array([0, 1, 2, 3, 4])
```

`np.arange()` accepts initial values and increments as optional arguments. The end value is *not* included.

[37]:
```
start = 2
end = 10
step = 2
arr3 = np.arange(start, end, step)
arr3
```

[37]:
```
array([2, 4, 6, 8])
```

As in Matlab, there is a `np.linspace()` function that creates a vector of uniformly-spaced real values.

[38]:
```
# Create 11 elements, equally spaced on the interval [0.0, 1.0]
arr5 = np.linspace(0.0, 1.0, 11)
arr5
```

[38]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

We create arrays of higher dimension by specifying the desired shape. Shapes are specified as `tuple` arguments; for example, the shape of an $m \times n$ matrix is `(m,n)`.

[39]:
```
mat = np.ones((2,2))    # Create 2x2 matrix of ones
mat
```

[39]:
```
array([[1., 1.],
       [1., 1.]])
```

Creating arrays from other Python objects

Arrays can be created from other objects such as lists and tuples by calling `np.array()`

[40]:
```
# Create array from list [1,2,3]
arr = np.array([1, 2, 3])
arr
```

[40]:
```
array([1, 2, 3])
```

[41]:
```
# Create array from tuple
arr = np.array((1.0, 2.0, 3.0))
arr
```

[41]:
```
array([1., 2., 3.])
```

[42]:
```
# Create two-dimensional array from nested list
arr = np.array([[1, 2, 3], [4, 5, 6]])
arr
```

[42]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

1.3.2 Reshaping arrays

The `reshape()` method of an array object can be used to reshape it to some other (conformable) shape.

[43]:
```
# Create vector of 4 elements and reshape it to a 2x2 matrix
mat = np.arange(4).reshape((2,2))
mat
```

[43]:
```
array([[0, 1],
       [2, 3]])
```

[44]:
```
# reshape back to vector of 4 elements
vec = mat.reshape(4)
vec
```

[44]:
```
array([0, 1, 2, 3])
```

We use `-1` to let NumPy automatically compute the size of *one* remaining dimension.

[45]:
```
# with 2 dimensions, second dimension must have size 2
mat = np.arange(4).reshape((2, -1))
mat
```

[45]:
```
array([[0, 1],
       [2, 3]])
```

If we want to convert an arbitrary array to a vector, we can alternatively use the `flatten()` method.

[46]:
```
mat.flatten()
```

[46]:
```
array([0, 1, 2, 3])
```

Important: the reshaped array must have the same number of elements!

[47]:
```
import numpy as np
mat = np.arange(6).reshape((2,-1))
mat.reshape((2,2))       # Cannot reshape 6 into 4 elements!
```

```
ValueError: cannot reshape array of size 6 into shape (2,2)
```

1.3.3 Indexing

Single element indexing

To retrieve a single element, we specify the element's index on each axis (axis is the NumPy terminology for an array dimension).

- Remember that just like Python in general, NumPy arrays use 0-based indices.
- Unlike lists or tuples, NumPy arrays support multi-dimensional indexing.

[48]:
```
import numpy as np

mat = np.arange(6).reshape((3,2))
mat
```

[48]:
```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

[49]:
```
mat[0,1]    # returns element in row 1, column 2
```

[49]:
```
1
```

It is important to pass multi-dimensional indices as a tuple within brackets, i.e., `[0,1]` in the above example. We could alternatively write `mat[0][1]`, which would give the same result:

[50]:
```
mat[0][1] == mat[0,1]       # don't do this!
```

[50]:
```
True
```

This is substantially less efficient, though, as it first creates a sub-dimensional array `mat[0]`, and then applies the second index to this array.

Index slices

There are numerous ways to retrieve a subset of elements from an array. The most common way is to specify a triplet of values `start:stop:step` called `slice` for some axis:

[51]:
```
# Create a 3x2 matrix
mat = np.arange(6).reshape((2,3))
mat
```

[51]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

[52]:
```
# Retrieve only the first and third columns:
mat[0:2,0:3:2]
```

[52]:
```
array([[0, 2],
       [3, 5]])
```

Indexing with slices can get quite intricate. Some basic rules:

- all tokens in `start:stop:step` are optional, with the obvious default values. We could therefore write `::` to include all indices, which is the same as `:`
- The end value is *not* included. Writing `vec[0:n]` does not include element with index n!
- Any of the elements of `start:stop:step` can be negative.
  - If `start` or `stop` are negative, elements are counted from the end of the array: `vec[:-1]` retrieves the whole vector except for the last element.
  - If `step` is negative, the order of elements is reversed.

```
[53]: vec = np.arange(5)
      # These are equivalent ways to return the WHOLE vector
      vec[0:5:1]      # all three tokens present
      vec[::]         # omit all tokens
      vec[:]          # omit all tokens
      vec[:5]         # end value only
      vec[-5:]        # start value only, using negative index
```

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

You can reverse the order like this:

```
[54]: vec[::-1]
```

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

With multi-dimensional arrays, the above rules apply for each dimension.

- We can, however, omit explicit indices for higher-order dimensions if all elements should be included.

```
[55]: mat[1]      # includes all columns of row 2; same as mat[1,:]
```

```
[55]: array([3, 4, 5])
```

We cannot omit the indices for leading axes, though! If an entire leading axis is to be included, we specify this using `:`

```
[56]: mat[:, 1]   # includes all rows of column 2
```

```
[56]: array([1, 4])
```

Indexing lists and tuples

The basic indexing rules we have covered so far also apply to the built-in `tuple` and `list` types. However, `list` and `tuple` do not support advanced indexing available for NumPy arrays which we study in later units.

```
[57]: # Apply start:stop:step indexing to tuple
      tpl = (1,2,3)
      tpl[:3:2]
```

```
[57]: (1, 3)
```

1.3.4 Numerical data types (advanced)

We can explicitly specify the numerical data type when creating NumPy arrays.

So far we haven't done so, and then NumPy does the following:

- Functions such as `zeros()` and `ones()` default to using `np.float64`, a 64-bit floating-point data type (this is also called *double precision*)
- Other functions such as `arange()` and `array()` inspect the input data and return a corresponding array.
- Most array creation routines accept a `dtype` argument which allows you to explicitly set the data type.

Examples:

[58]:
```python
import numpy as np

# Floating-point arguments return array of type np.float64
arr = np.arange(1.0, 5.0, 1.0)
arr.dtype
```

[58]:
```
dtype('float64')
```

[59]:
```python
# Integer arguments return array of type np.int64
arr = np.arange(1,5,1)
arr.dtype
```

[59]:
```
dtype('int64')
```

Often we don't care about the data type too much, but keep in mind that

- Floating-point has limited precision, even for integers if these are larger than (approximately) 10^{16}
- Integer values cannot represent fractional numbers and (often) have a more limited range.

This might lead to surprising consequences:

[60]:
```python
# Create integer array
arr = np.ones(5, dtype=np.int64)
# Store floating-point in second element
arr[1] = 1.234
arr
```

[60]:
```
array([1, 1, 1, 1, 1])
```

The array is unchanged because it's impossible to represent 1.234 as an integer value!

The take-away is to explicitly write floating-point literal values and specify a floating-point `dtype` argument when we want data to be interpreted as floating-point values. For example, always write 1.0 instead of 1, unless you *really* want an integer!

1.4 Optional exercises

Exercise 1: string operations

Experiment with operators applied to strings and integers.

1. Define two string variables and concatenate them using `+`
2. Define a string variable and multiply it by 2 using `*`. What happens?
3. Define two strings and compare whether they are equal using the `==` and `!=` relational operators.
4. Define a string. Use the operators `in` and `not in` to test whether a character is contained in the string.
5. Define two string variables and assign them the same value. Use the `is` operator to test whether these are identical objects.
6. Define a string variable and use the `+=` assignment operator to append another string.

   The `+=` operator is one of several operators in Python that combine assignment with another operation, such as addition. In this particular case, these statements are equivalent:

   ```
   a += b
   a = a + b
   ```

Exercise 2: string formatting

We frequently want to create strings that incorporate integer and floating-point data, possibly formatted in a particular way.

Python offers quite powerful formatting capabilities which can become so complex that they are called the *Format Specification Mini-Language* (see the docs). In this exercise, we explore a small but useful subset of formatting instructions.

A format specification is a string that contains one or several `{}`, for example:

```
s = 'The current version of Python is {}'
```

The token `{}` will be replaced with data converted to a string when we apply the `format()` method:

```
s = 'The current version of Python is {}'.format(3.10)
```

The string `s` now contains the value

```
'The current version of Python is 3.10'
```

What if we want to format the float `3.10` in a particular way? We can augment the `{}` to achieve that goal. For example, if the data to be formatted is of type integer, we can specify

- `{:wd}` where `w` denotes the total field width and `d` indicates that the data type is an integer.

  To print an integer into a field that is 3 characters wide, we would thus write `{:3d}`.

For floats we have additional options:

- `{:w.df}` specifies that a float should be formatted using a field width `w` and `d` decimal digits.

  To print a float into a field of 10 characters using 5 decimal digits, we would thus specify `{:10.5f}`
- `{:w.de}` is similar, but instead uses scientific notation with exponents.

  This is particularly useful for very large or very small numbers.
- `{:w.dg}`, where `g` stands for *general* format, is a superset of `f` and `e` formatting. Either fixed or exponential notation is used depending on a number's magnitude.

In all these cases the field width `w` is optional and can be omitted. Python then uses however many characters are required.

Now what we have introduced the formatting language, you are asked to perform the following exercises:

1. Define two strings and concatenate them using the `format()` function. Add a space between them.
2. Use the above example format string, but truncate the Python version to its major version number. Do you get the expected result?
3. Print π using a precision of 10 decimal digits. *Hint:* the value of π is available as

   ```
   from math import pi
   ```

4. Print `exp(10.0)` using exponential notation and three decimal digits. *Hint:* To use the exponential function, you need to import it using

   ```
   from math import exp
   ```

Exercise 3: string formatting with f-strings

Since Python 3.6 there is an additional, more convenient way to format strings, the so-called *formatted string literals* or *f-strings* (official documentation). Instead of calling the `format()` method as in the previous example, one can instead define a string which contain expressions that will be evaluated at runtime.

The simplest example is to print the value of a variable using default formatting:

[61]:
```python
name = 'Python'
s = f'{name} programming is fun!'
print(s)
```

```
Python programming is fun!
```

Note that the string needs to be prefixed by an `f` to indicate that it contains expressions which need to be evaluated. These expressions are again wrapped in braces. Within braces, a syntax similar to the one shown in the previous exercise can be used to specify detailed formatting instructions. For example, you can specify the number of decimal digits as follows:

[62]:
```python
value = 1.2345
s = f'Value with 2 significant digits: {value:.2f}'
print(s)
```

```
Value with 2 significant digits: 1.23
```

Note that in the above examples, the variables `name` and `value` need to be known when the f-string is being defined. Otherwise, you'll get the following error:

[63]:
```python
# cannot create f-string using unknown names
s = f'{unknown} is not defined'
```

```
NameError: name 'unknown' is not defined
```

Now that you have seen the basic usage, repeat Exercise 2 using f-strings instead of the `format()` method!

Exercise 4: operations on tuples and lists

Perform the following tasks and examine their results:

1. Create two lists and add them using `+`.
2. Multiply a list by the integer 2.
3. Create a list `list1` and inspect the result of

   ```python
   list1 += ['x', 'y', 'z']
   ```

4. Create a list `list1` and inspect the result of

   ```python
   list1 *= 2
   ```

Repeat steps 1-4 using tuples instead of lists.

Finally, create a list and a tuple and try to add them using `+`. Does this work?

1.5 Solutions

Solution for exercise 1

```
[64]: # 1. string concatenation using addition
      str1 = 'abc'
      str2 = 'xyz'

      # Concatenate two strings using +
      str1 + str2
```

```
[64]: 'abcxyz'
```

```
[65]: # 2. string multiplication by integers
      str1 = 'abc'
      # Repeat string using multiplication!
      str1 * 2
```

```
[65]: 'abcabc'
```

```
[66]: # 3. Test for string equality
      str1 = 'abc'
      str2 = 'xyz'
      str1 == str2
```

```
[66]: False
```

```
[67]: # 3. Test for string inequality
      str1 = 'abc'
      str2 = 'xyz'
      str1 != str2
```

```
[67]: True
```

```
[68]: # 4. Test whether individual character is included in string
      str1 = 'abc'
      'b' in str1
```

```
[68]: True
```

```
[69]: # 4. Test whether individual character is NOT included in string
      str1 = 'abc'
      'x' not in str1
```

```
[69]: True
```

The last two examples illustrate that in Python strings will be interpreted as collections (of characters), just like lists or tuples, if the context requires it. We can therefore apply the `in` operator to test for membership.

```
[70]: # 5. Test for identity
      str1 = 'abc'
      str2 = 'abc'
      str1 is str2
```

```
[70]: True
```

This result should be surprising and is somewhat specific to Python. We would not expect two objects that were created completely independently from each other to be *identical*, i.e., point to the same memory.

Python, however, caches string literals for reasons of efficiency, so it actually does keep only *one* copy of `'abc'` around, irrespective of how many variables containing `abc` are created.

```
[71]: # 6. Append using +=
      str1 = 'abc'
      str1 += 'xyx'        # Append 'xyz' to value in str1, assign result to str1
      str1
```

```
[71]: 'abcxyx'
```

Solution for exercise 2

```
[72]: # 1. String concatenation
      str1 = 'abc'
      str2 = 'xyz'

      # format specification to concatenate two string with a space in between
      fmt = '{} {}'
      fmt.format(str1, str2)
```

```
[72]: 'abc xyz'
```

```
[73]: # 2. Truncate Python version to major version number
      # To do this, we specify 0 decimal digits!
      fmt = 'The current major version of Python is {:.0f}'
      fmt.format(3.10)      # Now this didn't work out as intended :)
```

```
[73]: 'The current major version of Python is 3'
```

This does not work as intended because formatting with zero decimal digits rounds the floating-point number, instead of just truncating the decimal part. We could instead convert the `float` to `int` before applying formatting, since the function `int()` will truncate the fractional part:

```
[74]: version = 3.10
      fmt = 'The current major version of Python is {:d}'
      print(fmt.format(int(version)))
```

```
The current major version of Python is 3
```

```
[75]: # 3. Print pi using 10 decimal digits
      from math import pi
      fmt = 'The first 10 digits of pi: {:.10f}'
      fmt.format(pi)
```

```
[75]: 'The first 10 digits of pi: 3.1415926536'
```

```
[76]: # 4. Print exp(10.0) using three decimal digits and exponential notation
      from math import exp
      fmt = 'exp(10.0) = {:.3e}'
      fmt.format(exp(10.0))
```

```
[76]: 'exp(10.0) = 2.203e+04'
```

Solution for exercise 3

We now repeat exercise 2 using f-strings instead of the `format()` method.

[77]:
```
# 1. String concatenation
str1 = 'abc'
str2 = 'xyz'

# format specification to concatenate two string with a space inbetween
s = f'{str1} {str1}'
print(s)
```

```
abc abc
```

[78]:
```
# 2. Truncate Python version to major version number
# To do this, we specify 0 decimal digits!
version = 3.10
s = f'The current major version of Python is {version:.0f}'
print(s)        # does not work as intended!
```

```
The current major version of Python is 3
```

This does not work as intended because formatting with zero decimal digits rounds the floating-point number, instead of just truncating the decimal part. See the previous exercise for one possible solution.

[79]:
```
# 3. Print pi using 10 decimal digits
from math import pi
s = f'The first 10 digits of pi: {pi:.10f}'
print(s)
```

```
The first 10 digits of pi: 3.1415926536
```

[80]:
```
# 4. Print exp(10.0) using three decimal digits and exponential notation
from math import exp
s = f'exp(10.0) = {exp(10.0):.3e}'
print(s)
```

```
exp(10.0) = 2.203e+04
```

The last example illustrates that f-string expressions can also be function calls, not just variable names!

Solution for exercise 4

List operators

[81]:
```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']

# 1. Adding two lists concatenates the second list to the first
# and returns a new list object
list1 + list2
```

[81]:
```
[1, 2, 3, 'a', 'b', 'c']
```

[82]:
```
# 2. multiplication of list and integer replicates the list!
list1 * 2
```

[82]:
```
[1, 2, 3, 1, 2, 3]
```

[83]:
```
# 3. Extending list in place using +=
list1 += ['x', 'y', 'z']
list1
```

[83]:
```
[1, 2, 3, 'x', 'y', 'z']
```

Note that we cannot append an element to the list that is not a list:

[84]:
```
list1 += 10
```

```
TypeError: 'int' object is not iterable
```

Instead, we need to wrap a singular element to create a list like this:

[85]:
```
list1 += [10]
list1
```

[85]:
```
[1, 2, 3, 'x', 'y', 'z', 10]
```

[86]:
```
# 4. Replicating list in place using *=
list1 *= 2
list1
```

[86]:
```
[1, 2, 3, 'x', 'y', 'z', 10, 1, 2, 3, 'x', 'y', 'z', 10]
```

Tuple operators

[87]:
```
tpl1 = 1, 2, 3
tpl2 = 'a', 'b', 'c'

# 1. Adding two tuples concatenates the second tuple to the first
# and returns a new tuple object
tpl1 + tpl2
```

[87]:
```
(1, 2, 3, 'a', 'b', 'c')
```

[88]:
```
# 2. multiplication of tuple and integer replicates the tuple!
tpl1 * 2
```

[88]:
```
(1, 2, 3, 1, 2, 3)
```

[89]:
```
# 3. Extending tuple in place
tpl1 += ('x', 'y', 'z')
tpl1
```

[89]:
```
(1, 2, 3, 'x', 'y', 'z')
```

It might be surprising that this works since a `tuple` is an immutable object. However, what happens is that the original `tuple` is discarded and the reference `tpl1` now points to a newly created `tuple`.

The same happens when we replicate a `tuple` with `*=`:

[90]:
```
# 4. Replicate tuple in place using *=
tpl1 *= 2
tpl1
```

[90]:
```
(1, 2, 3, 'x', 'y', 'z', 1, 2, 3, 'x', 'y', 'z')
```

Tuple and list operators

We cannot mix tuples and lists as operands!

[91]:
```
lst = [1, 2, 3]
tpl = 'a', 'b', 'c'
```

```
# Cannot concatenate list and tuple!
lst + tpl
```

```
TypeError: can only concatenate list (not "tuple") to list
```

2 Control flow and list comprehensions

In this unit, we continue to explore basic concepts of the Python programming language such as conditional execution and loops.

2.1 Conditional execution

Frequently, we want to execute a code block only if some condition holds. We can do this using the `if` statement:

[1]:
```python
if 2*2 == 4:
    print('Python knows arithmetic!')
```

```
Python knows arithmetic!
```

A few observations:

- Conditional blocks are grouped using indentation (leading spaces). Remember from the previous unit that whitespace matters in Python!
- We write the equality operator using *two* equal signs, `==`. This is to distinguish it from the assignment operator `=`.

We can also add an `else` block that will be executed whenever a condition is false:

[2]:
```python
if 2*2 == 3:
    # this branch will never be executed
    print('Something is fishy here')
else:
    print('Python knows arithmetic!')
```

```
Python knows arithmetic!
```

Finally, we can add more than one conditional branch using the `elif` clause:

[3]:
```python
var = 1
if var == 0:
    print('var is 0')
elif var == 1:
    print('var is 1')
else:
    print('var is neither 0 nor 1')
```

```
var is 1
```

2.1.1 Truth value testing

We already encountered `==` to test whether two values are equal. Python offers many more operators that return either `True` or `False` and can be used to control conditional execution.

Expression	Description
`==`	Equal. Works for numerical values, strings, etc.
`!=`	Not equal. Works for numerical values, strings, etc.
`>`, `>=`, `<`,`<=`	Usual comparison of numerical values
`a is b`, `a is not b`	Test identity. `a is b` is `True` if `a` and `b` are the same object
`a in b`, `a not in b`	Test whether `a` is or is not included in `b` where `b` is a collection
`if obj`, `if not obj`	Most Python objects evaluate to `True` or `False` in an intuitive fashion (see below)

Additionally, there are logical operators that allow us to combine two logical values:

Expression	Description
`a and b`	`True` if both `a` and `b` are `True`
`a or b`	`True` if at least one of `a` or `b` is `True`

Examples:

[4]:
```
# list1 and list2 reference the same object
list1 = [1,2]
list2 = list1
list1 is list2      # objects are identical, returns True
```

[4]:
```
True
```

[5]:
```
# list1 and list2 do NOT reference the same object, but contain
# identical elements.
list2 = list1.copy()
list1 is list2      # returns False
```

[5]:
```
False
```

[6]:
```
# Check if collections contain the same elements
list1 == list2      # returns True
```

[6]:
```
True
```

[7]:
```
# Check whether element is in collection
1 in list1          # returns True
```

[7]:
```
True
```

[8]:
```
# Combine logical expressions using 'and'
1 in list1 and 2 in list1   # returns True
```

[8]:
```
True
```

We can also use the `in` operator to determine whether a key is contained in a dictionary:

[9]:
```
dct = {'institution': 'University of Glasgow'}
'institution' in dct        # prints True
```

[9]:
```
True
```

[10]:
```
# check whether a key is NOT in the dictionary
'course' not in dct          # prints True
```

[10]:
```
True
```

As mentioned above, most objects evaluate to `True` or `False` in an `if` statement:

```
if obj:
    # do something if obj evaluates to True
```

The rules are quite intuitive: an object evaluates to `False` if

- it has a numerical type and is `0` (or `0.0`, or complex `0+0j`)
- it is an empty collection (tuple, list, dictionary, array, etc.)
- it is of logical (boolean) type and has value `False`
- it is `None`, a special built-in value used to denote that a variable does not reference anything.

In most other cases, an expression evaluates to `True`.

Examples:

[11]:
```
# evaluate numerical variable
x = 0.0
if x:
    print('Value is non-zero')
else:
    print('Value is zero')
```

```
Value is zero
```

[12]:
```
# Evaluate boolean variable
flag = True
if flag:
    print('Flag is True')
else:
    print('Flag is False')
```

```
Flag is True
```

The most important exception to the rule that objects intuitively evaluate to `True` or `False` if needed are NumPy arrays:

[13]:
```
import numpy as np

# Create array with 5 zeros
arr = np.zeros(5)
if arr:
    print('true!')
```

```
ValueError: The truth value of an array with more than one element is ambiguous. Use a.any )
  ↪or a.all()
```

As the error message indicates, Python requires you to be more specific about what exactly should be tested.

2.1.2 Conditional expressions

Conditional expression are more compact than conditional statements and can be used to return a value depending on some condition. A conditional expression takes three arguments using the syntax

```
<value if true> if <condition> else <value if false>
```

The value of this expression can be assigned to a variable, passed to a function, etc.

To illustrate, imagine we have the following code:

[14]:
```python
x = 1

# Test whether x is divisible by 2 without remainder using
# the modulo operator %
if (x % 2) == 0:
    var = 'even'
else:
    var = 'odd'
```

This code sets the value of `var` to either `'even'` or `'odd'`, depending on whether `x` is an even or odd integer. We can formulate this more concisely using a conditional expression:

[15]:
```python
var = 'even' if (x % 2) == 0 else 'odd'
```

We can even directly print the value of this expression!

[16]:
```python
x = 1
print('even' if (x % 2) == 0 else 'odd')
```

```
odd
```

2.2 Loops

Whenever we want to iterate over several items, we use the `for` loop. The `for` loop in Python is particularly powerful because it can "magically" iterate over all sorts of data, not just integer ranges.

The standard use-case is to iterate over a set of integers:

[17]:
```python
# iterate over 0,...,3 and print each element
for i in range(4):
    print(i)
```

```
0
1
2
3
```

We use the built-in `range` function to define the sequence of integers over which to loop. As usual in Python, the last element is *not* included. We can explicitly specify the start value and increment using the more advanced syntax `range(start,stop,step)`:

[18]:
```python
# iterate over 1,3
for i in range(1, 4, 2):
    print(i)
```

```
1
3
```

Unlike with some other languages, we can directly iterate over elements of a collection:

[19]:
```python
cities = ('Glasgow', 'Edinburgh', 'St. Andrews')
for city in cities:
    print(city)
```

```
Glasgow
Edinburgh
St. Andrews
```

We could of course alternatively iterate over indices and extract the corresponding element, but there is no need to:

[20]:
```python
# Needlessly complicated and non-Pythonic
for i in range(len(cities)):
    # print city at index i
    print(cities[i])
```

```
Glasgow
Edinburgh
St. Andrews
```

Note that when looping over dictionaries, the default is to iterate over *keys*:

[21]:
```python
dct = {'key1': 'value1', 'key2': 'value2'}
for key in dct:
    print(key)
```

```
key1
key2
```

We can explicitly choose whether to iterate over keys, values, or both:

[22]:
```python
# iterate over keys, same as example above.
for key in dct.keys():
    print(key)
```

```
key1
key2
```

[23]:
```python
# iterate over values
for value in dct.values():
    print(value)
```

```
value1
value2
```

[24]:
```python
# iterate over keys and values at the same time
# using the items() method
for key, value in dct.items():
    # use format string to print key: value
    print(f'{key}: {value}')
```

```
key1: value1
key2: value2
```

Note that `items()` returns the key-value pairs as `tuples`, so we need to unpack each tuple by writing `key, value` as the running variables of the `for` loop.

Sometimes the set of items over which to iterate is not known ex ante, and then we can instead use the `while` loop with a terminal condition:

[25]:
```python
z = 1.001
i = 0
```

```
# How many iterations will be performed? Not obvious ex ante.
while z < 100.0:
    z = z*z + 0.234
    i = i + 1

# print number of iterations
print(f'loop terminated after {i} iterations')
```

```
loop terminated after 5 iterations
```

2.2.1 Advanced looping

Oftentimes, we want to iterate over a list of items and at the same time keep track of an item's index. We can do this elegantly using the `enumerate()` function:

[26]:
```
cities = ('Glasgow', 'Edinburgh', 'St. Andrews')

# Iterate over cities, keep track of index in variable i
for i, city in enumerate(cities):
    print(f'City {i}: {city}')
```

```
City 0: Glasgow
City 1: Edinburgh
City 2: St. Andrews
```

We can skip an iteration or terminate the loop using the `continue` and `break` statements, respectively:

[27]:
```
for city in cities:
    if city == 'Edinburgh':
        # skip to next iteration in case of Edinburgh
        continue
    print(city)
```

```
Glasgow
St. Andrews
```

[28]:
```
for city in cities:
    if city == 'Glasgow':
        # Terminate iteration as soon as we find Glasgow
        print('Found Glasgow')
        break
```

```
Found Glasgow
```

2.3 List comprehensions

Python implements a powerful feature called *list comprehensions* that can be used to create collections such as tuples and lists without writing loop statements.

For example, imagine we want to create a list of squares of the integers $0, \ldots, 4$. We can do this using a loop and a list's `append()` method:

[29]:
```
# Initialise empty list
squares = []

# Loop over integers 0,...,4
for i in range(5):
    # The power operator in Python is **
    squares.append(i**2)
```

```
squares
```

```
[29]: [0, 1, 4, 9, 16]
```

This is quite bloated and can be collapsed into a single expression using a list comprehension:

```
[30]: squares = [i**2 for i in range(5)]
      squares
```

```
[30]: [0, 1, 4, 9, 16]
```

If the desired result should be a `tuple`, we can instead write

```
[31]: squares = tuple(i**2 for i in range(5))
      squares
```

```
[31]: (0, 1, 4, 9, 16)
```

Alternatively, we can also create a dictionary using curly braces and the syntax `{key: <expression> for ...}`:

```
[32]: squares = {i: i**2 for i in range(5)}
      squares
```

```
[32]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

List comprehensions can be nested and combined with conditions to create almost arbitrarily complex expressions (this doesn't mean that you should, though!)

```
[33]: # Create incomprehensible list comprehension.
      # The modulo operator in Python is %
      items = [i*j for i in range(5) if i % 2 == 0 for j in range(i)]
      items
```

```
[33]: [0, 2, 0, 4, 8, 12]
```

Written out as two nested loops, this code is equivalent to

```
[34]: items = []
      for i in range(5):
          if i % 2 == 0:
              for j in range(i):
                  items.append(i*j)
      items
```

```
[34]: [0, 2, 0, 4, 8, 12]
```

2.4 Optional exercises

These exercises are not meant to demonstrate the most efficient use of Python, but to help you practice the material we have studied above. In fact, you'd most likely *not* want to use the solutions presented here in real code!

Exercise 1: Approximate Euler's number

Euler's number is defined as

$$e = \lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^n$$

1. Create a sequence of the approximations to e for $n = 10, 20, 30, \dots, 100$ using a list comprehension.
2. Compute the approximation error for each of the elements.

   Hint: To get the built-in value for e, use the import statement

   ```
   from math import e
   ```

Exercise 2: Approximate the sum of a geometric series

Let $\alpha \in (0, 1)$. The sum of the geometric series $(1, \alpha, \alpha^2, \dots)$ is given by

$$\sigma = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1 - \alpha}$$

Assume that $\alpha = 0.1$. Write a loop that accumulates the values of the sequence $(1, \alpha, \alpha^2, \dots)$ until the difference to the true value is smaller than 1×10^{-8}. How many elements does it take?

Hint: In Python (and many other languages) the floating-point value 1×10^{-8} is written as `1e-8`.

Exercise 3: Binary search (advanced)

The bisection method can be used to find to root of a function $f(x)$, i.e., the point x_0 such that $f(x_0) = 0$. In this exercise, we will use the same algorithm to find the interval of a strictly monotonic sequence of numbers that brackets the value zero (this is a binary search algorithm with approximate matching).

Assume that we have an array `x` of 11 increasing real numbers given by

```
[35]: import numpy as np
      x = np.linspace(-0.5, 1.0, 11)
      x
```

```
[35]: array([-0.5 , -0.35, -0.2 , -0.05,  0.1 ,  0.25,  0.4 ,  0.55,  0.7 ,
              0.85,  1.  ])
```

Write code that identifies the bracketing interval (in this case [-0.05,0.1]) using the following algorithm:

1. Initialize the index of the bracket lower bound to `lbound=0` and the index of the bracket upper bound to `ubound=len(x)-1`.
2. Compute the midpoint between these two indices (rounded to the nearest integer), `mid = (ubound + lbound) // 2`.

   Hint: The operator `//` truncates the result of a division to the nearest integer.
3. Inspect `x[mid]`, the value at index `mid`. If `x[mid]` has the same sign as `x[ubound]`, update the upper bound, `ubound=mid`. Otherwise, update the lower bound.
4. Continue until `ubound = lbound + 1`, i.e., until you have found the bracket `x[lbound] <= 0 < x[ubound]`.

Exercise 4: Diagonal and band matrices

Let a be a matrix of zeros with shape (m,n) with an integer data type:

```
a = np.zeros((m,n), dtype=int)
```

1. Set m = n = 4. Fill up the diagonal of a with ones so that it becomes an identity matrix. Verify that np.identity() gives the same result.
2. Recreate a to have dimensions m = 4 and n = 5. Modify the main, first lower and first upper diagonals so that the resulting matrix looks like this, where omitted elements are zeros:

$$\begin{bmatrix} 1 & 2 & & & \\ 3 & 1 & 2 & & \\ & 3 & 1 & 2 & \\ & & 3 & 1 & 2 \end{bmatrix}$$

Hint: Use nested for loops to set the diagonal elements.

Exercise 5: Triangular matrices

Let a be a matrix of zeroes with shape (m,n) and integer data type:

```
a = np.zeros((m,n), dtype=int)
```

Assume that m = n = 4. Using loops and if statements, modify the elements of a to obtain the following upper-triangular matrix, where omitted elements are set to zero:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ & 5 & 6 & 7 \\ & & 8 & 9 \\ & & & 10 \end{bmatrix}$$

2.5 Solutions

Solution for exercise 1

[36]:
```python
# Compute approximation for n = 10, 20,..., 100
euler_approx = [(1.0+1.0/i)**i for i in range(10,101,10)]
print('Approximate values')
print(euler_approx)

# import 'correct' value
from math import e

# We need to subtract e from each element to get the approximation error
euler_error = [approx - e for approx in euler_approx]
print('Approximation error')
print(euler_error)
```

```
Approximate values
[2.5937424601000023, 2.653297705144422, 2.6743187758703026, 2.685063838389963,
2.691588029073608, 2.6959701393302162, 2.6991163709761854, 2.7014849407533275,
2.703332461058186, 2.7048138294215285]
Approximation error
[-0.12453936835904278, -0.06498412331462289, -0.043963052588742446,
-0.03321799006908188, -0.026693799385437256, -0.02231168912882886,
-0.019165457482859694, -0.016796887705717634, -0.01494936740085917,
-0.01346799903751661]
```

Solution for exercise 2

We don't know now many iterations we will need to get to the required tolerance of 1×10^{-8}, so this is a good opportunity to use a `while` loop.

[37]:
```python
# Convergence tolerance
tol = 1e-8
alpha = 0.1
# The correct value
sigma_exact = 1.0/(1.0 - alpha)

# keep track of number of iterations
n = 0

# Initialise approximated sum
sigma = 0.0

# Iterate until absolute difference is smaller than tolerance level.
# The built-in function abs() returns the absolute value.

while abs(sigma - sigma_exact) > tol:
    # We can combine addition and assignment into a single operator +=
    # This is equivalent to
    #   sigma = sigma + alpha**n
    sigma += alpha**n
    # Increment exponent
    n += 1

print(f'Number of iterations: {n}, approx. sum: {sigma:.8f}')
```

```
Number of iterations: 9, approx. sum: 1.11111111
```

Solution for exercise 3

To complete the exercise, all you have to do is to translate the algorithm given in the exercise into code.

Since we don't know how many iterations it takes to find the bracket, we use a `while` loop that continues as long as `lbound` and `ubound` are more than 1 apart.

[38]:
```python
import numpy as np

# Given array of increasing numbers
x = np.linspace(-0.5, 1.0, 11)


lbound = 0
ubound = len(x) - 1

while ubound > (lbound + 1):
    # Index of mid point. Indices have to be integers, so
    # we need to truncate the division result to an integer.
    mid = (ubound + lbound) // 2

    if (x[mid] * x[ubound]) > 0.0:
        # x[mid] and x[ubound] have same sign!
        ubound = mid
        print(f'Setting upper bound index to {ubound}')
    else:
        # x[mid] and x[lbound] have the same sign
        # or at least one of them is zero
        lbound = mid
        print(f'Setting lower bound index to {lbound}')

print(f'Value at lower bound: {x[lbound]:.4g}')
print(f'Value at upper bound: {x[ubound]:.4g}')
```

```
Setting upper bound index to 5
Setting lower bound index to 2
Setting lower bound index to 3
Setting upper bound index to 4
Value at lower bound: -0.05
Value at upper bound: 0.1
```

In this implementation, we use the fact that two non-zero real numbers a and b have the same sign whenever $a \cdot b > 0$.

Solution for exercise 4

For the first part, we need to loop over the diagonal elements and overwrite the zeros with ones:

[39]:
```
# Part 1: create identity matrix

import numpy as np

n = 4
a = np.zeros((n,n), dtype=int)

# loop over diagonal elements, set them to 1
for i in range(n):
    a[i,i] = 1

# print identity matrix
a
```

[39]:
```
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1]])
```

You can get the same result using NumPy's `np.identity()` function:

[40]:
```
n = 4
np.identity(n, dtype=int)
```

[40]:
```
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1]])
```

For the second part, we need nested loops over rows and columns and we check at each position (i, j) whether it is on the main diagonal, or on the first upper or lower diagonal.

[41]:
```
# Part 2: create band matrix

import numpy as np

m = 4
n = 5

a = np.zeros((m,n), dtype=int)

# loop over rows
```

```
for i in range(m):
    # loop over columns
    for j in range(n):
        if i == j:
            # main diagonal element
            a[i,j] = 1
        elif i == (j - 1):
            # upper diagonal element
            a[i,j] = 2
        elif i == (j + 1):
            # lower diagonal element
            a[i,j] = 3

# print final matrix
a
```

```
[41]: array([[1, 2, 0, 0, 0],
             [3, 1, 2, 0, 0],
             [0, 3, 1, 2, 0],
             [0, 0, 3, 1, 2]])
```

Solution for exercise 5

We solve this exercise using two nested `for` loops, the first over rows, the second over columns:

```
[42]: import numpy as np

m = 4
n = 4

a = np.zeros((m,n), dtype=int)

# keep track of value to be inserted
value = 1

# loop over rows
for i in range(m):
    # loop over columns
    for j in range(i, n):
        a[i,j] = value
        # increment value for next matching element
        value += 1

# print final matrix
a
```

```
[42]: array([[ 1,  2,  3,  4],
             [ 0,  5,  6,  7],
             [ 0,  0,  8,  9],
             [ 0,  0,  0, 10]])
```

Note that the loop over columns uses the current row index as the lower bound, since we never need to insert anything at element (i, j) if $j < i$.

3 Reusing code: Functions, modules and packages

In this unit, we learn how to build reusable code with functions. We will also briefly discuss modules and packages.

3.1 Functions

Functions are used to implement code that performs a narrowly defined task. We use functions for two reasons:

1. A function can be called repeatedly without having to write code again and again.
2. Even if a function is not called frequently, functions allow us to write code that is "shielded" from other code we write and is called via a clean interface. This helps to write more robust and error-free code.

Functions are defined using the `def` keyword, and the function body needs to be an indented block:

[1]:
```
def func():
    print('func called')

# invoke func without arguments
func()
```

```
func called
```

3.1.1 Arguments

Functions can have an arbitrary number of positional arguments (also called parameters).

[2]:
```
# Define func to accept argument x
def func(x):
    print(f'func called with argument {x}')

# call function with various arguments.
func(1)
func('foo')
```

```
func called with argument 1
func called with argument foo
```

3.1.2 Return values

Functions can also return values to their caller using the `return` statement.

[3]:
```python
def func(x):
    return x * 2.0

result = func(1.0)
print(result)       # prints 2.0
```

```
2.0
```

A `return` statement without any argument immediately exits the functions. The default return value is the special type `None`.

A function can return multiple values which are then automatically collected into a tuple:

[4]:
```python
def func():
    return 1, 2, 3

values = func()         # call func(), get tuple of values
type(values)
```

[4]:
```
tuple
```

Python supports "unpacking" of tuples, lists, etc. We can use this to directly assign names to multiple return values:

[5]:
```python
def func():
    return 'a', 'b', 'c'

value1, value2, value3 = func()     # call func(), unpack return values
print(f'Value 1: {value1}, Value 2: {value2}, Value 3: {value3}')
```

```
Value 1: a, Value 2: b, Value 3: c
```

3.1.3 Accessing data from the outer scope

A function need not have arguments or a return value, but that limits its usefulness somewhat. However, a function can access outside data which is defined in the so-called outer scope:

[6]:
```python
x = 1.0
def func():
    # Read x from outer scope
    print(f'func accessing x from outer scope: x = {x}')

# prints value of x from within func()
func()
```

```
func accessing x from outer scope: x = 1.0
```

We can write functions without any arguments that only operate on outside data. However, this is terrible programming practice and should be avoided in most cases.

Because functions can operate on external data, they are not analogous to mathematical functions. If we write $f(x)$, we usually mean that f is a function of x only (and possibly some constant parameters). By definition we must have

$$x_1 = x_2 \Longrightarrow f(x_1) = f(x_2),$$

i.e., a function always returns the same value when called with the same parameters. However, this is not the case in Python or most other programming languages:

[7]:
```
a = 1.0
def func(x):
    return a*x

x = 1.0
print(func(x))      # prints 1.0

a = 2.0
print(func(x))      # x unchanged, but prints 2.0
```

```
1.0
2.0
```

3.1.4 More on arguments

Default arguments

Python offers an extremely convenient way to specify default values for arguments, so these need not be passed when the function is called:

[8]:
```
# define function with a default value for argument alpha
def func(x, alpha=1.0):
    return x * alpha

print(func(2.0))        # uses default value for alpha
print(func(2.0, 1.0))   # explicitly specified optional argument
print(func(2.0, 3.0))   # use some other value for alpha
```

```
2.0
2.0
6.0
```

Keyword (or named) arguments

Arguments don't need to be provided in the same order as specified in the function signature. We can use argument names to explicitly assign values to the corresponding argument.

[9]:
```
def func(arg1, arg2):
    print(f'arg1: {arg1}, arg2: {arg2}')

func(1, 2)              # Call using purely positional arguments
func(arg1=1, arg2=2)    # Use argument names to explicitly assign values
func(arg2=2, arg1=1)    # With keyword arguments, the order does not matter!
```

```
arg1: 1, arg2: 2
arg1: 1, arg2: 2
arg1: 1, arg2: 2
```

Arbitrary number of optional arguments

Python supports functions which accept an arbitrary number of positional and keyword arguments. This is accomplished via two special tokens:

- `*args`: collects any number of "excess" *positional arguments* and packs them into a tuple.
- `**kwargs`: collects any number of "excess" *keyword arguments* and packs them into a dictionary. It needs to be placed at the end of the argument list!

Examples:

[10]:
```python
# Define function with mandatory, optional, optional positional
# and optional keyword arguments

def func(x, opt='default', *args, **kwargs):
    print(f'Required argument x: {x}')
    print(f'Optional argument opt: {opt}')
    if args:
        # if the tuple 'args' is non-empty, print its contents
        print('Optional positional arguments:')
        for arg in args:
            print(f'  {arg}')
    if kwargs:
        # if the dictionary 'kwargs' is non-empty, print its contents
        print('Optional keyword arguments:')
        for key, value in kwargs.items():
            print(f'  {key}: {value}')
```

[11]:
```python
# Call only with required argument
func(0)
```

```
Required argument x: 0
Optional argument opt: default
```

[12]:
```python
# Call with required and optional arguments
func(0, 'optional')
```

```
Required argument x: 0
Optional argument opt: optional
```

[13]:
```python
# Call with required and optional arguments, and
# optional positional arguments
func(0, 'optional', 1, 2, 3)
```

```
Required argument x: 0
Optional argument opt: optional
Optional positional arguments:
  1
  2
  3
```

[14]:
```python
# Call with required and optional arguments, and
# optional positional and keyword arguments
func(0, 'optional', 1, 2, 3, arg1='value1', arg2='value2')
```

```
Required argument x: 0
Optional argument opt: optional
Optional positional arguments:
  1
  2
  3
Optional keyword arguments:
  arg1: value1
  arg2: value2
```

We don't even need to specify arguments in the order they are defined in the function, except for optional positional arguments, since these have no argument names. We can just use the `name=value` syntax:

[15]:
```python
# call func() with interchanged argument order
func(opt='optional value', x=1)
```

```
Required argument x: 1
Optional argument opt: optional value
```

Note, however, that in a function call any positional arguments must come first and those passed as `name=value` pairs last:

[16]:
```
x = 1

# this will not work, cannot specify positional arguments last
func(opt='optional value', x)
```

```
  Cell In[16], line 4
    func(opt='optional value', x)
                                ^

SyntaxError: positional argument follows keyword argument
```

The same applies for optional arguments passed in via `*args` and `**kwargs`:

[17]:
```
# fails because arguments collected in *args must
# be specified before arguments collected in **kwargs!
func(1.0, 'opt', arg1='value1', arg2='value2', 1, 2, 3)
```

```
  Cell In[17], line 3
    func(1.0, 'opt', arg1='value1', arg2='value2', 1, 2, 3)
                                                          ^

SyntaxError: positional argument follows keyword argument
```

3.1.5 Modifying data in the outer scope

So far, we covered read-only access to data defined outside of a function and relied on return values to pass results back to the caller. However, it is possible to directly *modify* data in the outer scope, even though this should usually be avoided:

- Using arguments and return values clearly defines a function's interface, there are no unpleasant surprises.
- Conversely, if a function starts modifying values in the caller's environment, there is no way to be sure what the function is changing in the outer scope other than by examining its source code.

Consider first the following attempt to modify a value defined outside of the function:

[18]:
```
var = 'outer scope'

# Create function, assign value to var
def modify_var():
    var = 'inner scope'

print(var)
modify_var()
print(var)
```

```
outer scope
outer scope
```

This code prints `'outer scope'` twice. What happened? Without any further instructions, the assignment inside the function creates a *local* variable `var` that is completely disconnected from `var` in the outer scope!

We need to use the `global` statement to tell Python to instead assign to a variable in the global (outer) scope:

[19]:
```python
var  = 'outer scope'

def modify_var():
    global var
    var = 'inner scope'

print(var)
modify_var()
print(var)
```

```
outer scope
inner scope
```

The second output now reads `'inner scope'`.

Note that `global` in Python actually means global to a module, i.e., a symbol that is defined at the top level within a module (we discuss modules below). There are no truly global variables in Python unlike in languages such as C.

The requirement that the name in the `global` statement refers to a global variable has subtle implications. Consider the following example of two *nested* functions:

[20]:
```python
def outer():
    var = 'outer function'

    def inner():
        # Bind var to global name var
        global var
        var = 'inner function'

    print(var)
    inner()
    print(var)

outer()
```

```
outer function
outer function
```

Surprisingly, the code above prints `'outer function'` twice. The reason is that `var` defined in `outer()` is *not* a global variable as it was not defined at the top level within a module. Instead, it is a *local* variable in `outer()`.

For such scenarios, Python has the `nonlocal` statement which works similarly to `global` except that it operates on a name in the immediate outer scope, irrespective of whether this outer scope is another function or the module itself.

We can use `nonlocal` to get the desired behaviour:

[21]:
```python
def outer():
    # var is in outer's local scope
    var = 'outer function'

    def inner():
        # bind var to name in immediate outer scope,
        # which is the local scope of outer()
        nonlocal var
        var = 'inner function'

    print(var)
    inner()
    print(var)

outer()
```

```
outer function
inner function
```

3.1.6 Pass by value or pass by reference?

Can functions modify their arguments? This questions usually comes down to whether a function call uses *pass by value* or *pass by reference*:

- *pass by value* means that a copy of every argument is created before it is passed into the function. A function therefore cannot modify a value in the caller's environment.
- *pass by reference* means that only a reference to a value is passed to the function, so the function can directly modify values at the call site.

This programming model is used in languages such as C (pass by value) or Fortran (pass by reference), but not in Python. Sloppily speaking, we can say that in Python a reference ("variable name") is passed by value. This means assigning a different value to an argument ("the reference") within a function has no effect outside of the function:

[22]:
```
def func(x):
    # x now points to something else
    x = 1.0

x = 123
func(x)

x       # prints 123, x in the outer scope is unchanged
```

[22]:
```
123
```

However, if a variable is a mutable object (such as a `list` or a `dict`), the function can use its own copy of the reference to that object to modify the object even in the outer scope:

[23]:
```
def func(x):
    # uses reference x to modify list object outside of func()
    x.append(4)

lst = [1,2,3]
func(lst)
lst     # prints [1,2,3,4]
```

[23]:
```
[1, 2, 3, 4]
```

Nevertheless, even for mutable objects the rule from above applies: when a new value is *assigned* to a named argument, that name then references a different object, leaving the original object unmodified:

[24]:
```
def func(x):
    # this does not modify object in outer scope,
    # x now references a new (local) object.
    x = ['a', 'b', 'c']

lst = [1,2,3]
# pass list, which is mutable and can thus be changed in func()
func(x)

lst     # prints [1,2,3]
```

```
[24]: [1, 2, 3]
```

For immutable objects such as tuples, the reference passed to the function of course cannot be used to modify the object inside the function:

```
[25]: def func(x):
          x[0] = 'modified in func'

      items = (1, 2, 3)          # create tuple of integers
      func(items)
```

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

Passing in a mutable collection such as a list, however, works as expected:

```
[26]: items = [1, 2, 3]

      func(items)

      items
```

```
[26]: ['modified in func', 2, 3]
```

3.1.7 Methods

Methods are simply functions that perform an action on a particular object which they are bound to. We will not write methods in this course ourselves (they are part of what's called object-oriented programming), but we frequently use them when we invoke actions on objects such as lists:

```
[27]: # Create a list
      lst = [1,2,3]

      # append() is a method of the list class and can be invoked
      # on list objects.
      lst.append(4)
      lst
```

```
[27]: [1, 2, 3, 4]
```

3.1.8 Functions as objects

Functions are objects in their own right, which means that you can perform various operations with them:

- Assign a function to a variable.
- Store functions in collections.
- Pass function as an argument to other functions.

Examples:

```
[28]: def func1(x):
          print(f'func1 called with argument {x}')

      def func2(x):
          print(f'func2 called with argument {x}')
```

```
# List of functions
funcs = [func1, func2]

# Assign functions to variable f
for f in funcs:
    # call function referenced by f
    f('foo')
```

```
func1 called with argument foo
func2 called with argument foo
```

[29]:
```
# Pass one function as argument to another function
func1(func2)
```

```
func1 called with argument <function func2 at 0x7f668ddbf5b0>
```

3.1.9 lambda expressions

You can think of lambda expressions as light-weight functions. The syntax is

```
lambda x: <do something with x>
```

The return value of a lambda expression is whatever its body evaluates to. There is no need (or possibility) to explicitly add a `return` statement.

One big difference to regular functions is that lambda expressions are *expressions*, not statements.

- At this point we gain little from a technical discussion on *statements* vs *expressions*. Loosely speaking, statements are one level above expressions in the Python syntax hierarchy, and the language puts restrictions on where statements can appear. Function definitions, `for` and `while` loops, and `if/elif/else` blocks are statements, among others.
- Conversely, *expressions* are more flexible and can appear basically anywhere. They usually evaluate to some object that can be assigned, passed to a function, etc., whereas statements usually can't.

The take-away is that we can fiddle in lambda expressions almost anywhere, even as arguments in function calls!

For example, we might have a function that applies some algebraic operation to its arguments, and the operation can be flexibly defined by the caller.

[30]:
```
def func(items, operation=lambda z: z + 1):
    # default operation: increment value by 1
    result = [operation(i) for i in items]
    return result

numbers = [1.0, 2.0, 3.0]
# call with default operation
func(numbers)              # prints [2.0, 3.0, 4.0]
```

[30]:
```
[2.0, 3.0, 4.0]
```

[31]:
```
# We can also use lambda expressions to specify
# an alternative operation directly in the call!

func(numbers, lambda x: x**2.0)     # prints [1.0, 4.0, 9.0]
```

[31]:
```
[1.0, 4.0, 9.0]
```

While we could have defined the operation using a "regular" function statement, this is shorter.

3.2 Modules and packages

3.2.1 Modules

Modules allow us to further encapsulate code that implements some particular functionality.

- Each Python file (with the extension `.py`) automatically corresponds to a module of the same name.
- Objects defined within such a module are by default not visible outside of the module, thus helping to avoid naming conflicts.

To actually demonstrate the usage of modules, we need to use files outside of this notebook. To this end, there is an additional Python file in the current directory:

```
lectures/
    unit03_mod.py
```

The module `unit03_mod.py` contains the following definitions:

```
# Contents of unit03_mod.py

# global variable in this module
var = 'Variable defined in unit03_mod'

# global function in this module
def func():
    print(f'func in module unit03_mod called')
```

Module search path

Before getting into the details, we first need to verify that we can import the module `unit03_mod` using the `import` statement:

[32]:
```
import unit03_mod
```

Depending on where exactly you are running this code, the above import statement might fail with a `ModuleNotFoundError` (if no error was raised you can skip the rest of this section). This happens whenever the directory in which the module resides is not in the *module search path* used by Python.

To fix this, check the module search path as follows:

[33]:
```
import sys
sys.path
```

[33]:
```
['/home/richard/repos/teaching/python-intro-PGR/lectures',
 '/home/richard/.conda/envs/python-intro-PGR/lib/python310.zip',
 '/home/richard/.conda/envs/python-intro-PGR/lib/python3.10',
 '/home/richard/.conda/envs/python-intro-PGR/lib/python3.10/lib-dynload',
 '',
 '/home/richard/.local/lib/python3.10/site-packages',
 '/home/richard/.conda/envs/python-intro-PGR/lib/python3.10/site-packages',
 '/home/richard/.conda/envs/python-intro-PGR/lib/python3.10/site-
packages/PyQt5_sip-12.11.0-py3.10-linux-x86_64.egg']
```

If the `lectures/` directory is not included in this list, you can add it manually. For example, this notebook is executed in the git repository's root directory, you need to exectute

[34]:
```
import sys
# add lectures/ directory using a relative path
sys.path.append('./lectures/')
```

Note: Loading custom modules that reside in the GitHub repository currently does *not* work if you opened this notebook in Google Colab.

Importing symbols

We now want to use `func` and `var` in our notebook. However, by default these symbols are not visible and first need to be imported. We can do this in several ways:

1. We can import the module and use fully qualified names to reference objects from `unit03_mod`.
2. We can select which names from `unit03_mod` should be directly accessible.

The first variant looks like this:

[35]:
```
import unit03_mod

# Access variable defined in unit03_mod
print(unit03_mod.var)

# Call function defined in unit03_mod
unit03_mod.func()
```

```
Variable defined in unit03_mod
func in module unit03_mod called
```

If a symbol from `unit03_mod` is used frequently, we might want to make it accessible without the `unit03_mod` prefix. This is the second variant:

[36]:
```
from unit03_mod import var, func

# Access variable defined in unit03_mod
print(var)

# Call function defined in unit03_mod
func()
```

```
Variable defined in unit03_mod
func in module unit03_mod called
```

What if our notebook itself defines a function `func()` which would overwrite the reference to the one imported from `unit03_mod`, as in the following example?

[37]:
```
from unit03_mod import func

# Calls func() defined in unit03_mod
func()

# overwrites definition from unit03_mod with local version
def func():
    print('func in notebook called')

# Calls func() defined in notebook
func()
```

```
func in module unit03_mod called
func in notebook called
```

In such a scenario, we can assign aliases to imported symbols using `as`:

[38]:
```
from unit03_mod import func as imported_func     # The function formerly known
                                                 # as func is now imported_func

def func():
    print('func in notebook called')
```

```
# call our own func
func()

# call func from module unit03_mod
imported_func()
```

```
func in notebook called
func in module unit03_mod called
```

We can even alias the module name itself, as we frequently do with widely used modules such as `numpy`:

```
[39]: import unit03_mod as u3m

      u3m.func()    # call function from module unit03_mod
```

```
func in module unit03_mod called
```

```
[40]: # universal convention to import numpy like this
      import numpy as np
```

3.2.2 Packages

Packages are roughly speaking collections of modules and a little magic on top. We will not be creating packages, but we have already been using them: basically everything besides the built-in functions is defined in some package. For example, the NumPy library is a collection of packages.

3.3 Optional exercises

Exercise 1: Sign function

Implement a function `sign` which returns the following values:

$$sign(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

Test your function on a negative, zero and positive argument.

Exercise 2: Sum of arbitrary number of elements

Create a function called `my_sum` which accepts an arbitrary number of arguments (possibly zero) and returns their sum. Assume that all arguments are numeric.

Test your function with the following arguments:

```
my_sum(10.0)     # one argument
my_sum(1,2,3)    # multiple arguments
my_sum()         # no arguments
```

Make sure that in the last case your function returns zero, which is the sum over an empty set.

Exercise 3: Fibonacci sequence

A classical introductory exercise to programming is to write a function that returns the first n terms of the Fibonacci sequence. The i-th element of this sequence is the integer x_i defined as

$$x_i = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ x_{i-1} + x_{i-2} & \text{else} \end{cases}$$

Write a function `fibonacci(i)`,

```
def fibonacci(i):
    ...
```

which returns the i-th item in the sequence using recursion. A recursive function is a function that calls itself to perform (part of) its task, i.e., you should compute x_i like this:

```
xi = fibonacci(i-1) + fibonacci(i-2)
```

Use this function to compute the first 10 elements of this sequence with a list comprehension.

Exercise 4: Factorials

1. Implement a function that computes the factorial of a non-negative integer n defined as $n! = \prod_{i=1}^{n} i$. Keep in mind that this definition implies that $0! = 1$. Use the list comprehension syntax to create a tuple that contains the factorials for the integers $n = 1, \dots, 10$.

   Hint: The factorial can be written as a recurrence relation $n! = n \cdot (n-1)!$, which you can use to implement the recursive function.

2. Provide an alternative implementation that does not rely on recursion, but instead uses NumPy's `prod()` function to compute the product of a sequence of numbers. Again, create a `tuple` that contains the factorials for the integers $n = 1, \dots, 10$ using a list comprehension.

   Hint 1: To compute the product of the integers $i, i+1, \dots, j$, you can use `np.prod(range(i,j+1))`.

   Hint 2: The product of an empty set is 1, which is what `np.prod()` returns.

Exercise 5: Bisection root-finding algorithm (advanced)

We revisit the binary search algorithm from unit 2, this time applied to finding the root of a continuous function. This is called the bisection method.

Implement a function `bisection(f, a, b, tol, xtol)` which finds the root of the function $f(x)$, i.e., the value x_0 where $f(x_0) = 0$, on the interval $[a, b]$. Assume that $a < b$ and that the function values $f(a)$ and $f(b)$ have opposite signs.

Test your implementation using the function $f(x) = x^2 - 4$ on the interval $[-3, 0]$, which has a (unique) root at $x_0 = -2$.

Hint: The bisection algorithm proceeds as follows:

1. Define tolerance levels $\epsilon > 0$ and $\epsilon_x > 0$. The algorithm completes successfully whenever we have either $|f(x_0)| < \epsilon$ or $|b - a| < \epsilon_x$.
2. Main loop of the algorithm:
   1. Compute the midpoint $x_m = (a+b)/2$
   2. Compute function value $f_m = f(x_m)$
   3. If either $|f_m| < \epsilon$ or $|b - a| < \epsilon_x$, accept x_m as the solution and exit.
   4. Otherwise, update either a or b:

   1. If $sign(f(b)) = sign(f_m)$, set $b = x_m$
      Hint: One way to check whether two non-zero values have the same sign is to check if $f(b) \cdot f_m > 0$.
   2. Otherwise, $a = x_m$
5. Proceed to next iteration of main loop.

Exercise 6: Root-finding with SciPy

In the previous exercise, you were asked to implement your own root-finder based on the bisection algorithm. This will rarely be necessary in real applications since libraries such as SciPy implement ready-to-use root-finding algorithms in the `scipy.optimize` package for you. In this exercise, we explore how to use these routines.

Assume we have a function of a single scalar variable, $f(x)$,

```
def fcn(x):
    # Compute function value fx = f(x)
    return fx
```

which has a root on the interval $[a, b]$. We can use SciPy's implementation of the bisection algorithm `bisect()` as follows:

```
from scipy.optimize import bisect

root = bisect(fcn, a, b)
```

Define a Python function $f(x) = x^2 - 4$ and use SciPy's `bisect()` to locate the root on the interval $[-3, 0]$. Print the root of $f(x)$.

Exercise 7: Minimisation with SciPy

In addition to the root-finding routines discussed in the previous exercise, the `scipy.optimize` package also includes numerous functions to perform minimisation. We demonstrate one such application here.

For a function $f(x)$ of a scalar variable x, we can use the function `minimize_scalar()` to perform the minimization as follows:

```
from scipy.optimize import minimize_scalar

# Call minimizer with custom function fcn
result = minimize_scalar(fcn)

# Print minimisation result
print(f'Minimum found at {result.x}')
```

For this to work, we first need to define the objective function `fcn` or pass a lambda expression. Importantly, `minimize_scalar()` returns an object of type `OptimizeResult` which contains information about the minimisation process. For our purposes, the most relevant attribute is `x` which can be used to recover the minimum, as illustrated above. The function value at the minimum is stored in the attribute `fun`.

Consider the function $f(x) = (x - 2)^2 + 1$. Use SciPy's `minimize_scalar()` to find the (unique) global minimum of this function numerically and print the minimum (the minimum is located at $x = 2$ and can easily be found analytically in this case).

Exercise 8: Maximisation of multivariate functions with SciPy

Frequently, we want to maximize a multivariate function that depends on more than one scalar argument. To fix ideas, assume that we have an objective function given by $f(x_1, x_2; a, b)$ where x_1 and x_2 are the arguments while a and b are additional parameters that are assumed constant. To find the maximum of such a function, we proceed as follows:

1. We use SciPy's `minimize()` which can handle functions with multiple arguments. These arguments have to be passed in as a vector, i.e., the objective has the form $f(\mathbf{x})$ with $\mathbf{x} \in \mathbb{R}^n$.
2. Because SciPy routines all perform minimisation whereas we are interested in maximising the function $f(\mathbf{x})$, we need to return the *negative* value of our objective function.
3. If $f(\bullet)$ requires additional parameters, e.g., $f(\mathbf{x}; a, b)$, we can pass these additional arguments as `args=(a,b)` to `minimize()`.
4. Lastly, `minimize()` requires an initial guess $\mathbf{x}_0$ which can be passed as `x0=...`.

The code fragment below illustrates all of these points.

```
from scipy.optimize import minimize

def fcn(x, a, b):
    # Compute function value f(x, a, b)
    # Return NEGATIVE function value for minimiser
    return - fx

result = minimize(fcn, x0=(0, 0), args=(a, b))
```

Now consider the function $f(x_1, x_2; a, b) = -\left[(x_1 - a)^2 + (x_2 - b)^2\right]$ where $\mathbf{x} = (x_1, x_2)$ is the two-dimensional argument and a and b are additional parameters. Find the maximum of this function for $a = 1$ and $b = -2$ and print both the maximiser $\mathbf{x}$ and the function value at the maximum. Define the function $f(\bullet)$ as shown in the above code fragment, i.e., a and b should be passed as additional arguments.

3.4 Solutions

Solution for exercise 1

[41]:
```
import numpy as np

def sign(x):
    if x < 0.0:
        return -1.0
    elif x == 0.0:
        return 0.0
    elif x > 0.0:
        return 1.0
    else:
        # Argument is not a proper numerical value, return NaN
        # (NaN = Not a Number)
        return np.nan

# Test on a few values
print(sign(-123))
print(sign(0))
print(sign(12345))
```

```
-1.0
0.0
1.0
```

Note that NumPy has a "proper" sign function, `np.sign()`, which implements the same logic but is more robust, accepts array arguments, etc.

Solution for exercise 2

For a function to accept an arbitrary number of elements, we need to declare an `*args` argument.

One possible implementation of `my_sum()` looks as follows:

[42]:
```python
def my_sum(*args):
    # Initialise sum to 0
    s = 0
    for x in args:
        s += x
    return s


# Test with built-in range() object
print(my_sum(10.0))
print(my_sum(1, 2, 3))
print(my_sum())
```

```
10.0
6
0
```

Of course in real code we would use the built-in function `sum()`, or preferably the NumPy variant `np.sum()`:

[43]:
```python
import numpy as np

print(np.sum(10.0))

# Need to pass argument as collection
print(np.sum((1, 2, 3)))

# np.sum() cannot be invoked without arguments, but we can
# call it with an empty tuple ()
np.sum(())
```

```
10.0
6
```

[43]:
```
0.0
```

Solution for exercise 3

The recursive definition of `fibonacci(i)` could look like this:

[44]:
```python
def fibonacci(i):
    if i == 0:
        # No recursion needed
        xi = 0
    elif i == 1:
        # No recursion needed
        xi = 1
    else:
        # Assume that i > 1. We will learn later how to
        # return an error if this is not the case.
        # Use recursion to compute the two preceding values
        # of the sequence.
        xi = fibonacci(i-1) + fibonacci(i-2)
    return xi

# Compute the first 10 elements of the sequence using a list comprehension
first10 = [fibonacci(i) for i in range(10)]
first10
```

```
[44]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Note that this is a terribly inefficient way to compute things, as the same elements of the sequence will needlessly be calculated over and over again.

Also, Python has a built-in recursion limit, so you cannot call a function recursively arbitrarily many times. You can find out what this limit is as follows:

```
[45]: import sys
      print(sys.getrecursionlimit())
```

```
3000
```

Solution for exercise 4

The following code shows a function computing the factorial $n!$ using recursion:

```
[46]: def factorial(n):
          if n == 0:
              return 1
          else:
              # Use recursion to compute factorial
              return n * factorial(n-1)

      fact10 = tuple(factorial(n) for n in range(10))
      fact10
```

```
[46]: (1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880)
```

An implementation without recursion can be created using NumPy's `prod()` function which computes the product of a sequence of numbers:

```
[47]: import numpy as np
      fact10 =  tuple(np.prod(range(1,n+1)) for n in range(10))
      fact10
```

```
[47]: (1.0, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880)
```

Notice that the first element of this sequence is a floating-point value 1.0, while the remaining elements are integers. Why is that? Examine the argument passed to `np.prod()` for `n=0`:

```
[48]: n = 0
      # We have to embed range() in an expression that forces the Python
      # interpreter to actually expand the range object, such as a tuple().
      tuple(range(1,n+1))
```

```
[48]: ()
```

As you see, for `n=0` this is an empty container without elements. The mathematical convention is that the product over an empty set is $\prod_{i \in \emptyset} = 1$, and this is exactly what `np.prod()` returns. However, by default NumPy creates floating-point values, and so the return value is 1.0, not 1.

You can get around this by explicitly specifying the data type using the `dtype` argument, which is accepted by many NumPy functions.

[49]:
```
import numpy as np
# Force result to be of integer type
fact10 =  tuple(np.prod(range(1,n+1), dtype=int) for n in range(10))
fact10
```

[49]:
```
(1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880)
```

Alternatively, we can use `np.arange()` instead of `range()` as the former by default returns integer arrays, even if they are empty:

[50]:
```
import numpy as np
# Force result to be of integer type
fact10 =  tuple(np.prod(np.arange(1,n+1)) for n in range(10))
fact10
```

[50]:
```
(1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880)
```

Finally, you of course would not need to implement the factorial function yourself, as there is one in the `math` module shipped with Python:

[51]:
```
import math
fact10 = tuple(math.factorial(n) for n in range(10))
fact10
```

[51]:
```
(1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880)
```

Solution for exercise 5

Below you find a simple implementation of a bisection algorithm. This function does not perform any error checking and assumes that the initial bracket $[a, b]$ actually contains a root, and that the values $f(a)$ and $f(b)$ have opposite signs.

Note that we impose two termination criteria, and the algorithm will end successfully whenever one of them is satisfied:

1. The function value is sufficiently close to zero, i.e., $|f(x_0)| < \epsilon$ for some small $\epsilon > 0$.
2. The bracket is sufficiently small, i.e., $|b - a| < \epsilon_x$, again for some small $\epsilon_x > 0$

This is standard practice in numerical optimisation since we don't want the algorithm to continue unnecessarily if the desired degree of precision was achieved.

We specify the termination tolerance as optional arguments `tol` and `xtol` with sensible defaults. We also add the maximum permissible number of iterations as an optional argument `maxiter`.

[52]:
```
def bisect(f, a, b, tol=1.0e-6, xtol=1.0e-6, maxiter=100):

    for iteration in range(maxiter):
        # Compute candidate value as midpoint between a and b
        mid = (a + b) / 2.0
        if abs(b-a) < xtol:
            # Remaining interval is too small
            break

        fmid = f(mid)

        if abs(fmid) < tol:
            # function value is close enough to zero
            break
        print(f'Iteration {iteration}: f(mid) = {fmid:.4e}')
        if fmid*f(b) > 0.0:
            # f(mid) and f(b) have the same sign, update upper bound b
            print(f'  Updating upper bound to {mid:.8f}')
            b = mid
        else:
            # f(mid) and f(a) have the same sign, or at least one of
            # them is zero.
            print(f'  Updating lower bound to {mid:.8f}')
            a = mid

    return mid

# Compute root of f(x) = x^2 - 4 on the interval [-3, 0]
# We pass the function f as the first argument, and use a lambda expression
# to define the function directly in the call.
x0 = bisect(lambda x: x**2.0 - 4.0, -3.0, 0.0)

# Print root. The true value is -2.0
x0
```

```
Iteration 0: f(mid) = -1.7500e+00
  Updating upper bound to -1.50000000
Iteration 1: f(mid) = 1.0625e+00
  Updating lower bound to -2.25000000
Iteration 2: f(mid) = -4.8438e-01
  Updating upper bound to -1.87500000
Iteration 3: f(mid) = 2.5391e-01
  Updating lower bound to -2.06250000
Iteration 4: f(mid) = -1.2402e-01
  Updating upper bound to -1.96875000
Iteration 5: f(mid) = 6.2744e-02
  Updating lower bound to -2.01562500
Iteration 6: f(mid) = -3.1189e-02
  Updating upper bound to -1.99218750
Iteration 7: f(mid) = 1.5640e-02
  Updating lower bound to -2.00390625
Iteration 8: f(mid) = -7.8087e-03
  Updating upper bound to -1.99804688
Iteration 9: f(mid) = 3.9072e-03
  Updating lower bound to -2.00097656
Iteration 10: f(mid) = -1.9529e-03
  Updating upper bound to -1.99951172
Iteration 11: f(mid) = 9.7662e-04
  Updating lower bound to -2.00024414
Iteration 12: f(mid) = -4.8827e-04
  Updating upper bound to -1.99987793
Iteration 13: f(mid) = 2.4414e-04
  Updating lower bound to -2.00006104
Iteration 14: f(mid) = -1.2207e-04
  Updating upper bound to -1.99996948
Iteration 15: f(mid) = 6.1035e-05
  Updating lower bound to -2.00001526
Iteration 16: f(mid) = -3.0518e-05
  Updating upper bound to -1.99999237
Iteration 17: f(mid) = 1.5259e-05
  Updating lower bound to -2.00000381
Iteration 18: f(mid) = -7.6294e-06
  Updating upper bound to -1.99999809
Iteration 19: f(mid) = 3.8147e-06
  Updating lower bound to -2.00000095
Iteration 20: f(mid) = -1.9073e-06
```

```
Updating upper bound to -1.99999952
```

[52]:
```
-2.000000238418579
```

Solution for exercise 6

All we need to do is to define the function `fcn` and the interval boundaries `a` and `b` which we then pass to SciPy's `bisect()`.

[53]:
```python
from scipy.optimize import bisect

# Define function whose root should be located
def fcn(x):
    fx = x**2.0 - 4.0
    return fx

# Interval for root-finder
a = -3
b = 1

# Call Scipy's bisect() to do all the work
x0 = bisect(fcn, a, b)

# Print root
print(f'Root is located at {x0}')
```

```
Root is located at -2.0
```

Solution for exercise 7

To find the minimum of the function $f(x) = (x-2)^2 + 1$, we can simply pass a lambda expression to `minimize_scalar()` as follows:

[54]:
```python
from scipy.optimize import minimize_scalar

# Call minimizer with lambda expression
result = minimize_scalar(lambda x: (x-2.0)**2.0 + 1.0)

# Print minimisation result
print(f'Minimum found at {result.x}')
```

```
Minimum found at 1.9999999999999998
```

Solution for exercise 8

It is straightforward to show that the maximum of this function is located at $(1, -2)$ which is what `minimize()` returns:

[55]:
```python
from scipy.optimize import minimize

def fcn(x, a, b):
    # Unpack vector x into scalars x1 and x2
    x1, x2 = x
    # Evaluate function f(x1, x2)
    fx = - ((x1 - a)**2.0 + (x2 - b)**2.0)
    # Return NEGATIVE function value for minimiser
    return - fx
```

```python
# Additional arguments passed to objective
a = 1
b = -2
args = (a, b)

# Initial guess
x0 = (0.0, 0.0)

# Perform maximisation
result = minimize(fcn, x0, args=args)

# Print maximising vector and function at maximum
print(f'Maximum found at {result.x}; f(x) = {result.fun:.3f}')
```

```
Maximum found at [ 0.99999998 -2.00000003]; f(x) = 0.000
```

4 Plotting

In this unit, we take a first look at plotting numerical data. Python itself does not have any built-in plotting capabilities, so we will be using `matplotlib` (MPL), the most popular graphics library for Python.

- For details on a particular plotting function, see the official documentation.
- There is an official introductory tutorial which you can use along-side this unit.

In order to access the functions and objects from matplotlib, we first need to import them. The general convention is to use the namespace `plt` for this purpose:

[1]:
```python
import matplotlib.pyplot as plt
```

Advanced plotting backend for Jupyter notebooks (optional)

When using matplotlib in interactive Jupyter notebooks, we can enable a more fancy plotting backend that allows us to dynamically adjust the zoom, etc. This is done by adding the line

```
%matplotlib widget
```

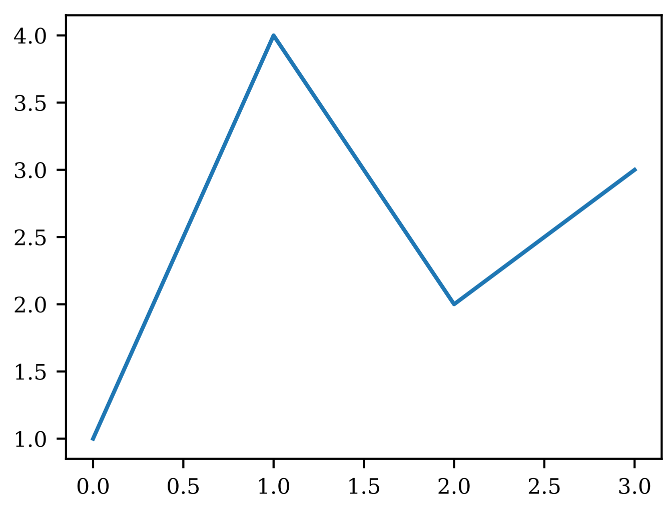
For this to work, the `ipympl` package needs to be installed, see here for details. Note that this is not supported (and in fact produces a syntax error) in regular `*.py` Python script files.

4.1 Line plots

One of the simplest plots we can generate using the `plot()` function is a line defined by a list of *y*-values.

[2]:
```python
# import matplotlib library
import matplotlib.pyplot as plt

# Plot list of integers
yvalues = [1, 4, 2, 3]
plt.plot(yvalues)
```

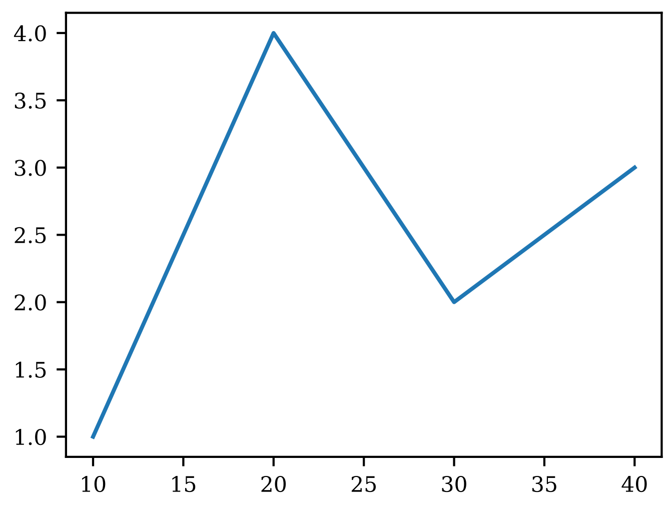
[2]:
```
[<matplotlib.lines.Line2D at 0x7f5fa74a3820>]
```



We didn't even have to specify the corresponding x-values, as MPL automatically assumes them to be [0, 1, 2, . . .]. Usually, we want to plot for a given set of x-values like this:

```
[3]: # explicitly specify x-values
     xvalues = [10, 20, 30, 40]
     plt.plot(xvalues, yvalues)
```

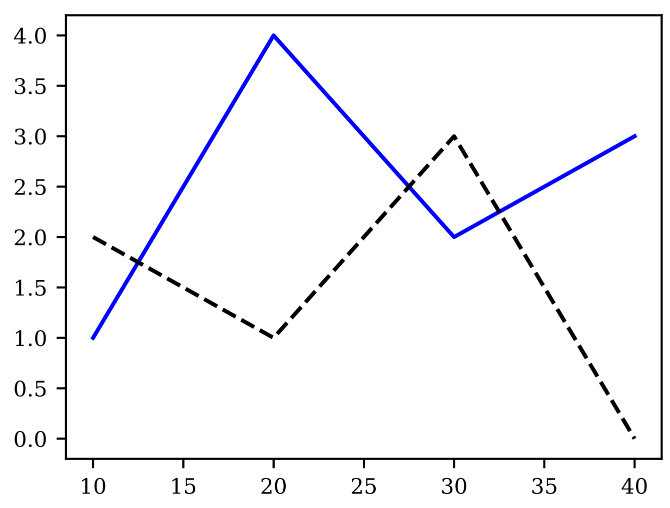
```
[3]: [<matplotlib.lines.Line2D at 0x7f5fa45d76a0>]
```



Similar to Matlab, we can also specify multiple lines to be plotted in a single graph:

```
[4]: yvalues2 = [2.0, 1.0, 3.0, 0.0]
     plt.plot(xvalues, yvalues, 'b-', xvalues, yvalues2, 'k--')
```

```
[4]: [<matplotlib.lines.Line2D at 0x7f5fa4437b50>,
      <matplotlib.lines.Line2D at 0x7f5fa4437c70>]
```



The characters following each set of y-values are style specifications that are very similar to the ones used in Matlab. More specifically, the letters are short-hand notations for colours (see here for details):

- `b`: blue
- `g`: green
- `r`: red
- `c`: cyan
- `m`: magenta
- `y`: yellow
- `k`: black
- `w`: white

The remaining characters set the line styles. Valid values are

- `-` solid line
- `--` dashed line
- `-.` dash-dotted line
- `:` dotted line

Additionally, we can append marker symbols to the style specification. The most frequently used ones are

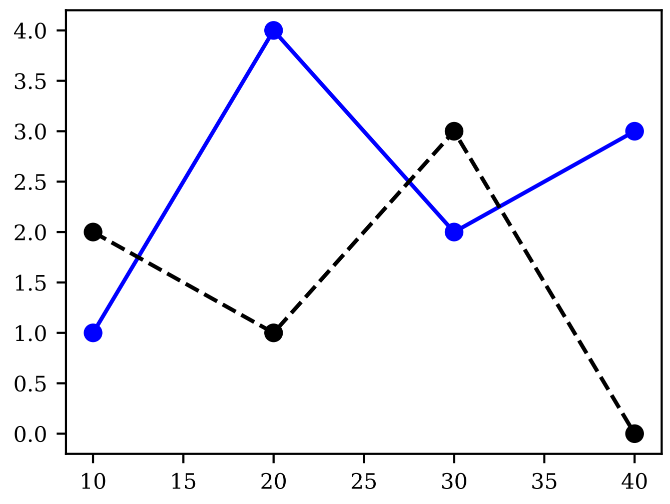
- `o`: circle
- `s`: square
- `*`: star
- `x`: x
- `d`: (thin) diamond

The whole list of supported symbols can be found here.

Instead of passing multiple values to be plotted at once, we can also repeatedly call `plot()` to add additional elements to a graph. This is more flexible since we can pass additional arguments which are specific to one particular set of data, such as labels displayed in legends.

[5]:
```
# Plot two lines by calling plot() twice
plt.plot(xvalues, yvalues, 'b-o')
plt.plot(xvalues, yvalues2, 'k--o')
```

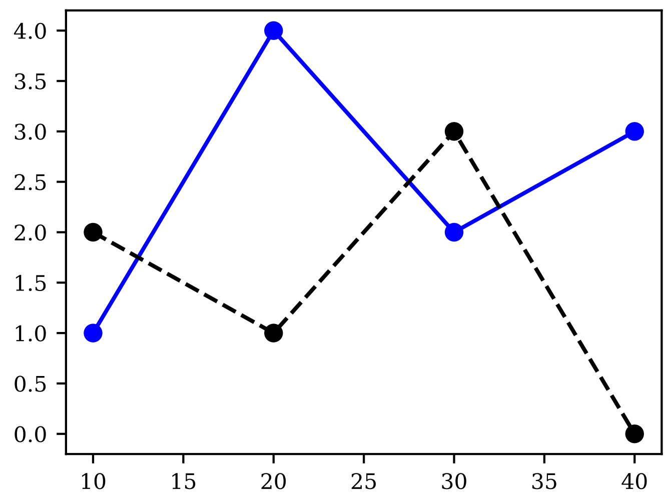
[5]:
```
[<matplotlib.lines.Line2D at 0x7f5fa4468f40>]
```



Individual calls to `plot()` also allow us to specify styles more explicitly using keyword arguments:

[6]:
```python
# pass plot styles as explicit keyword arguments
plt.plot(xvalues, yvalues, color='blue', linestyle='-', marker='o')
plt.plot(xvalues, yvalues2, color='black', linestyle='--', marker='o')
```

[6]:
```
[<matplotlib.lines.Line2D at 0x7f5fa44c25f0>]
```



Note that in the example above, we use named colours such as `red` or `blue` (see here for the complete list of named colors). Alternatively, we can use RGB color codes of the form `#dddddd` where `d` are hexadecimal digits.

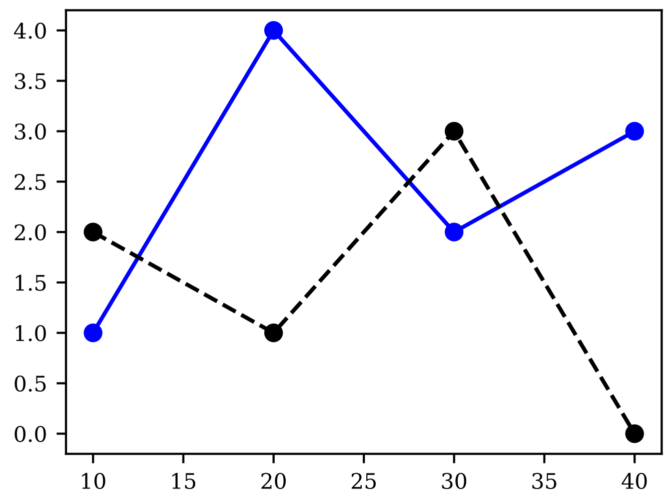
Matplotlib accepts abbreviations for the most common style definitions using the following shortcuts:

- `c` or `color`
- `ls` or `linestyle`
- `lw` or `linewidth`

We can write thus rewrite the above code as follows:

[7]:
```python
# abbreviate plot style keywords
plt.plot(xvalues, yvalues, c='blue', ls='-', marker='o')
plt.plot(xvalues, yvalues2, c='black', ls='--', marker='o')
```

[7]:
```
[<matplotlib.lines.Line2D at 0x7f5fa44f3d30>]
```



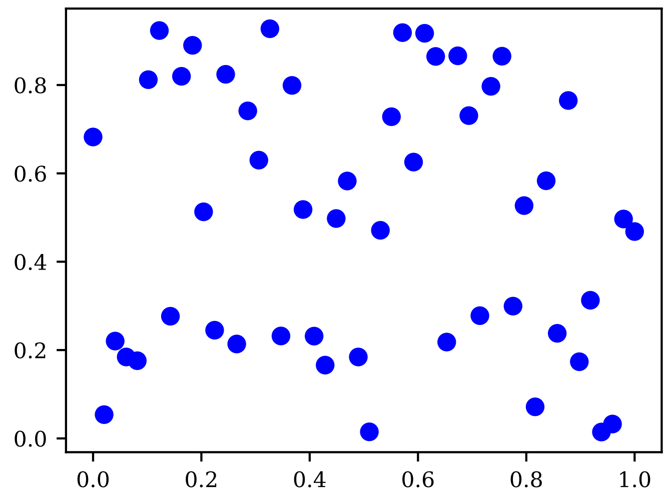
There is a third way to plot multiple lines in the same plot by passing the y-values as a 2-dimensional array. We explore this alternative in the exercises.

4.2 Scatter plots

We use the `scatter()` function to create scatter plots in a similar fashion to line plots:

[8]:
```python
import matplotlib.pyplot as plt
import numpy as np

# Create 50 uniformly-spaced values on unit interval
xvalues = np.linspace(0.0, 1.0, 50)
# Draw random numbers (we learn how to do this in a later unit)
yvalues = np.random.default_rng(123).random(50)

plt.scatter(xvalues, yvalues, color='blue')
```

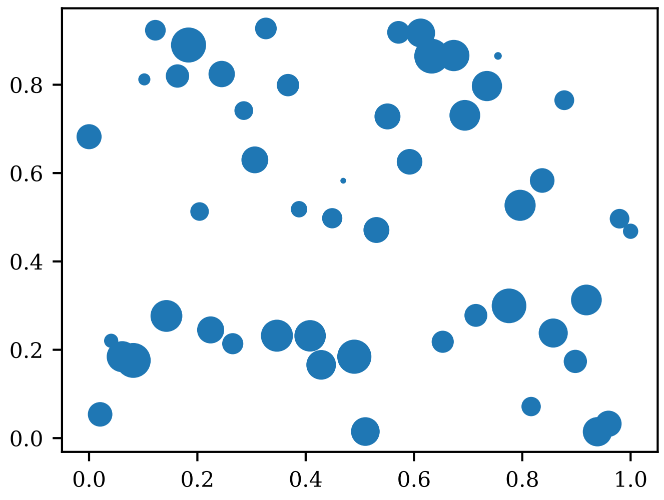
[8]:
```
<matplotlib.collections.PathCollection at 0x7f5fa4370f70>
```



We could in principle create scatter plots using `plot()` by turning off the connecting lines. However, `scatter()` allows us to specify the color and marker size as collections, so we can vary these for every point. `plot()`, on the other hand, imposes the same style on all points plotted in that particular function call.

[9]:
```
# Draw random marker sizes
size = np.random.default_rng(456).random(len(yvalues)) * 150.0

# plot with point-specific marker sizes
plt.scatter(xvalues, yvalues, s=size)
```

[9]:
```
<matplotlib.collections.PathCollection at 0x7f5fa43b32b0>
```



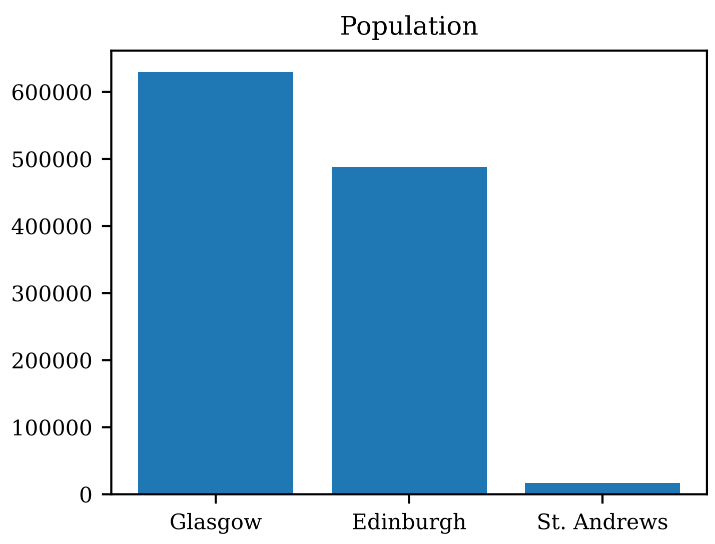
4.3 Plotting categorical data

Instead of numerical values on the *x*-axis, we can also plot categorical variables using the function `bar()`.

For example, assume we have three categories and each has an associated numerical value:

[10]:
```
import matplotlib.pyplot as plt

# Define category labels
cities = ['Glasgow', 'Edinburgh', 'St. Andrews']
population = [630000, 488000,  16800]

# Create bar chart
plt.bar(cities, population)

# Add overall title
plt.title('Population')
```

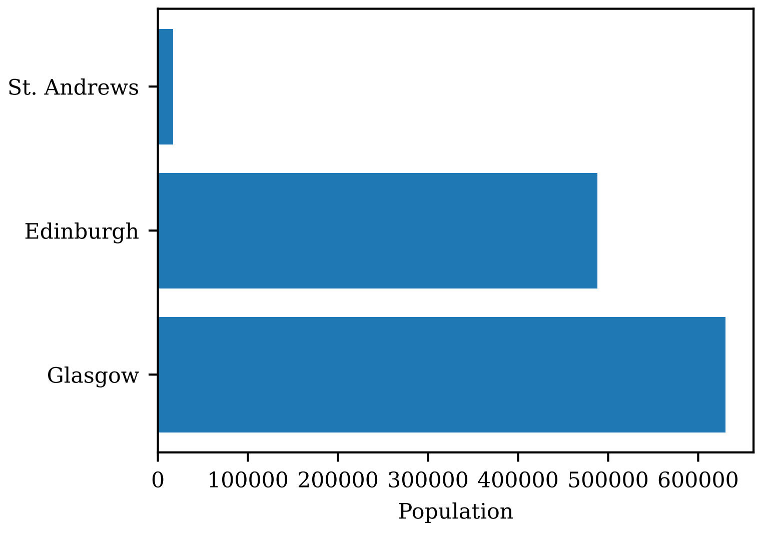
[10]:
```
Text(0.5, 1.0, 'Population')
```



We use `barh()` to create *horizontal* bars:

```
[11]: plt.barh(cities, population)
      plt.xlabel('Population')
```

```
[11]: Text(0.5, 0, 'Population')
```



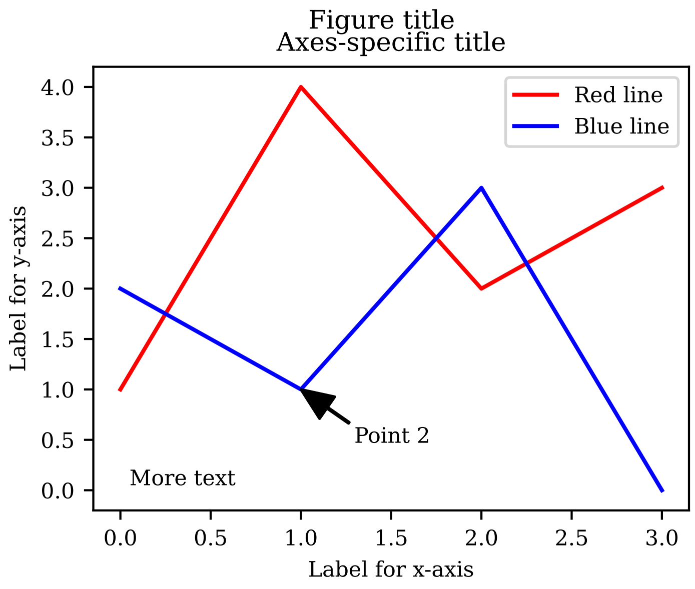
4.4 Adding labels and annotations

Matplotlib has numerous functions to add labels and annotations:

- Use `title()` and `suptitle()` to add titles to your graphs. The latter adds a title for the whole figure, which might span multiple plots (axes).
- We can add axis labels by calling `xlabel()` and `ylabel()`.
- To add a legend, call `legend()`, which in its most simple form takes a list of labels which are in the same order as the plotted data.
- Use `text()` to add additional text at arbitrary locations.
- Use `annotate()` to display text next to some data point; it's easier to position correctly than `text()` and you can add arrows!

[12]:
```
import matplotlib.pyplot as plt

xvalues = [0, 1, 2, 3]
yvalues = [1, 4, 2, 3]
yvalues2 = [2.0, 1.0, 3.0, 0.0]

plt.plot(xvalues, yvalues, 'r', xvalues, yvalues2, 'b')
plt.suptitle('Figure title')
plt.title('Axes-specific title')
plt.xlabel('Label for x-axis')
plt.ylabel('Label for y-axis')
plt.legend(['Red line', 'Blue line'])

# Adds text at data coordinates (0.05, 0.05)
plt.text(0.05, 0.05, 'More text')

# Annotate second point
plt.annotate('Point 2',
   (xvalues[1], yvalues2[1]), (20, -20),
   textcoords='offset points',
   arrowprops={'facecolor': 'black', 'width': 0.5, 'headwidth': 10.0}
)
```

[12]:
```
Text(20, -20, 'Point 2')
```



4.5 Plot limits, ticks and tick labels

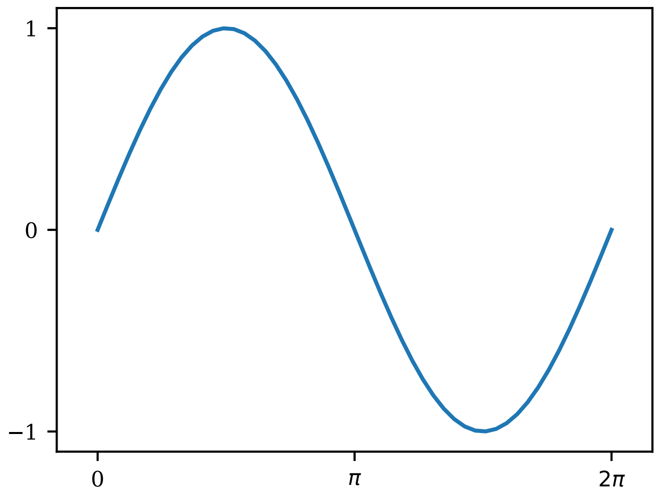
We adjust the plot limits, ticks and tick labels as follows:

- Plotting limits are set using the `xlim()` and `ylim()` functions. Each accepts a tuple `(min,max)` to set the desired range.
- Ticks and tick labels can be set by calling `xticks()` or `yticks()`.

[13]:
```
import matplotlib.pyplot as plt
import numpy as np

xvalues = np.linspace(0.0, 2*np.pi, 50)
plt.plot(xvalues, np.sin(xvalues))
```

```
# Set major ticks for x and y axes, and xtick labels.
# We can use LaTeX code in labels!
plt.xticks([0.0, np.pi, 2*np.pi], ['0', r'$\pi$', r'$2\pi$'])
plt.yticks([-1.0, 0.0, 1.0])

# Adjust plot limits in x and y direction
plt.xlim((-0.5, 2*np.pi + 0.5))
plt.ylim((-1.1, 1.1))
```

[13]:
```
(-1.1, 1.1)
```



4.6 Adding straight lines

Quite often, we want to add horizontal or vertical lines to highlight a particular value. We can do this using the functions

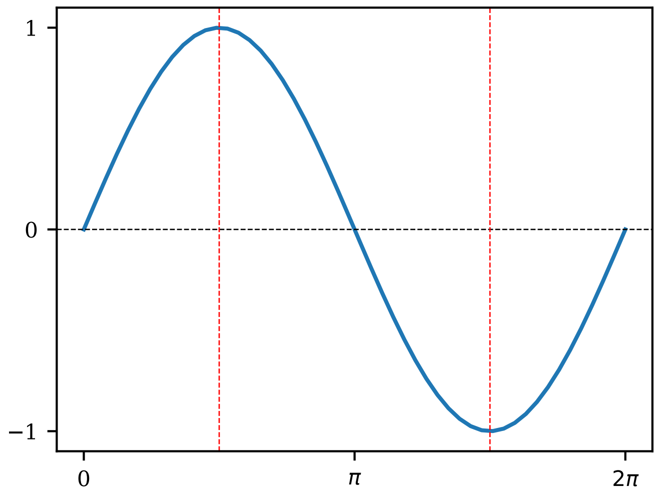
- `axhline()` to add a *horizontal* line at a given y-value.
- `axvline()` to add a *vertical* line at a given x-value.
- `axline()` to add a line defined by two points or by a single point and a slope.

Examples:

Consider the sine function from above. We can add a horizontal line at 0 and two vertical lines at the points where the function attains its minimum and maximum as follows:

[14]:
```
import matplotlib.pyplot as plt
import numpy as np

# Plot sine function (same as above)
xvalues = np.linspace(0.0, 2*np.pi, 50)
plt.plot(xvalues, np.sin(xvalues))
plt.xticks([0.0, np.pi, 2*np.pi], ['0', r'$\pi$', r'$2\pi$'])
plt.yticks([-1, 0, 1])

# Add black dashed horizontal line at y-value 0
plt.axhline(0.0, lw=0.5, ls='--', c='black')

# Add red dashed vertical lines at maximum / minimum points
plt.axvline(0.5*np.pi, lw=0.5, ls='--', c='red')
plt.axvline(1.5*np.pi, lw=0.5, ls='--', c='red')
```

```
[14]: <matplotlib.lines.Line2D at 0x7f5fa405afe0>
```



4.7 Object-oriented interface

So far, we have only used the so-called `pyplot` interface which involves calling *global* plotting functions from `matplotlib.pyplot`. This interface is intended to be similar to Matlab, but is also somewhat limited and less clean.

We can instead use the object-oriented interface (called this way because we call methods of the `Figure` and `Axes` objects). While there is not much point in using the object-oriented interface in a Jupyter notebook when we want to create a single graph, it should be the preferred method when writing re-usable code in Python files.

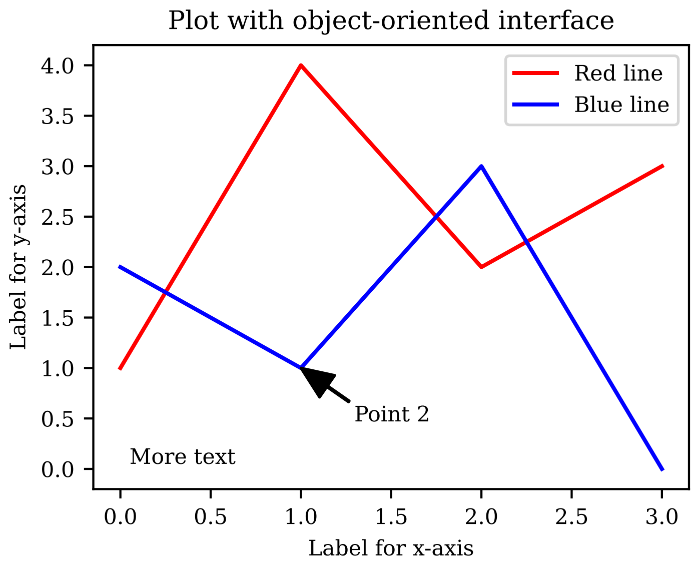
To use the object-oriented interface, we need to get figure and axes objects. The easiest way to accomplish this is using the `subplots()` function, like this:

```
fig, ax = plt.subplots()
```

As an example, we recreate the graph from the section on labels and annotations using the object-oriented interface:

```
[15]: import matplotlib.pyplot as plt

xvalues = [0, 1, 2, 3]
yvalues = [1, 4, 2, 3]
yvalues2 = [2.0, 1.0, 3.0, 0.0]

fig, ax = plt.subplots()
ax.plot(xvalues, yvalues, color='red', label='Red line')
ax.plot(xvalues, yvalues2, color='blue', label='Blue line')
ax.set_xlabel('Label for x-axis')
ax.set_ylabel('Label for y-axis')
ax.legend()
ax.set_title('Plot with object-oriented interface')
ax.text(0.05, 0.05, 'More text')

# Annotate second point
ax.annotate('Point 2',
    (xvalues[1], yvalues2[1]), (20, -20),
    textcoords='offset points',
    arrowprops={'facecolor': 'black', 'width': 0.5, 'headwidth': 10.0}
)
```

```
[15]: Text(20, -20, 'Point 2')
```



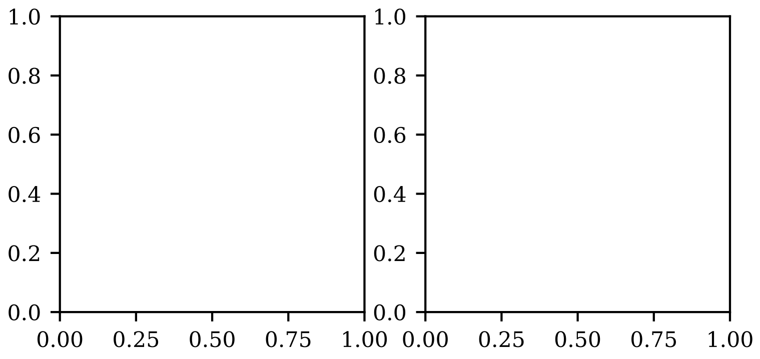
The code is quite similar, except that attributes are set using the `set_xxx()` methods of the `ax` object. For example, instead of calling `xlim()`, we use `ax.set_xlim()`.

4.8 Working with multiple plots (axes)

The object-oriented interface becomes particularly useful if we want to create multiple axes (or figures). This can also be achieved using with the `pyplot` programming model but is somewhat more obscure.

For example, to create a row with two subplots, we use:

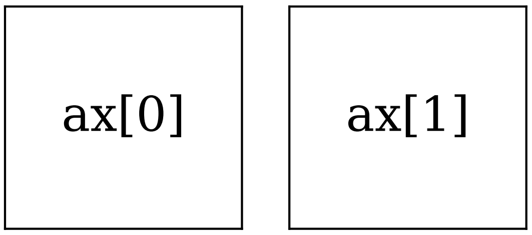
```
[16]: import matplotlib.pyplot as plt

# Create one figure with 2 axes objects, arranged as two columns in a single row
fig, ax = plt.subplots(1, 2, figsize=(4.5, 2.0))
```



With multiple axes objects in a single figure (as in the above example), the `ax` returned by `subplots()` is a NumPy array. Its elements map to the individual panels within the figure in a natural way.

We can visualise this mapping for the case of a single row and two columns as follows:

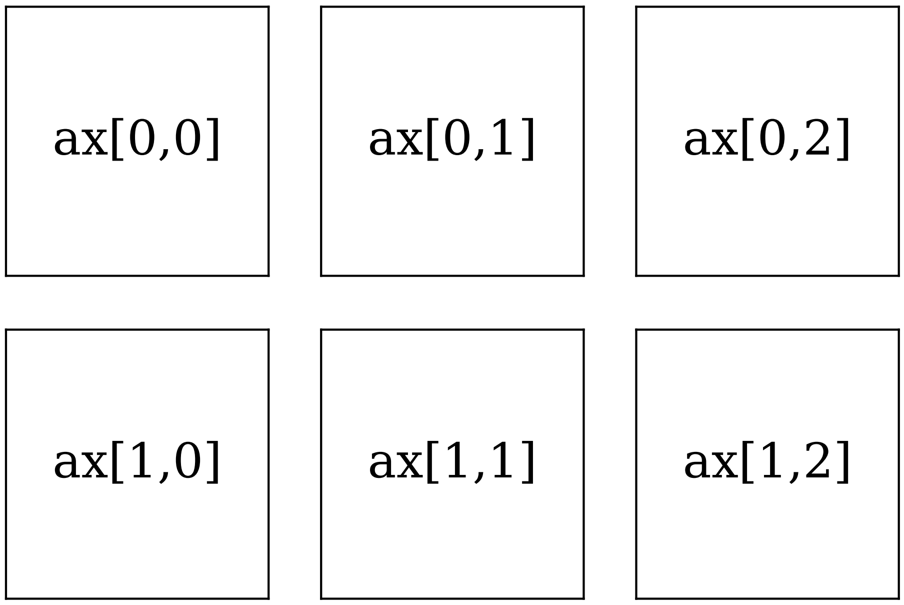
```
[17]: fig, ax = plt.subplots(1, 2, figsize=(3.5,1.5))
for i, axes in enumerate(ax):
    # Turn off ticks of both axes
    axes.set_xticks(())
    axes.set_yticks(())
    # Label axes object
    text = f'ax[{i}]'
    axes.text(0.5, 0.5, text, transform=axes.transAxes, va='center',
        ha='center', fontsize=18)
```



Don't worry about the details of how this graph is generated, the only take-away here is how axes objects are mapped to the panels in the figure.

If we request panels in two dimensions, the `ax` object will be a 2-dimensional array, and the mapping of axes objects to panels will look like this instead:

[18]:
```
# Request figure with 2 rows, 3 columns
fig, ax = plt.subplots(2, 3, figsize=(6,4))
for i, axrow in enumerate(ax):
    for j, axes in enumerate(axrow):
        # Turn off ticks of both axes
        axes.set_xticks(())
        axes.set_yticks(())
        # Label axes object
        text = f'ax[{i},{j}]'
        axes.text(0.5, 0.5, text, transform=axes.transAxes, va='center',
            ha='center', fontsize=18)
```

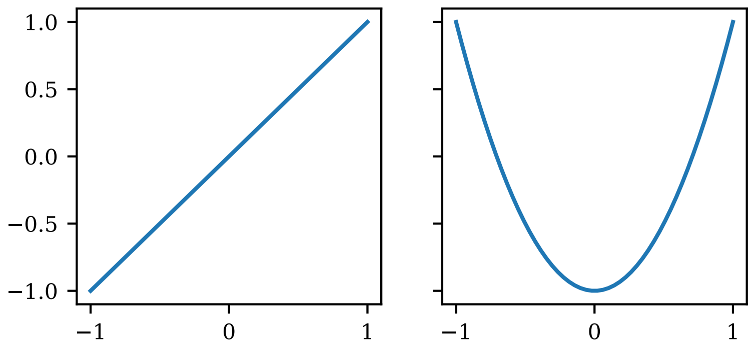


Returning to our initial example, we can use the elements of `ax` to plot into individual panels:

[19]:
```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(4.5, 2.0))
```

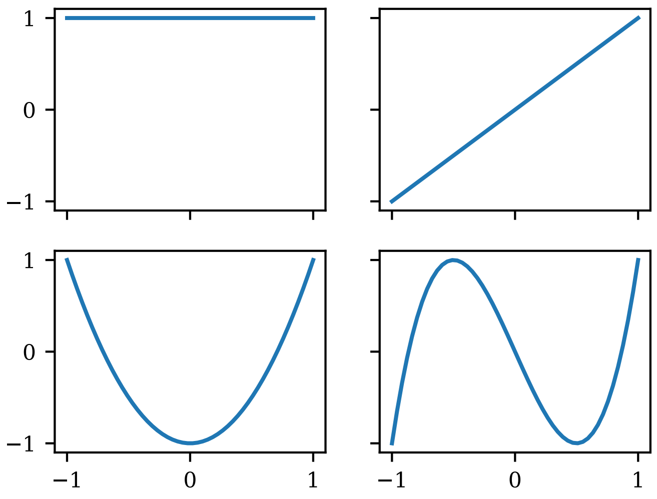
```
xvalues = np.linspace(-1.0, 1.0, 50)

# Plot into first column
ax[0].plot(xvalues, xvalues)

# Plot into second column
ax[1].plot(xvalues, 2*xvalues**2.0 - 1)
```

```
[19]: [<matplotlib.lines.Line2D at 0x7f5fa3c188b0>]
```



The next example illustrates how to create a figure with four panels:

```
[20]: # create figure with 2 rows, 2 columns
fig, ax = plt.subplots(2, 2, sharex=True, sharey=True)

xvalues = np.linspace(-1.0, 1.0, 50)

# Plot the first four Chebyshev polynomials on the interval [-1,1]
for i in range(2):
    for j in range(2):
        yvalues = np.cos((j + i*2) * np.arccos(xvalues))
        ax[i,j].plot(xvalues, yvalues)
```



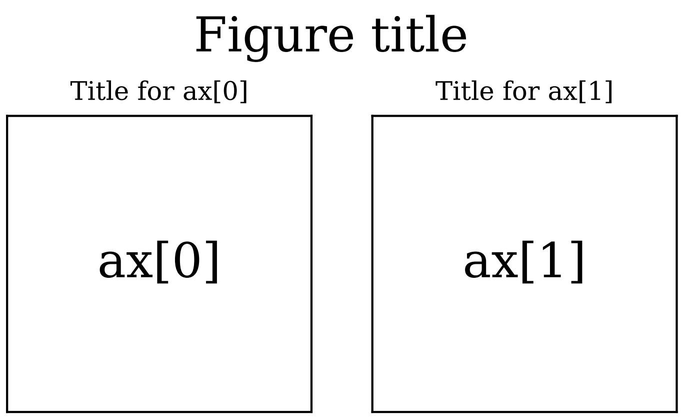
Note the use of `sharex=True` and `sharey=True`. This tells matplotlib that all axes share the same plot limits, so the tick labels can be omitted in the figure's interior to preserve space.

With multiple axes per figure, we can also see the difference between the labels generated by `set_title()` and `suptitle()`.

To illustrate, we re-use the previous example with two panels in a single row:

```
[21]: fig, ax = plt.subplots(1, 2, figsize=(4.5,2.0))
      for i, axes in enumerate(ax):
          axes.set_xticks(())
          axes.set_yticks(())
          text = f'ax[{i}]'
          axes.text(0.5, 0.5, text, transform=axes.transAxes, va='center',
              ha='center', fontsize=18)

          # Add axes-specific title
          axes.set_title(f'Title for ax[{i}]')

      # set overall figure title:
      # this is an attribute of the Figure object!
      fig.suptitle('Figure title', fontsize=18, va='bottom', y=1.02)
```

```
[21]: Text(0.5, 1.02, 'Figure title')
```



4.9 Optional exercises

Exercise 1: Trigonometric functions

Plot the functions $\sin(x)$ and $\cos(x)$ on the interval $[-\pi, \pi]$, each in a separate graph. Include a legend for each plot, and add pretty tick labels at $[-\pi, 0, \pi]$ which use the LaTeX symbol for π. Add an overall title "Trigonometric functions".

Hint: NumPy defines the functions `np.sin()` and `np.cos()` as well as the value `np.pi`.

Exercise 2: Logarithmic scaling

In economics and finance, we often plot using the $\log_{10}$ scale if the plotted data are of very different orders of magnitude.

Create a figure with two sub-plots, each plotting the function $f(x) = 10^x$ on a uniformly-spaced interval $[-5, 5]$ with 100 points. Use the (default) linear scale in the first plot, but apply the $\log_{10}$ scale in the second.

Hint: You can set the axis scale to log by calling `yscale('log')`, or `set_yscale('log')` when using the object-oriented interface.

Exercise 3: Multiple lines in single plot

In this exercise, we explore yet another alternative to plot multiple lines in a single graph.

The `plot()` function accepts y-values specified as a matrix in which each column corresponds to a different line. The number of rows must correspond to the length of the vector of x-values, which are assumed to be identical for all columns of y-values.

1. Consider the following family of polynomials in x parametrised by a:

   $$p(x;a) = a(x-0.5)^2$$

   Assume there are 5 such polynomials with a's given by the values

   ```
   a = np.linspace(0.4, 4.0, 5)
   ```

   - Create a common set of x-values using an equidistant grid of 50 points on the interval $[0,1]$.
   - Construct the matrix of y-values with shape `(50,5)` and plot all polynomials with a single call to `plot()`.
   - Add a legend that maps a value of a to the corresponding line in the plot.

2. Plot each polynomial separately using a loop. Use the `i`-th elements of the following arrays as plot styles for the `i`-th polynomial:

   ```
   colors = ['red', 'blue', 'black', 'green', 'purple']
   linewidths = [1.0, 1.5, 2.0, 1.5, 1.0]
   linestyles = ['-', '--', '-.', ':', '-']
   ```

   Add a legend that maps a value of a to the corresponding line in the plot.

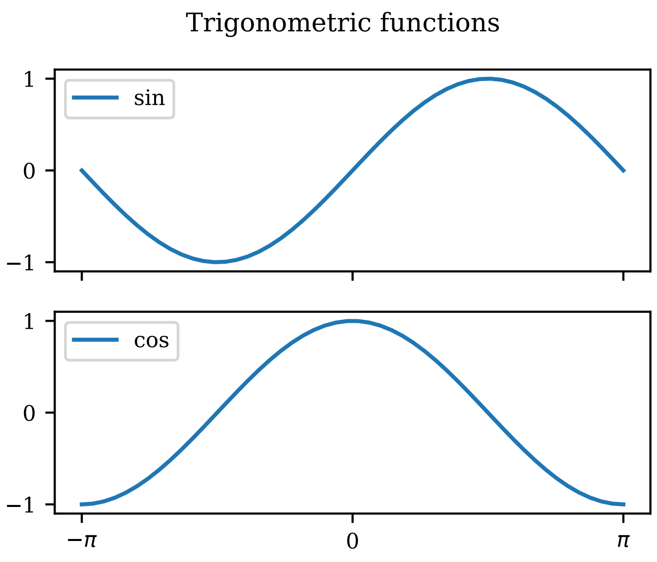
4.10 Solutions

Solution for exercise 1

[22]:
```python
import matplotlib.pyplot as plt
import numpy as np

xvalues = np.linspace(-np.pi, np.pi, 50)
# Create figure with two rows, one column
fig, ax = plt.subplots(2, 1, sharey=True, sharex=True)

xticks = [-np.pi, 0.0, np.pi]
# Tick labels use LaTeX notation for pi, which is \pi and has to be
# surrounded by $$.
xticklabels = [r'$-\pi$', '0', r'$\pi$']
yticks = [-1.0, 0.0, 1.0]

# Create sin() plot using first axes object
ax[0].plot(xvalues, np.sin(xvalues), label='sin')
ax[0].set_xticks(xticks)
ax[0].set_xticklabels(xticklabels)
ax[0].set_yticks(yticks)
ax[0].legend(loc='upper left')

# Create cos() plot using second axes object
ax[1].plot(xvalues, np.cos(xvalues), label='cos')
ax[1].set_xticks(xticks)
ax[1].set_xticklabels(xticklabels)
ax[1].set_yticks(yticks)
ax[1].legend(loc='upper left')
```

```
# Add overall figure title (this is not axes-specific)
fig.suptitle('Trigonometric functions')
```

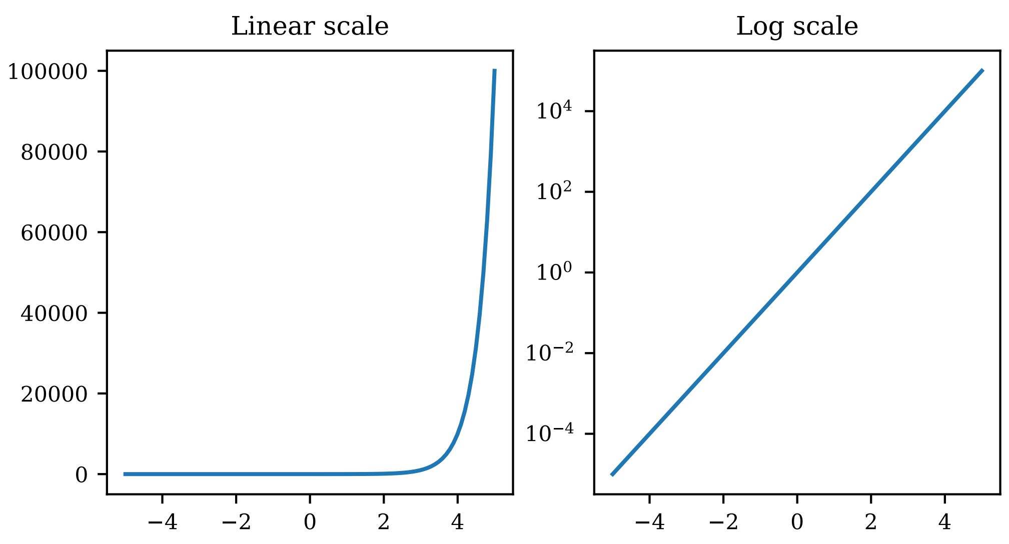
```
[22]: Text(0.5, 0.98, 'Trigonometric functions')
```



Solution for exercise 2

```
[23]: import matplotlib.pyplot as plt
      import numpy as np

      xvalues = np.linspace(-5.0, 5.0, 100)
      fig, ax = plt.subplots(1, 2, sharex=True, figsize=(6,3))
      ax[0].plot(xvalues, 10.0**xvalues)
      ax[0].set_title('Linear scale')

      ax[1].plot(xvalues, 10.0**xvalues)
      # Set y-axis to log scale (assumes base-10 log)
      ax[1].set_yscale('log')
      ax[1].set_title('Log scale')
```

```
[23]: Text(0.5, 1.0, 'Log scale')
```

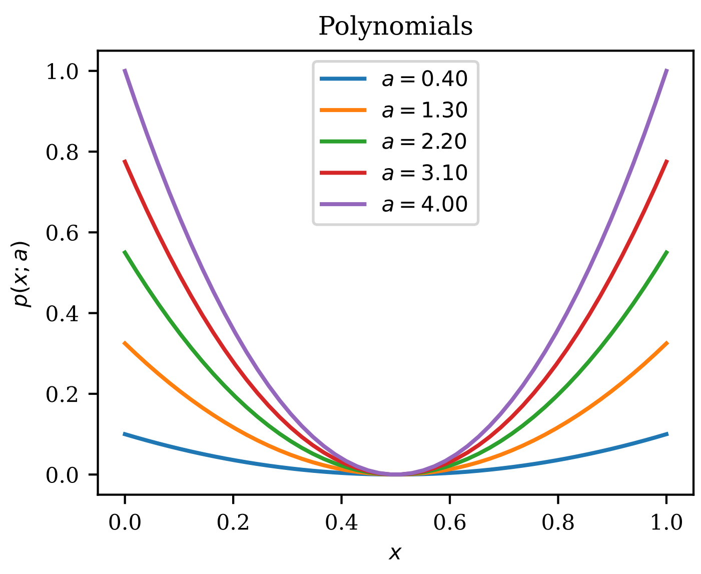


Solution for exercise 3

First we plot all polynomials in a single call to `plot()`. For this to work, we need to pass the *y*-values as a matrix where each column corresponds to a different line.

[24]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Create common x-values
xvalues = np.linspace(0.0, 1.0, 50)

# Create parameters
a = np.linspace(0.4, 4.0, 5)

# Evaluate polynomials on common x-values:
# each column corresponds to a different parametrisation
poly = np.zeros((len(xvalues), len(a)))
# Iterate over parameters and create each set of corresponding y-values
for i, ai in enumerate(a):
    poly[:,i] = ai * (xvalues - 0.5)**2.0

plt.plot(xvalues, poly)
plt.xlabel('$x$')
plt.ylabel(r'$p(x;a)$')
plt.title('Polynomials')

# Create legend:
# we need to pass labels in same order as the corresponding columns
labels = [r'$a={:.2f}$'.format(ai) for ai in a]
plt.legend(labels)
```

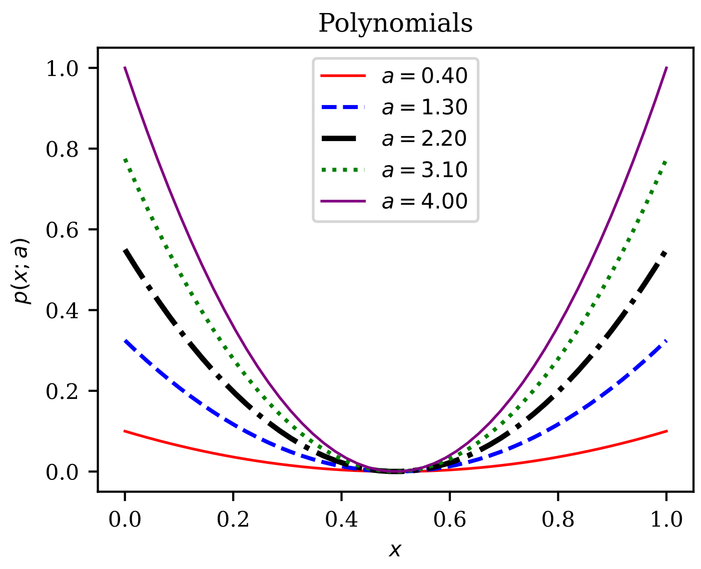
[24]:
```
<matplotlib.legend.Legend at 0x7f5fa3a459f0>
```



We now call `plot()` for each polynomial separately. This allows us to specify detailed style settings for each line.

[25]:
```
fig, ax = plt.subplots(1,1)

# Different styles for each parametrisation
colors = ['red', 'blue', 'black', 'green', 'purple']
linewidths = [1.0, 1.5, 2.0, 1.5, 1.0]
linestyles = ['-', '--', '-.', ':', '-']

n = poly.shape[1]
for i in range(n):
    label = r'$a={:.2f}$'.format(a[i])
    ax.plot(xvalues, poly[:,i], c=colors[i], lw=linewidths[i],
            ls=linestyles[i], label=label)

# calling legend without arguments will use the
# text provided as label argument to each plot()
ax.legend()
ax.set_xlabel('$x$')
ax.set_ylabel(r'$p(x;a)$')
ax.set_title('Polynomials')
```

[25]:
```
Text(0.5, 1.0, 'Polynomials')
```


Polynomials
$a = 0.40$
$a = 1.30$
$a = 2.20$
$a = 3.10$
$a = 4.00$
$p(x; a)$
x
0.0
0.2
0.4
0.6
0.8
1.0

5 Advanced NumPy

We already encountered NumPy arrays and their basic usage throughout this course. In this unit, we will take a more in-depth look at NumPy.

5.1 Why NumPy arrays?

Why don't we just stick with built-in types such as Python lists to store and process data? It turns out that while the built-in objects are quite flexible, this flexibility comes at the cost of decreased performance:

- `list` objects can store arbitrary data types, and the data type of any item can change:

  ```
  items = ['foo']
  items[0] = 1.0        # item was a string, now it's a float!
  ```

- There is no guarantee where in memory the data will be stored. In fact, two consecutive items could be very "far" from each other in memory, which imposes a performance penalty.
- Even primitive data types such as `int` and `float` are not "raw" data, but full-fledged objects. That, again, is bad for performance.

On the other hand, the approach taken by NumPy is to store and process data in a way very similar to low-level languages such as C and Fortran. This means that

- Arrays contain a *homogenous* data type. *All* elements are either 64-bit integers (`np.int64`), 64-bit floating-point numbers (`np.float64`), or some other of the many data types supported by NumPy.

  It is technically possible to get around this by specifying an array's data type (`dtype`) to be `object`, which is the most generic Python data type. However, we would never want to do this for numerical computations.

- NumPy arrays are usually *contiguous* in memory. This means that adjacent array elements are actually guaranteed to be stored next to each other, which allows for much more efficient computations.
- NumPy arrays support numerous operations used in scientific computing. For example, with a NumPy array we can write

  ```
  x = np.array([1, 2, 3])
  y = x + 1         # We would expect this to work
  ```

  With lists, however, we cannot:

  ```
  x = [1, 2, 3]
  y = x + 1         # Does not work!
  ```

  Lists don't implement an addition operator that accepts integer arguments, so this code triggers an error.

You can see the performance uplift provided by NumPy arrays in this simple example:

[1]:
```
# Create list 0, 1, 2, ..., 999
lst = list(range(1000))

# Compute squares, time how long it takes
%timeit [i**2 for i in lst]
```

```
162 µs ± 313 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[2]:
```
# Repeat using NumPy arrays
import numpy as np
arr = np.arange(1000)

%timeit arr**2
```

```
973 ns ± 8.85 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

- On my machine, squaring 1000 elements of a `list` takes approximately 200 times longer than the computation using NumPy arrays!
- Also, as mentioned above, NumPy supports squaring an array directly, while we have to manually loop through the `list` and square each element individually.

Note: `%timeit` is a so-called magic command that only works in notebooks, but not in regular Python files. [See documentation]

5.2 Creating arrays

We have already encountered some of the most frequently used array creation routines:

- `np.array()` creates an array from a given argument, which can be
  - a scalar;
  - a collection such as a list or tuple;
  - some other iterable object, e.g., something created by `range()`.
- `np.empty()` allocates memory for a given array shape, but does not overwrite it with initial values.
- `np.zeros()` creates an array of a given shape and initializes it to zeros.
- `np.ones()` creates an array of a given shape and initializes it to ones.
- `np.arange(start,stop,step)` creates an array with evenly spaced elements over the range $[start, stop)$.
  - `start` and `step` can be omitted and then default to `start=0` and `step=1`.
  - Note that the number `stop` is never included in the resulting array!
- `np.linspace(start,stop,num)` returns a vector of `num` elements which are evenly spaced over the interval $[start, stop]$.
- `np.identity(n)` returns the identity matrix of a size $n \times n$.
- `np.eye()` is a more flexible variant of `identity()` that can, for example, also create non-squared matrices.

There are many more array creation functions for more exotic use-cases, see the NumPy documentation for details.

Examples:

[3]:
```
import numpy as np

# Create array from list
lst = [1, 2, 3]
np.array(lst)
```

[3]:
```
array([1, 2, 3])
```

[4]:
```
# Create array from tuple
tpl = 1.0, 2.0, 3.0
np.array(tpl)
```

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

```
[5]: # arange: end point is not included!
     np.arange(5)
```

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

```
[6]: # arange: increments can be negative too!
     np.arange(5, 1, -1)
```

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

```
[7]: # arange also works on floats
     np.arange(1.0, 3.0, 0.5678)
```

```
[7]: array([1.    , 1.5678, 2.1356, 2.7034])
```

```
[8]: # linspace DOES include the end point
     np.linspace(0.0, 1.0, 11)
```

```
[8]: array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

5.3 Array shape

Many of the array creation routines take the desired shape of the array as their first argument. Array shapes are usually specified as tuples:

- A vector with 5 elements has shape `(5, )`.

  Note the comma `,`: we need to specify a tuple with a single element using this comma, since `(5)` is just the integer 5, not a tuple.

  It is worth pointing out that this is not the same as a 2-dimensional array with shape `(1, 5)` or `(5, 1)`, even though they have the same number of elements.
- A 2×2 matrix has shape `(2, 2)`.
- A higher-dimensional array has shape `(k, l, m, n, ...)`.
- A *scalar* NumPy array has shape `()`, an empty tuple.

  While "scalar array" sounds like an oxymoron, it does exist.

We can query the shape of an array using the `shape` attribute, and the number of dimensions is stored in the `ndim` attribute.

Examples:

```
[9]: import numpy as np

     # Scalar array
     x = np.array(0.0)
     print(f'Scalar array with shape={x.shape} and ndim={x.ndim}')
```

```
Scalar array with shape=() and ndim=0
```

Note that a scalar NumPy array is not the same as a Python scalar. The built-in type `float` has neither a `shape`, nor an `ndim`, nor any other of the NumPy array attributes.

[10]:
```
scalar = 1.0
scalar.shape
```

```
AttributeError: 'float' object has no attribute 'shape'
```

We create an empty array as follows:

[11]:
```
# 1-dimensional array (vector), values not initialised
x = np.empty((5,))
x       # could contain arbitrary garbage
```

[11]:
```
array([4.68013972e-310, 0.00000000e+000, 4.67905024e-310, 4.67905021e-310,
       2.37151510e-322])
```

An array created with `empty()` will contain arbitrary garbage since the memory block assigned to the array is not initialised. The result will most likely differ on each invocation and across computers.

Most functions accept an integer value instead of a `tuple` when creating 1-dimensional arrays, which is interpreted as the number of elements:

[12]:
```
# 1-dimensional array
x = np.empty(5)          # equivalent to np.empty((5,))
```

Higher-dimensional arrays are creating by passing in tuples with more than one element:

[13]:
```
np.ones((1, 2, 3))      # 3d-array
```

[13]:
```
array([[[1., 1., 1.],
        [1., 1., 1.]]])
```

Recall from unit 2 that we can use the `reshape()` method to convert arrays to a different shape:

- The resulting number of elements must remain unchanged!
- *One* dimension can be specified using `-1`, which will prompt NumPy to compute the implied dimension size itself.

[14]:
```
x = np.zeros((2, 1, 3))
x = x.reshape((3, -1))      # Infer number of columns
x
```

[14]:
```
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

We can reshape any array to a 1-dimensional vector using any of the following expressions:

[15]:
```
x.reshape((-1, ))           # pass shape as tuple
x.reshape(-1)               # pass shape as integer
x.flatten()
```

[15]:
```
array([0., 0., 0., 0., 0., 0.])
```

This even works on scalar (0-dimensional) arrays:

[16]:
```
np.array(0.0).flatten()
```

[16]:
```
array([0.])
```

5.4 Advanced indexing

We previously discussed single element indexing and slicing, which works the same way for both Python `list` and `tuple` objects as well as NumPy arrays. NumPy additionally implements more sophisticated indexing mechanisms which we cover now. You might also want to consult the NumPy indexing tutorial.

5.4.1 Boolean or "mask" indexing

We can pass logical arrays as indices:

- Logical (or boolean) arrays consist of elements that can only take on values `True` and `False`
- We usually don't create logical arrays manually, but apply an operation that results in `True`/`False` values, such as a comparison.
- The boolean index array usually has the *same* shape as the indexed array.

Examples:

[17]:
```
import numpy as np

vec = np.arange(5)
mask = (vec > 1)        # apply comparison to create boolean array
mask
```

[17]:
```
array([False, False,  True,  True,  True])
```

[18]:
```
vec[mask]               # use mask to retrieve only elements greater than 1
```

[18]:
```
array([2, 3, 4])
```

We can even apply boolean indexing to multi-dimensional arrays. The result will be flatted to a 1-dimensional array, though.

[19]:
```
mat = np.arange(6).reshape((2,3))
mat
```

[19]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

[20]:
```
mask = (mat > 1)        # create boolean array
mask
```

[20]:
```
array([[False, False,  True],
       [ True,  True,  True]])
```

[21]:
```
mat[mask]             # collapses result to 1-d array
```

[21]:
```
array([2, 3, 4, 5])
```

Note that logical indexing does *not* work with `tuple` and `list`

[22]:
```
tpl = (1, 2, 3)
mask = (True, False, True)
tpl[mask]               # error
```

```
TypeError: tuple indices must be integers or slices, not tuple
```

5.4.2 Integer index arrays

We can also use index arrays of *integer* type to select specific elements on each axis. These are straightforward to use for 1-dimensional arrays, but can get fairly complex with multiple dimensions.

[23]:
```
import numpy as np

data = np.arange(10)
index = [1, 2, 9]       # select second, third and 10th element
data[index]
```

[23]:
```
array([1, 2, 9])
```

As you see, the index array does not have to be a NumPy array, but can also be a list (not a tuple, though!).

In general, if we are using an index array to select elements along an axis of length n, then

- the index must only contain integers between 0 and $n-1$, or negative integers from $-n$ to -1 (which, as usual, count from the end of the axis).
- the index can be of arbitrary length. We can therefore select the same element multiple times.

[24]:
```
data = np.arange(5, 10)     # array with 5 elements, [5,...,9]
data
```

[24]:
```
array([5, 6, 7, 8, 9])
```

[25]:
```
index = [0, 1, 1, 2, 2, 3, 3, 4, 4]         # select elements multiple times
data[index]
```

[25]:
```
array([5, 6, 6, 7, 7, 8, 8, 9, 9])
```

The same restrictions apply when indexing multi-dimensional arrays. Moreover,

- if more than one axis is indexed using index arrays, the index arrays have to be of equal length.
- we can combine integer array indexing on one axis with other types of indices on the remaining axes.

Examples:

[26]:
```
data = np.arange(12).reshape((3, 4))
data
```

[26]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

[27]:
```
index1 = [0, 2]     # row indices
index2 = [1, 3]     # column indices
data[index1, index2]
```

[27]:
```
array([ 1, 11])
```

The code above selects two elements, the first at position `(0,1)`, the second at position `(2,3)`.

We can combine index arrays on one axis with another indexing method on a different axis:

[28]:
```
data[index1, 2]     # return elements in 3rd column from rows given
                    # in index1
```

[28]:
```
array([ 2, 10])
```

Using different indexing methods, in particular index arrays, on higher-dimensional data can quickly become a mess, and you should be extra careful to see if the results make sense.

5.5 Numerical operations

5.5.1 Element-wise operations

Element-wise operations are performed on each element individually and leave the resulting array's shape unchanged.

There are three types of such operations:

1. One operand is an array and one is a scalar.
2. Both operands are arrays, either of identical shape, or broadcastable to an identical shape (we discuss broadcasting below)
3. A function is applied to each array element.

Case 1: Array-scalar operations. These intuitively behave as you would expect:

[29]:
```
import numpy as np

x = np.arange(10)
scalar = 1

# The resulting array y has the same shape as x:
y = x + scalar      # addition
y = x - scalar      # subtraction
y = x * scalar      # multiplication
y = x / scalar      # division
y = x // scalar     # division with integer truncation
y = x % scalar      # modulo operator
y = x ** scalar     # power function
y = x == scalar     # comparison: also >, >=, <=, <
```

Note that unlike in Matlab, the "standard" operators work element-wise, so `x * y` is *not* matrix multiplication!

Case 2: Both operands are arrays of equal shape:

[30]:
```
x = np.arange(10)
y = np.arange(10, 20)       # has same shape as x

# Resulting array z has the same shape as x and y:
z = x + y           # addition
z = x - y           # subtraction
z = x * y           # multiplication
z = x / y           # division
z = x // y          # division with integer truncation
z = x % y           # modulo operator
z = x ** y          # power function
z = x == y          # comparison: also >, >=, <=, <
```

Case 3: Applying element-wise functions. This case covers numerous functions defined in NumPy, such as

- `np.sqrt()`: square root
- `np.exp()`, `np.log()`, `np.log10()`: exponential and logarithmic functions
- `np.sin()`, `np.cos()`, etc.: trigonometric functions

You can find a complete list of mathematical functions in the NumPy documentation (not all functions listed there operate element-wise, though).

```
[31]: # element-wise functions
      x = np.arange(1, 11)
      y = np.exp(x)       # apply exponential function
      y = np.log(x)       # apply natural logarithm
```

5.5.2 Matrix operations

Transpose

You can transpose a matrix using the `T` attribute:

```
[32]: mat = np.arange(6).reshape((2,3))
      mat
```

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

```
[33]: mat.T
```

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

For higher-dimensional arrays, the `np.transpose()` function can be used to permute the axes of an array. For two-dimensional arrays, `np.transpose(mat)` and `mat.T` are equivalent.

Matrix multiplication

Matrix multiplication is performed using the `np.dot()` function ("dot product"). The operands need not be matrices but can be vectors as well, or even high-dimensional arrays (the result is then not entirely obvious and one should check the documentation).

Every newer version of Python and NumPy additionally interprets `@` as the matrix multiplication operator.

```
[34]: import numpy as np

      mat = np.arange(9).reshape((3, 3))      # 3x3 matrix
      vec = np.arange(3)                      # vector of length 3

      # matrix-matrix multiplication
      np.dot(mat, mat)    # or: mat @ mat
```

```
[34]: array([[ 15,  18,  21],
             [ 42,  54,  66],
             [ 69,  90, 111]])
```

```
[35]: # vector dot product (returns a scalar)
      np.dot(vec, vec)    # or: vec @ vec
```

```
[35]: 5
```

```
[36]: # matrix-vector product (returns vector)
      np.dot(mat, vec)    # or: mat @ vec
```

```
[36]: array([ 5, 14, 23])
```

We must of course make sure that matrices and vector have conformable dimensions!

```
[37]: mat = np.arange(6).reshape((2, 3))
      mat
```

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

```
[38]: np.dot(mat, mat)         # raises error, cannot multiply 2x3 matrix with
                               # 2x3 matrix
```

```
ValueError: shapes (2,3) and (2,3) not aligned: 3 (dim 1) != 2 (dim 0)
```

```
[39]: np.dot(mat, mat.T)       # transpose second operand, then it works!
```

```
[39]: array([[ 5, 14],
             [14, 50]])
```

5.5.3 Reductions

Reductions are operations that reduce the dimensionality of the data. For example, computing the mean of an array reduces a collection of data points to a single scalar, its mean.

Basic reduction operations include:

- `np.sum()`: sum of array elements
- `np.prod()`: product of array elements
- `np.amin()`, `np.amax()`: minimum and maximum element
- `np.argmin()`, `np.argmax()`: location of minimum and maximum element
- `np.mean()`, `np.average()`: mean of array elements
- `np.median()`: median of array elements
- `np.std()`, `np.var()`: standard deviation and variance of array elements
- `np.percentile()`: percentiles of array elements

Most if not all reductions accept an `axis` argument which restricts the operation to a specific axis.

- If an axis is specified, the resulting array will have one dimension less than the input.
- If no axis is specified, the operation is applied to the whole (flattened) array.

Examples:

```
[40]: import numpy as np

      # 1-dimensional input data
      data = np.linspace(0.0, 1.0, 11)

      # Compute mean and std. of input data
      m = np.mean(data)
      s = np.std(data)
      print(f'Mean: {m:.2f}, std. dev.: {s:.2f}')
```

```
Mean: 0.50, std. dev.: 0.32
```

```
[41]: # 2-dimensional input data
      data = np.linspace(0.0, 1.0, 21).reshape((3, 7))
      data
```

```
[41]: array([[0.  , 0.05, 0.1 , 0.15, 0.2 , 0.25, 0.3 ],
             [0.35, 0.4 , 0.45, 0.5 , 0.55, 0.6 , 0.65],
       [0.7 , 0.75, 0.8 , 0.85, 0.9 , 0.95, 1.  ]])
```

[42]:
```
# Compute mean of each row, ie along the column axis
m = np.mean(data, axis=1)
m           # Result is a vector of 3 elements, one for each row
```

[42]:
```
array([0.15, 0.5 , 0.85])
```

5.6 Broadcasting

Element-wise operations in most programming languages require input arrays to have identical shapes. NumPy relaxes this constraint and allows us to use arrays with different shapes that can be "broadcast" to identical shapes.

Simple example

What do we mean by "broadcasting"? We introduce the concept using a specific example, and will discuss the technical details below.

- Imagine we want to add a 2×3 matrix to a length-2 vector.
- This operation does not make sense, unless we interpret the (column) vector as a 2×1 matrix, and replicate it 3 times to obtain a 2×3 matrix. This is exactly what NumPy does.

Examples:

[43]:
```
import numpy as np

# Create 3x2 matrix
mat = np.arange(6).reshape((2, 3))
mat
```

[43]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

[44]:
```
# Create 2-element vector
vec = np.arange(2)
vec
```

[44]:
```
array([0, 1])
```

[45]:
```
# Trying to add matrix to vector fails
mat + vec
```

```
ValueError: operands could not be broadcast together with shapes (2,3) (2,)
```

[46]:
```
# However, we can explicitly reshape the vector to a 2x1 column vector
colvec = vec.reshape((-1, 1))
colvec
```

[46]:
```
array([[0],
       [1]])
```

[47]:
```
# Now, broadcasting replicates column vector to match matrix columns
mat + colvec
```

```
[47]: array([[0, 1, 2],
             [4, 5, 6]])
```

We do not need to `reshape()` data, but can instead use a feature of NumPy that allows us to increase the number of dimensions on the spot:

```
[48]: # use vec[:, None] to append an additional dimension to vec
      mat + vec[:, None]
```

```
[48]: array([[0, 1, 2],
             [4, 5, 6]])
```

Specifying `None` as an array index inserts a new axis of length 1 at that position (since it's of length 1, this new axis does not change the overall size of the array!).

For more examples, see the official NumPy tutorial on broadcasting.

Technical details (advanced)

We are now ready to look at the technical details underlying broadcasting. The NumPy documentation on broadcasting is quite comprehensive, so we will just summarise the points made there.

Broadcasting is applied in four steps:

1. Determine the largest dimension (`ndim` attribute) among all arrays involved in an operation. Any array of smaller dimension will have 1's *prepended* to its shape until its dimension corresponds to the largest one.

   Example: given array `a` with shape `(m,n)` and array `b` with shape `(n,)`, the maximum dimension is 2, and `b` will be implicitly reshaped to `(1,n)`.
2. The size of the output array is determined as the maximum size of all arrays along each dimension.

   Example: Continuing with our example from above, the maximum size along dimension 1 is `m`, and the maximum size along dimension 2 is `n`, so the output array has shape `(m,n)`.
3. An input array can be used in the computation if for every dimension its size either matches the output size or is equal to 1. If this is not the case, broadcasting cannot be applied and the operation fails.

   Example: In the above example, the shape of `a` matches the output shape exactly. The implied shape of `b` is `(1,n)`, so it matches exactly along the second dimension, and is 1 along the first, and thus can be used.
4. For any input array with size 1 along some dimension, the (unique) element in this dimension will be used for all calculations along that dimension.

   Example: Any element `a[i,j]` will be matched with the element `b[0,j]` to the calculate the output value at `(i,j)`.

Because additional dimensions are added *at the beginning* to create the desired output shape, broadcasting will not work automatically if we want to multiply arrays of shape `a.shape = (m,n)` and `b.shape = (m,)`.

- Following the above steps, `b` will implicitly be reshaped to `(1,m)` and the operation will fail at step 3:

```
[49]: m = 3
      n = 2
      a = np.arange(m*n).reshape((m, n))      # matrix of shape (m,n)
      b = np.arange(1, m+1)                   # vector of shape (m,)
```

```
a * b           # will not work!
```

```
ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

We therefore have to explicitly *append* a degenerate axis to `b` such that both arrays have the same dimension:

[50]:
```
b = b[:,None]
a * b
```

[50]:
```
array([[ 0,  1],
       [ 4,  6],
       [12, 15]])
```

Because `a` has shape `(m,n)` and `b` now has shape `(m,1)`, `b[:,0]` will be replicated across all columns of `a` to perform the operation.

It is worthwhile to take some time to master broadcasting as it's essential to using NumPy efficiently. You might think that one can simply replicate array operands along some dimension to get the same effect, which is what we do in languages that do not support broadcasting.

- This included Matlab until release R2016b, where implicit expansion for some arithmetic and logical operations was introduced.

  Prior to that, users had to manually expand input arrays using `repmat()`, or use the rather inelegant `bsxfun()` function.

  Note that even today, NumPy broadcasting goes beyond Matlab's capabilities.

To illustrate the difference between broadcasting and manual replication of data, we perform the element-wise multiplication of a 3-dimensional array with a (1-dimensional) vector:

[51]:
```
# Dimensions of 3d array
k = 10
m = 11
n = 12

a = np.arange(k*m*n).reshape((k, m, n))     # create 3d array
b = np.arange(n)                            # create 1d vector
```

We can manually expand the vector to have the same shape as the array `a` using `np.tile()` which creates `k * m` copies of the `n` elements in `b`:

[52]:
```
b_exp = np.tile(b, reps=(k, m, 1))
b_exp.shape
```

[52]:
```
(10, 11, 12)
```

The following code compares the execution time of computing `a * b` using broadcasting to the case where we first explicitly expand `b`:

[53]:
```
# Multiplication with broadcasting
%timeit a * b
```

```
2.01 µs ± 10.7 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

[54]:
```
# Multiplication with explicitly expanded operands
%timeit a * np.tile(b, reps=(k, m, 1))
```

```
5.68 µs ± 25.5 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

As you see, not only is the second operation more complex and difficult to read, it also takes approximately three times as long to execute! The reason is that `np.tile()` replicates the data in memory, which is expensive. No copying is done when using broadcasting.

5.7 Vectorisation

Vectorisation is the concept of applying operations to whole arrays of data instead of every singular element (note that the term also has other meanings in computer science). In Python, as well as languages such as Matlab and R, we use this programming technique to increase performance for two reasons:

1. Looping over elements is slow.
2. Calling a function on every single element is also slow.

These performance penalties are less pronounced for compiled languages such as C or Fortran, so we try to move the looping to code written in one of these languages. In particular, since NumPy's core parts are implemented in C, we always want to do looping "within" NumPy.

For example, consider element-wise addition of two arrays, `a` and `b`:

[55]:
```
import numpy as np

# array size
N = 100

# input arrays
a = np.linspace(0.0, 1.0, N)
b = np.linspace(1.0, 2.0, N)
```

Benchmarking a non-vectorised loop in pure Python against NumPy's vectorised implementation reveals some striking differences:

[56]:
```
%%timeit
# Compute c = a + b using Python loops
c = np.empty(N)         # allocate output array
for i in range(N):
    c[i] = a[i] + b[i]
```

```
20.2 µs ± 742 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[57]:
```
# Compute c = a + b using vectorised addition
%timeit c = a + b
```

```
396 ns ± 0.881 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

We see that the vectorised variant is about 100 times faster! What is going on?

- NumPy implements a vectorised operator `+` which accepts arrays as operands.
- NumPy performs looping over individual elements in C which is compiled to high-performance machine code.

Of course this example is somewhat artificial: you would not have implemented a loop in Python in the first place, you probably just assumed that `+` works with array operands. However, we can easily construct more complex computations which may not be implemented in NumPy but which can use vectorised building blocks to speed up computations.

As an example, consider the following function which computes the sum of finite of elements of an array, ignoring infinity and NaNs which we detect using the function `np.isfinite()`.

[58]:
```
# Compute sum of finite elements in x
def finite_sum(x):
    # initialise sum
    s = 0.0
    # loop over array elements
    for xi in x:
        # Check whether a value is finite
        if np.isfinite(xi):
            # Add to running sum
            s += xi
    return s
```

Next, we create a sample array with a few NaNs and infinite values and to test `finite_sum()`. `NaN` stands for "not a number" as is used as a special value to flag results of invalid operations such as `0/0`:

[59]:
```
arr = np.linspace(0.0, 1.0, 100)
arr[::3] = np.inf       # assign infinity to every 3rd element
arr[::5] = np.nan       # assign NaN to every 5th element
```

Comparing our `finite_sum()` to a vectorised version again shows the performance advantage of vectorised code:

[60]:
```
%timeit finite_sum(arr)
```

```
128 µs ± 382 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[61]:
```
%timeit np.sum(arr[np.isfinite(arr)])
```

```
4.34 µs ± 15.4 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

The vectorised version is approximately 25 times faster, despite the fact that

- NumPy does not implement this specific function; and
- the vectorised implementation creates two temporary arrays:
  - one when calling `np.isfinite()`; and
  - one when indexing `arr` with the boolean array returned by `np.isfinite()`. As we discuss below, indexing with boolean arrays always creates a copy!

This illustrates that creating vectorised code by combining several vectorised functions also yields considerable speed-ups.

As an aside, note that NumPy actually implements `np.nansum()` which drops NaNs, but it does not discard infinite values.

5.8 Copies and views (advanced)

Recall that assignment in Python does *not* create a copy (unlike in C, Fortran or Matlab):

[62]:
```
a = [0, 0, 0]
b = a               # b references the same object as a
b[1] = 1            # modify second element of b (and a!)
a == b              # a and b are still the same
```

[62]:
```
True
```

NumPy adds another layer to this type of data sharing: whenever you perform an assignment or indexing operation, NumPy tries hard *not* to copy the underlying data but instead creates a so-called view which points to the same block of memory. It does so for performance reasons (copying is expensive).

We can illustrate this using array slicing:

```
[63]: import numpy as np

      x = np.arange(10)
      x
```

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

```
[64]: y = x[3:8]          # Create array that points to elements 4-8 of x
      y
```

```
[64]: array([3, 4, 5, 6, 7])
```

The arrays x and y are two different Python objects, which we can verify using the built-in id() function:

```
[65]: print(id(x))
      print(id(y))
```

```
139955888146160
139955888146832
```

```
[66]: id(x) == id(y)
```

```
[66]: False
```

And yet, the NumPy implementation makes sure that they reference the same block of memory!

We can see this easily by modifying y:

```
[67]: y[:] = 0        # overwrite all elements of y with zeros
      y
```

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

```
[68]: x               # elements of x that are also referenced by y
                      # are now also zero!
```

```
[68]: array([0, 1, 2, 0, 0, 0, 0, 0, 8, 9])
```

This behaviour is even triggered when y references non-adjacent elements in x. For example, we can let y be a view onto every *second* element in x:

```
[69]: x = np.arange(10)
      y = x[::2]      # y now points to every second element of x
      y[:] = 0        # overwrite all elements of y with zeros
      x               # every second element in x is now zero!
```

```
[69]: array([0, 1, 0, 3, 0, 5, 0, 7, 0, 9])
```

As a rule of thumb, NumPy will create a view as opposed to copying data if

- An array is created from another array via slicing (i.e., indexing using the `start:stop:step` triplet)

Conversely, a *copy* is created whenever

- An array is created from another array via boolean (mask) indexing.
- An array is created from another array via integer array indexing.

Moreover, you can always force NumPy to create a copy by calling `np.copy()`!

Examples:

[70]:
```
# Copies are created with boolean indexing
x = np.arange(10)
mask = (x > 4)      # boolean mask
y = x[mask]         # create y using boolean indexing
y[:] = 0
x                   # x is unmodified
```

[70]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[71]:
```
# Copies are created with integer array indexing
x = np.arange(10)
index = [3, 4, 5]   # List of indices to include in y
y = x[index]
y[:] = 0
x                   # x is unmodified
```

[71]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[72]:
```
# Forced copy with slicing
x = np.arange(10)
y = np.copy(x[3:8]) # force copy with np.copy()
y[:] = 0
x                   # x is unmodified
```

[72]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

As an alternative to `np.copy()`, we can directly call the `copy()` method of an array:

[73]:
```
y = x[3:8].copy()
```

5.9 NumPy data types (advanced)

5.9.1 Default data types

We have already touched upon the numerical data types used by NumPy. If we do not explicitly request a data type using the `dtype` keyword argument, NumPy by default behaves as follows:

1. The default data type for most array creation routines which create arrays of a given shape or size, such as `np.empty()`, `np.ones()` and `np.zeros()`, is a 64-bit floating-point number (`np.float64`).
2. Array creation routines that accept numerical input data will use the data type of this input data to determine the array data type.

   Examples of such functions are `np.arange()` and `np.array()`.
3. Arrays that are implicitly created as a result of an operation (addition, etc.) are assigned the most suitable type to represent the result.

   For example, when adding a floating-point and an integer array, the result will be a floating-point array.

Examples:

Case 1: default data type is `np.float64`:

```
[74]: import numpy as np

      x = np.ones(1)       # length-1 vector of ones
      x.dtype              # default type: float64
```

```
[74]: dtype('float64')
```

Case 2: data type depends on input data:

```
[75]: # Argument is an integer
      x = np.arange(5)
      x.dtype              # data type is np.int64
```

```
[75]: dtype('int64')
```

```
[76]: # Argument is a float
      x = np.arange(5.0)
      x.dtype              # data type is np.float64
```

```
[76]: dtype('float64')
```

Case 3: data type determined to accommodate result

```
[77]: # Add two integer arrays
      arr1 = np.arange(3)
      arr2 = np.arange(3, 0, -1)      # creates [3, 2, 1]
      result = arr1 + arr2
      print(result)
      result.dtype                    # data type is np.int64
```

```
[3 3 3]
```

```
[77]: dtype('int64')
```

```
[78]: # Add integer to floating-point array
      arr1 = np.arange(3)
      arr2 = np.arange(3.0, 0.0, -1.0)    # creates [3.0, 2.0, 1.0]
      result = arr1 + arr2
      print(result)
      result.dtype                        # data type is np.float64
```

```
[3. 3. 3.]
```

```
[78]: dtype('float64')
```

Even though the resulting array is `[3.0, 3.0, 3.0]` and can thus be represented as integers without loss of data, NumPy only takes into account that one of the operands is floating-point, and thus the result has to be of floating-point type!

5.9.2 Explicit data types

We can almost always explicitly request an array to be of a particular data type by passing the `dtype` keyword argument. The most common types are:

- `np.float64`: a 64-bit floating-point number, also called *double precision* in other languages.

  This is the most commonly used floating-point data type. It can represent numbers with up to 16 decimal digits, and covers a range of approximately $\pm 10^{308}$.

  Note that `dtype=float` is a synonym for `dtype=np.float64` on most platforms you are likely to encounter, and we'll be using the shorter variant.

- `np.int64`: a 64-bit integer which can represent integer values on the interval of (approximately) $\pm 10^{19}$.

  Unlike floating-point, the integer representation is *exact*, but covers a much smaller range (and, obviously, no fractional numbers)

  Note that `dtype=int` is a synonym for `dtype=np.int64` on most platforms you are likely to encounter, and we'll be using the shorter variant.

- `np.float32`, `np.float16`: single-precision and half-precision floating-point numbers. These occupy only 32 and 16 bits of memory, respectively.

  They thus trade off storage requirements for a loss of precision and range.

- `np.int32`, `np.int16`, `np.int8` represent integers using 32, 16 and 8 bits, respectively.

  They require less memory, but can represent only a smaller range of integers. For example, `np.int8` can only store integer values from -128 to 127.

- NumPy also supports complex numerical types to represent imaginary numbers. We will not be using those in this tutorial.

You can find a complete list of NumPy data types here and here.

Would we ever want to use anything other than the default data types, which in most cases are either `np.float64` and `np.int64`? These, after all, support the largest range and highest precision. This is true in general, but there are special cases where other data types need to be used:

1. *Storage requirements:* if you work with large amounts of data, for example arrays with many dimensions, you can run out of memory or storage space (when saving results to files).

   In this case, you can store data as `np.float32` instead of `np.float64`, which halves the storage requirement.

   Similarly, if you know that your integer data only takes on values between -128 and 127, you can store them as `np.int8` which consumes only 1/8 of the space compared to `np.int64`!

2. *Performance:* Some tasks simply don't require high precision or range. For example, some machine learning tasks can be performed using only 8-bit integers, and companies like Google have developed dedicated processors to considerably speed up workloads using 8-bit integers.

   Even if you are not using any specialised CPUs or GPUs, data has to be transferred from memory to the processor and this is a major performance bottleneck. The less data needs to be transferred, the better!

   In general, this is nothing you need to worry about at this point, but might become relevant once you start writing complex high-performance code.

Examples:

[79]:
```python
import numpy as np

# Explicitly specify data type
x = np.ones(1, dtype=np.float16)
x       # prints np.float16
```

[79]:
```
array([1.], dtype=float16)
```

We can use `dtype` to override the data type inferred from input data:

```
[80]: lst = [1, 2, 3]
      x = np.array(lst)       # given list of integers, creates integer array
      x.dtype                 # prints np.int64
```

```
[80]: dtype('int64')
```

```
[81]: # override inferred data type:
      # created floating-point array even if integers were given
      x = np.array(lst, dtype=np.float64)
      x.dtype                 # prints np.float64
```

```
[81]: dtype('float64')
```

```
[82]: # override inferred data type:
      # created integer array even if floats were given,
      # thus truncating input data!
      lst = [1.234, 4.567, 6.789]
      x = np.array(lst, dtype=int)
      print(x)                # prints [1, 4, 6]
      x.dtype                 # prints np.int64
```

```
[1 4 6]
```

```
[82]: dtype('int64')
```

5.10 Array storage order (advanced)

Computer memory is linear, so a multi-dimensional array is mapped to a one-dimensional block in memory. This can be done in two ways:

1. NumPy uses the so-called *row-major order* (also called *C order*, because its the same as in C programming language)
2. This is exactly the opposite of Matlab, which uses *column-major order* (also called *F order*, because its the same as in the Fortran programming language)

```
[83]: import numpy as np

      mat = np.arange(6).reshape((2,3))
      mat
```

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

```
[84]: # The matrix mat is stored in memory like this
      mat.reshape(-1, order='C')
```

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

```
[85]: # ... and NOT like this
      mat.reshape(-1, order='F')      # use order='F' to convert to column-major storage order
```

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

While this is not particularly important initially, as an advanced user you should remember that you usually want to avoid performing operations on non-contiguous blocks of memory. This can have devastating effects on performance!

[86]:
```
# Avoid operations on non-contiguous array sections such as
mat[:, 1]

# Contiguous array sections are fine
mat[1]
```

[86]:
```
array([3, 4, 5])
```

5.11 Optional exercises

Exercise 1: Broadcasting

Let `m = 2`, `n = 3` and `k = 4`. Create an array `a` with shape `(m,n)` like this:

```
a = np.arange(m*n).reshape((m,n))
```

Perform the following tasks:

1. Define the vector `b = np.arange(n) * 10` and use broadcasting to compute `c = a * b` such that `c` has shape `(m,n)`.
2. Define the vector `b = np.arange(m) * 10` and use broadcasting to compute `c = a * b` such that `c` has shape `(m,n)`.
3. Define the 3-dimensional array `b`,

   ```
   b = np.arange(m*n*k).reshape((m,k,n)) * 10
   ```

   and use broadcasting to compute `c = a * b` such that `c` has shape `(m,k,n)`.

Exercise 2: Boolean indexing

Let `m = 8` and `n = 9`. Create an array `a` with shape `(m,n)` as follows:

```
a = np.arange(m*n).reshape((m,n)) % 7
```

The `%` is the modulo operator which returns the remainder of a division of one number by another (in this case the division by 7). The resulting array `a` will therefore contain integers between 0 and 6.

1. Create a boolean array called `mask` which has the same shape as `a` and is `True` whenever an element in `a` is between 1 and 4 (inclusive).

   Hint: The character `&` works as a logical `and` operator for NumPy arrays. Alternatively, you can use the function `np.logical_and()`.
2. Compute the number of elements in `a` that satisfy this criterion.
3. Compute the average of these elements.

Exercise 3: Diagonal matrices

In this exercise, we'll create diagonal matrices using integer array indexing.

1. Create a square matrix of zeros with shape `(n,n)` for `n = 5` and `dtype = int` as its data type: `a = np.zeros((n,n), dtype=int)` Use integer array indexing to modify its diagonal to construct the following matrix, where omitted elements are zero:

$$\begin{bmatrix} 1 & & & & \\ & 2 & & & \\ & & 3 & & \\ & & & 4 & \\ & & & & 5 \end{bmatrix}$$

2. Repeat the exercise, but now use a non-squared matrix with shape `(4,5)`, and insert `1,2,...` as the values of the first diagonal above the main diagonal:

$$\begin{bmatrix} 0 & 1 & & & \\ & & 2 & & \\ & & & 3 & \\ & & & & 4 \end{bmatrix}$$

3. Repeat the exercise, but now use a matrix with shape `(6,5)`. Adapt your code so that it can handle matrix shapes `(m,n)` for cases `n > m`, `n = m` and `n < m`:

$$\begin{bmatrix} 0 & 1 & & & \\ & & 2 & & \\ & & & 3 & \\ & & & & 4 \\ & & & & \\ & & & & \end{bmatrix}$$

Exercise 4: Triangular matrices

Create a matrix of zeros with shape `(m,n)`, with `m = 4`, `n = 5`, and `dtype = int`:

```
a = np.zeros((m,n), dtype=int)
```

Transform it to an upper-triangular matrix so that it looks like this:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ & 6 & 7 & 8 & 9 \\ & & 10 & 11 & 12 \\ & & & 13 & 14 \end{bmatrix}$$

where the omitted elements are zeros. Do this without using loops.

Hint: For any upper-diagonal element at position (i, j) it holds that $j \geq i$. Create arrays of row and column indices and use these to build a mask with shape `(m,n)` which selects all upper-triangular elements. Exploit the fact that logical array operations support broadcasting!

Challenge: Modify your code to construct the following upper-triangular matrix instead:

$$\begin{bmatrix} 1 & 2 & 4 & 7 & 11 \\ & 3 & 5 & 8 & 12 \\ & & 6 & 9 & 13 \\ & & & 10 & 14 \end{bmatrix}$$

Hint: One solution is to build a lower-triangular matrix and transpose it!

Exercise 5: Row averages (ignoring NaNs)

Construct a matrix with shape `(m,n)` where `m = 5` and `n = 8` as follows:

```
a = np.arange(m*n, dtype=float).reshape((m, n))
mask = (a % 5) == 0
a[mask] = np.nan
```

Lines two and three set all elements of a which are divisible by 5 without remainder to `np.nan`, the floating-point value signalling that something is "not a number" (NaN).

Write code to perform the following tasks:

1. Define a function `rowmeans(x)` which takes a matrix `x` as an argument and returns a vector of row averages of elements in `x` which are not NaN. The return vector should therefore have the same length as the number of rows in `x`.

   Hint: Use `np.isnan()` to check whether something is NaN.
2. Compare your results to the output of `np.nanmean()` with argument `axis = 1`.
3. Use the `%timeit` magic to benchmark the run time of your routine against the (vectorised) `np.nanmean()`.

Exercise 6: Locating maxima (advanced)

Consider the following quadratic polynomial in x which is parametrised by the positive real numbers a, b and c:

$$p(x; a, b, c) = -a(x-b)^2 + c$$

Imagine that we have a set of `m = 10` such functions, each with different values for a, b and c. These parameters take on the following values:

```
m = 10

a = np.linspace(0.4, 2.0, m)
b = np.linspace(-1.0, 0.0, m)
c = np.linspace(0.0, 3.0, m)
```

The parameters for the first polynomial are thus `a[0]`, `b[0]` and `c[0]`, and similarly for the remaining polynomials.

Perform the following tasks:

1. Evaluate each polynomial on an equidistant grid of `n = 50` points on the interval `[-2.0, 2.0]`. Store the results as an $m \times n$ matrix called `pvalues`.
2. Create a (single) graph which plots all `m` polynomials.
3. Write a function `find_max(pvalues)` which accepts this matrix as an argument and returns an integer array of length `m`. Each element `i` should contain the location of the maximum for the `i`-th row of `pvalues`.
4. Add the maxima you found to the graph you created: each maximum should be marked as a black dot at the correct (x, y) coordinates.
5. Use vectorised code to perform the same task as the function `find_max()`:
   1. You actually don't have to do anything here since NumPy implements the function `np.argmax()` which does what you need (just correctly specify the `axis` argument.)
   2. Use the `%timeit` magic to compare the runtime of your `find_max()` to NumPy's `np.argmax()`.

5.12 Solutions

Solution for exercise 1

To solve this exercise, we need to make sure that the broadcasting rules can be applied by inserting an additional axis in either a or b as needed.

[87]:
```
# Define problem dimensions
m = 2
n = 3
k = 4

# Create array a
a = np.arange(m*n).reshape((m, n))
a
```

[87]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

[88]:
```
# Task 1
b = np.arange(n) * 10
# Broadcasting works as is, axis will be prepended to b
c = a * b

# If we really want, we can prepend new axis manually:
c = a * b[None]
c
```

[88]:
```
array([[  0,  10,  40],
       [  0,  40, 100]])
```

[89]:
```
# Task 2
b = np.arange(m) * 10
# Manually append new axis to b to make broadcasting work
c = a * b[:,None]
c
```

[89]:
```
array([[ 0,  0,  0],
       [30, 40, 50]])
```

[90]:
```
# Task 3
b = np.arange(n*m*k).reshape((m, k, n)) * 10
b
```

[90]:
```
array([[[  0,  10,  20],
        [ 30,  40,  50],
        [ 60,  70,  80],
        [ 90, 100, 110]],

       [[120, 130, 140],
        [150, 160, 170],
        [180, 190, 200],
        [210, 220, 230]]])
```

[91]:
```
# Need to insert new axis in between existing axes in a
c = a[:,None] * b
c
```

[91]:
```
array([[[   0,   10,   40],
        [   0,   40,  100],
        [   0,   70,  160],
        [   0,  100,  220]],

       [[ 360,  520,  700],
        [ 450,  640,  850],
        [ 540,  760, 1000],
        [ 630,  880, 1150]]])
```

Solution for exercise 2

[92]:
```
import numpy as np

# Array dimensions
m = 8
n = 9

# Create array a
a = np.arange(m*n).reshape((m,n)) % 7

# Create mask that selects all elements
# between 1 and 4
mask = (a > 0) & (a < 5)     # same as np.logical_and(a > 0, a < 5)
mask
```

[92]:
```
array([[False,  True,  True,  True,  True, False, False, False,  True],
       [ True,  True,  True, False, False, False,  True,  True,  True],
       [ True, False, False, False,  True,  True,  True,  True, False],
       [False, False,  True,  True,  True,  True, False, False, False],
       [ True,  True,  True,  True, False, False, False,  True,  True],
       [ True,  True, False, False, False,  True,  True,  True,  True],
       [False, False, False,  True,  True,  True,  True, False, False],
       [False,  True,  True,  True,  True, False, False, False,  True]])
```

We use `&` to obtain the set of elements for which the conditions `a > 0` and `a < 5` are `True` at the same time.

Boolean arrays only contain values `False` and `True`. However, arithmetic operations automatically interpret `False` as 0 and `True` as 1, so we can simply sum over the array to obtain the number of elements that are `True`:

[93]:
```
# Count number of elements
nmask = np.sum(mask)
nmask
```

[93]:
```
41
```

Finally, we compute the average as follows:

[94]:
```
# Compute average of elements selected by mask
# We do this by summing over selected elements and
# dividing by the number of such elements.
mean = np.sum(a[mask]) / nmask
mean
```

[94]:
```
2.4634146341463414
```

Solution for exercise 3

[95]:
```
import numpy as np

# Square matrix with 1,2,...n diagonal

n = 5
# Create matrix of zeros
mat = np.zeros((n,n), dtype=int)

# Create row and column indices for the diagonal elements
irow = np.arange(n)
icol = irow             # column indices are the same since
                        # this is the diagonal of a square matrix

# Vector to insert as diagonal
diag = np.arange(1, n+1)

# Set new diagonal
mat[irow, icol] = diag

mat         # print result
```

[95]:
```
array([[1, 0, 0, 0, 0],
       [0, 2, 0, 0, 0],
       [0, 0, 3, 0, 0],
       [0, 0, 0, 4, 0],
       [0, 0, 0, 0, 5]])
```

You can achieve the same result with NumPy's `np.diag()` function:

[96]:
```
np.diag(np.arange(n) + 1)
```

[96]:
```
array([[1, 0, 0, 0, 0],
       [0, 2, 0, 0, 0],
       [0, 0, 3, 0, 0],
       [0, 0, 0, 4, 0],
       [0, 0, 0, 0, 5]])
```

For task 2, we need to specify row and column index arrays that are no longer identical:

[97]:
```
# Task 2: set upper diagonal elements of non-square matrix
import numpy as np

m = 4
n = 5

# Create matrix of zeros
mat = np.zeros((m, n), dtype=int)

# Row and column indices
irow = np.arange(m)
icol = np.arange(n-1) + 1

diag = np.arange(m) + 1

# Set new diagonal
mat[irow, icol] = diag

mat                 # print result
```

[97]:
```
array([[0, 1, 0, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 0, 3, 0],
       [0, 0, 0, 0, 4]])
```

We now create a more generic version of the code above that can handle the cases `m < n`, `m = n` and `m > n`. The above code will fail for `m > n`.

[98]:
```
import numpy as np

m = 6
n = 5

# Create matrix of zeros
mat = np.zeros((m, n), dtype=int)

# Row indices: this is the minimum of the number of rows,
#       and the number of cols-1, since the first col
#       is newer included.
irow = np.arange(min(m,n-1))

# Column indices: array needs to have same length as
# row indices. First column is omitted so we
# need to shift all indices by 1.
icol = np.arange(len(irow)) + 1

# Number of elements to insert needs to be identical
# to length of array index.
diag = np.arange(len(irow)) + 1

# Set new diagonal
mat[irow, icol] = diag

mat                 # print result
```

[98]:
```
array([[0, 1, 0, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 0, 3, 0],
       [0, 0, 0, 0, 4],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
```

For the special case of `m = n`, NumPy's `np.diag()` can also insert values at diagonals other than the main diagonal. For example, for `m = n = 5`, we can create a similar matrix as follows:

[99]:
```
n = 5
# the k argument tells diag() which diagonal to manipulate.
# k = 1 corresponds to first diagonal above
# the main diagonal.
np.diag(np.arange(n-1)+1, k=1)
```

[99]:
```
array([[0, 1, 0, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 0, 3, 0],
       [0, 0, 0, 0, 4],
       [0, 0, 0, 0, 0]])
```

Solution for exercise 4

[100]:
```
m = 4
n = 5

# Create arrays of valid row and column indices.
irow = np.arange(m)
icol = np.arange(n)

# Use broadcasting to identify all upper-triangular elements
mask = icol[None] >= irow[:,None]
mask
```

[100]:
```
array([[ True,  True,  True,  True,  True],
       [False,  True,  True,  True,  True],
       [False, False,  True,  True,  True],
       [False, False, False,  True,  True]])
```

[101]:
```
# Create matrix of zeros
mat = np.zeros((m, n), dtype=int)

# count number of True values in mask
ntrue = np.sum(mask)

# Values to insert into upper-triangular part:
# 1, 2, ...
values = np.arange(ntrue) + 1

mat[mask] = values
mat
```

[101]:
```
array([[ 1,  2,  3,  4,  5],
       [ 0,  6,  7,  8,  9],
       [ 0,  0, 10, 11, 12],
       [ 0,  0,  0, 13, 14]])
```

To create the second matrix, we need to make sure that the sequence of integers is arranged column-wise instead of by row, as in the code above.

This is most likely not possible to achieve with masked indexing and C-ordered arrays. If we want to use masked indexing, we instead create the transposed matrix using the same approach as above, and then transpose it to get the final matrix.

[102]:
```
# dimensions of TRANSPOSED matrix
m = 5
n = 4

# row and column indices of TRANSPOSED matrix
irow = np.arange(m)
icol = np.arange(n)

# mask to select lower-triangular elements
# of transposed matrix
mask = irow[:, None] >= icol[None]
mask
```

[102]:
```
array([[ True, False, False, False],
       [ True,  True, False, False],
       [ True,  True,  True, False],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

[103]:
```
# Create matrix of zeros
mat = np.zeros((m, n), dtype=int)

# Number of True elements in mask
ntrue = np.sum(mask)

# Values to insert into lower-triangular part:
# 1, 2, ...
values = np.arange(ntrue) + 1
mat[mask] = values

# Transpose to get final matrix
mat = mat.T
mat
```

[103]:
```
array([[ 1,  2,  4,  7, 11],
       [ 0,  3,  5,  8, 12],
       [ 0,  0,  6,  9, 13],
       [ 0,  0,  0, 10, 14]])
```

Of course, you can also solve this using the brute-force way with loops, but that does not help us practice using NumPy:

[104]:
```
m = 4
n = 5

mat = np.zeros((m, n), dtype=int)

# keep track of current value to be inserted
# into matrix
value = 1

# loop over columns
for j in range(n):
    # loop over rows
    for i in range(m):
        if j >= i:
            mat[i,j] = value
            # increment value for next applicable
            # element
            value += 1

mat
```

[104]:
```
array([[ 1,  2,  4,  7, 11],
       [ 0,  3,  5,  8, 12],
       [ 0,  0,  6,  9, 13],
       [ 0,  0,  0, 10, 14]])
```

Solution for exercise 5

One implementation of `rowmeans()` could look as follows:

[105]:
```
import numpy as np

def rowmeans(x):
    # Number of rows and columns in x
    m, n = x.shape

    # Array to store results
    means = np.zeros(m)

    # Loop over rows
    for i in range(m):
        # number of non-NaN elements in current row
        count = 0
        # sum of non-Nan elements in current row
        s = 0.0
        for j in range(n):
            value = x[i,j]
            if not np.isnan(value):
                s += value
                # increments number of non-NaN elements
                count += 1

        # compute mean, store in output vector
        means[i] = s / count

    return means
```

We create the array given in the exercise to test our function:

[106]:
```
m = 5
n = 8
a = np.arange(m*n, dtype=float).reshape((m, n))
mask = (a % 5) == 0
a[mask] = np.nan
a
```

[106]:
```
array([[nan,  1.,  2.,  3.,  4., nan,  6.,  7.],
       [ 8.,  9., nan, 11., 12., 13., 14., nan],
       [16., 17., 18., 19., nan, 21., 22., 23.],
       [24., nan, 26., 27., 28., 29., nan, 31.],
       [32., 33., 34., nan, 36., 37., 38., 39.]])
```

As you see, NumPy indicates elements that are NaN using the string 'nan'.

[107]:
```
# call rowmeans() using test data
means = rowmeans(a)
means
```

[107]:
```
array([ 3.83333333, 11.16666667, 19.42857143, 27.5       , 35.57142857])
```

We verify our results using the NumPy routine `np.nanmean()` which implements the same functionality. Since we are computing row averages (and thus compute averages *across* columns), we need to pass in the argument `axis = 1`: axes are numbered starting at 0, so `axis = 1` refers to the second axis, i.e., the columns.

[108]:
```
means2 = np.nanmean(a, axis=1)
means2
```

[108]:
```
array([ 3.83333333, 11.16666667, 19.42857143, 27.5       , 35.57142857])
```

To test whether the results are the same, we check whether their absolute difference is below some tolerance level, in this case 10^{-8}. The routine `np.all()` evaluates to `True` if this is the case for *all* elements.

[109]:
```
np.all(np.abs(means - means2) < 1.0e-8)
```

[109]:
```
True
```

Note that we rarely want to compare floating-point numbers resulting from computations for exact equality using `==`. Floating-point has limited precision, and different operations can potentially produce different rounding errors. It is therefore unlikely that two floating-point results will be *exactly* identical.

Finally, we use the %timeit magic to benchmark our implementation against NumPy's.

[110]:
```
%timeit rowmeans(a)
```

```
58.1 µs ± 109 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[111]:
```
%timeit np.nanmean(a, axis=1)
```

```
21.4 µs ± 174 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

You may be surprised that `np.nanmean()` is only twice as fast as our implementation, but this is purely due to the small array size, as then there are only a few iterations performed in our Python loop.

Try increasing the array dimensions to `(500,800)` instead of `(5,8)` and you will see that then NumPy is about 500 times faster!

Solution for exercise 6

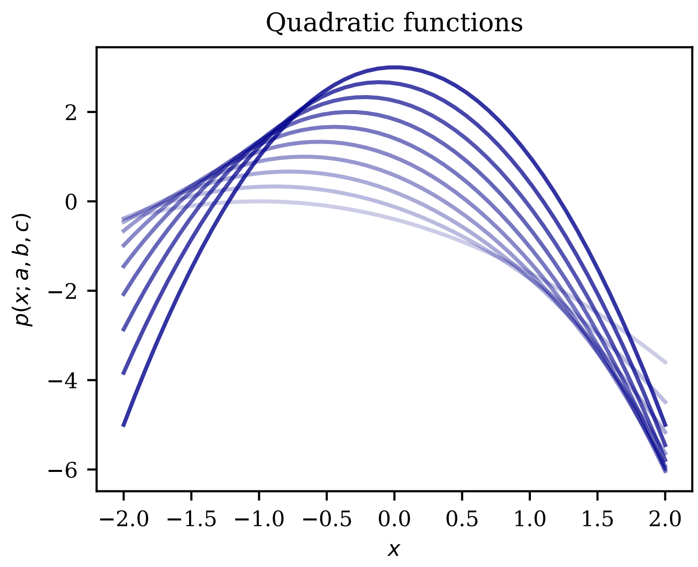
We first create the matrix `pvalues` which contains the all polynomials (for all `m = 10`) parametrisations evaluated on a common grid of x-values. Each row of this matrix represents a different polynomial parametrisation.

[112]:
```
import numpy as np
import matplotlib.pyplot as plt

m = 10
n = 50

# Parameters for each polynomial
a = np.linspace(0.4, 2.0, m)
b = np.linspace(-1.0, 0.0, m)
c = np.linspace(0.0, 3.0, m)

# grid of x values on which to evaluate polynomials
xgrid = np.linspace(-2.0, 2.0, n)

# polynomials on x, for each parameter tuple (a,b,c)
pvalues = - a[:,None] * (xgrid[None] - b[:,None])**2.0 + c[:,None]
```

Next, we create a function to plot all polynomials. We will be reusing this code, so it is convenient to encapsulate it in a function instead of copy-pasting it again and again!

[113]:
```
# function to plot quadratic polynomials
def plot_quad(xvalues, pvalues):
    # Number of different polynomials in pvalues
    m = pvalues.shape[0]

    # Use different transparency (alpha) level for each polynomial
    alphas = np.linspace(0.2, 0.8, m)

    # Plot each row against the common x-values
    for i in range(m):
        plt.plot(xvalues, pvalues[i], color='darkblue', alpha=alphas[i])

    # Label axes, figure
    plt.xlabel(r'$x$')
    plt.ylabel(r'$p(x;a,b,c)$')
    plt.title('Quadratic functions')
```

[114]:
```
# Call plotting routine, passing x-values and y-values
# as arguments
plot_quad(xgrid, pvalues)
```



5.12.1 Locating maxima using loops

Below is one possible way to implement a function that returns a vector of indices, each index storing the location of the maximum element in the corresponding row.

[115]:
```
# Function to locate the maximum value in each row
def find_max(pvalues):
    # unpack rows and columns from shape attribute
    nrow, ncol = pvalues.shape
    # Create array to store location of maximum for each row.
    # Location is an index, so choose integer array type!
    imax = np.zeros(nrow, dtype=int)

    # iterate over all row
    for i in range(nrow):
        # initial guess for location of row maximum
        jmax = 0
        # iterate over all columns, locate index of maximum
        for j in range(1, ncol):
            if pvalues[i,j] > pvalues[i,jmax]:
                # value at (i,j) is larger than
                # value at current max:
                # update jmax
                jmax = j
        # store index of maximum for current row
        imax[i] = jmax

    return imax
```

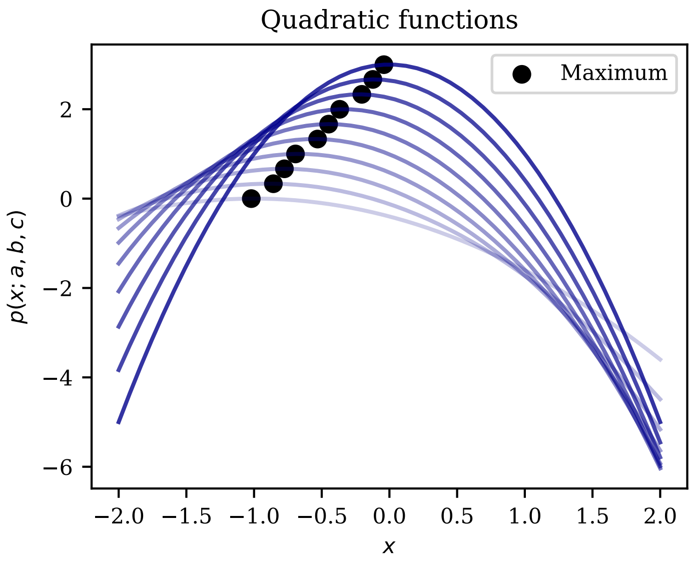
[116]:
```
# use find_max() to locale indices of each row maximum
ipmax = find_max(pvalues)
ipmax
```

```
[116]: array([12, 14, 15, 16, 18, 19, 20, 22, 23, 24])
```

We plot the polynomials using the function we defined above. We then add the maxima to the *same* plot.

```
[117]: # Recreate original plot from above
       plot_quad(xgrid, pvalues)

       # Plot maxima on top of previous graph
       ix = np.arange(m)
       plt.scatter(xgrid[ipmax], pvalues[ix,ipmax], c='black', label='Maximum')
       plt.legend()
```

```
[117]: <matplotlib.legend.Legend at 0x7f49de9a28c0>
```



5.12.2 Vectorised version

The vectorised version simply uses NumPy's `np.argmax()` function. We need to pass the argument `axis = 1` as the maximum should be computed across all columns for any given row.

```
[118]: # Find row maxima using np.argmax()

       ipmax2 = np.argmax(pvalues, axis=1)
       ipmax2
```

```
[118]: array([12, 14, 15, 16, 18, 19, 20, 22, 23, 24])
```

We verify that the results of our and NumPy's implementation are identical. Since the values here are integers, we can directly compare them using `==`.

```
[119]: # Check that these are the same as what we computed above
       np.all(ipmax == ipmax2)
```

```
[119]: True
```

Finally, we benchmark both implementations using `%timeit`. The NumPy version is approximately 100 times faster!

[120]:
```
# Time our manual implementation
%timeit find_max(pvalues)
```

```
92.8 µs ± 638 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[121]:
```
# Time NumPy's implementation
%timeit np.argmax(pvalues, axis=1)
```

```
1.32 µs ± 4.96 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

6 Handling data with pandas

6.1 Motivation

So far, we have encountered NumPy arrays as the only way to store numerical data (we mostly ignored the built-in containers provided directly in Python). However, while NumPy arrays are great for storing *homogenous* data without any particular structure, they are somewhat limited when we want to use them for high-level data analysis.

For example, we usually want to process data sets with

1. several variables;
2. multiple observations, which need not be identical across variables (some values may be missing);
3. non-homogenous data types: for examples, names need to be stored as strings, birthdays as dates and income as a floating-point number.

While NumPy can in principle handle such situations, it puts all the burden on the user. Most users would prefer to not have to deal with such low-level details.

Imagine we want to store names, birth dates and annual income for two people:

Name	Date of birth	Income
Alice	1985-01-01	30,000
Bob	1997-05-12	-

No income was reported for Bob, so it's missing. With NumPy, we could do this as follows:

[1]:
```
import numpy as np
from datetime import date

date1 = date(1985, 1, 1)            # birth date for Alice
date2 = date(1997, 5, 12)           # birth date for Bob

data = np.array([
    ['Alice', date1, 30000.0],
    ['Bob', date2, None]
])

data
```

[1]:
```
array([['Alice', datetime.date(1985, 1, 1), 30000.0],
       ['Bob', datetime.date(1997, 5, 12), None]], dtype=object)
```

[2]:
```
data.dtype          # print array data type
```

[2]:
```
dtype('O')
```

While we can create such arrays, they are almost useless for data analysis, in particular since everything is stored as a generic `object`.

Pandas was created to offer more versatile data structures that are straightforward to use for storing, manipulating and analysing heterogeneous data:

1. Data is clearly organised in *variables* and *observations*, similar to econometrics programs such as Stata.
2. Each variable is permitted to have a different data type.
3. We can use *labels* to select observations instead of having to use a linear numerical index as with NumPy.

   We could, for example, index a data set using National Insurance Numbers.
4. Pandas offers many convenient data aggregation and reduction routines that can be applied to subsets of data.

   For example, we can easily group observations by city and compute average incomes.
5. Pandas also offers many convenient data import / export functions that go beyond what's in NumPy.

Should we be using pandas at all times, then? No!

- For low-level tasks where performance is essential, use NumPy.
- For homogenous data without any particular data structure, use NumPy.
- On the other hand, if data is heterogeneous, needs to be imported from an external data source and cleaned or transformed before performing computations, use pandas.

There are numerous tutorials on pandas on the internet, so we will keep this unit short and illustrate only the main concepts. Useful references to additional material include:

- The official user guide.
- The official pandas cheat sheet which nicely illustrates the most frequently used operations.
- The official API reference with details on every pandas object and function.
- There are numerous tutorials (including videos) available on the internet. See here for a list.

6.2 Creating pandas data structures

Pandas has two main data structures:

1. `Series` represents observations of a single variable.
2. `DataFrame` is a container for several variables. You can think of each individual column of a `DataFrame` as a `Series`, and each row represents one observation.

The easiest way to create a `Series` or `DataFrame` is to create them from pre-existing data.

To access pandas data structures and routines, we need to import them first. The near-universal convention is to make pandas available using the name `pd`:

```
import pandas as pd
```

Example: Create Series from 1-dimensional NumPy array

[3]:
```
import numpy as np
import pandas as pd             # universal convention: import using pd

data = np.arange(5, 10)

# Create pandas Series from 1d array
pd.Series(data)
```

[3]:
```
0    5
1    6
2    7
3    8
4    9
```

```
dtype: int64
```

Example: Create DataFrame from NumPy array

We can create a `DataFrame` from a NumPy array:

```
[4]: # Create matrix of data
     data = np.arange(15).reshape((-1, 3))

     # Define variable (or column) names
     varnames = ['A', 'B', 'C']

     # Create pandas DataFrame from matrix
     pd.DataFrame(data, columns=varnames)
```

```
[4]:     A   B   C
     0   0   1   2
     1   3   4   5
     2   6   7   8
     3   9  10  11
     4  12  13  14
```

This code creates a `DataFrame` of three variables called `A`, `B` and `C` with 5 observations each.

Example: Create from dictionary

Alternatively, we can create a `DataFrame` from non-homogenous data as follows:

```
[5]: # Names (strings)
     names = ['Alice', 'Bob']

     # Birth dates (datetime objects)
     bdates = pd.to_datetime(['1985-01-01', '1997-05-12'])

     # Incomes (floats)
     incomes = np.array([35000, np.nan])         # code missing income as NaN

     # create DataFrame from dictionary
     pd.DataFrame({'Name': names, 'Birthdate': bdates, 'Income': incomes})
```

```
[5]:     Name  Birthdate   Income
     0  Alice 1985-01-01  35000.0
     1    Bob 1997-05-12      NaN
```

If data types differ across columns, as in the above example, it is often convenient to create the `DataFrame` by passing a dictionary as an argument. Each key represents a column name and each corresponding value contains the data for that variable.

6.3 Viewing data

With large data sets, you hardly ever want to print the entire `DataFrame`. Pandas by default limits the amount of data shown. You can use the `head()` and `tail()` methods to explicitly display a specific number of rows from the top or the end of a `DataFrame`.

To illustrate, we use a data set of 23 UK universities that contains the following variables:

- `Instititution`: Name of the institution
- `Country`: Country/nation within the UK (England, Scotland, ...)
- `Founded`: Year in which university (or a predecessor institution) was founded

- `Students`: Total number of students
- `Staff`: Number of academic staff
- `Admin`: Number of administrative staff
- `Budget`: Budget in million pounds
- `Russell`: Binary indicator whether university is a member of the Russell Group, an association of the UK's top research universities.

The data was compiled based on information from Wikipedia.

Before we read in any data, it is convenient to define a variable pointing to the directory where the data resides. We can either use a relative local path `../data`, which, however, will not work when running the notebook in some cloud environments such as Google Colab. Alternatively, we can use the full URL to the data file in the GitHub repository.

[6]:
```
# Uncomment this to use files in the local data/ directory
DATA_PATH = '../data'

# Uncomment this to load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/python-intro-PGR/main/data'
```

We can now read in the data stored in the file `universities.csv` like this:

[7]:
```
import pandas as pd

# URL to CSV file in GitHub repository
file = f'{DATA_PATH}/universities.csv'

# Load sample data set of UK universities. Individual fields are separated
# using ; so we need to pass sep=';' as an argument.
df = pd.read_csv(file, sep=';')
```

We can now display the first and last three rows:

[8]:
```
df.head(3)      # show first three rows
```

[8]:
```
                Institution   Country  Founded  Students   Staff   Admin  \
0      University of Glasgow  Scotland     1451     30805  2942.0  4003.0
1    University of Edinburgh  Scotland     1583     34275  4589.0  6107.0
2  University of St Andrews  Scotland     1413      8984  1137.0  1576.0

   Budget  Russell
0   626.5        1
1  1102.0        1
2   251.2        0
```

[9]:
```
df.tail(3)      # show last three rows
```

[9]:
```
                   Institution           Country  Founded  Students   Staff  \
20      University of Stirling          Scotland     1967      9548     NaN
21  Queen's University Belfast  Northern Ireland     1810     18438  2414.0
22          Swansea University             Wales     1920     20620     NaN

     Admin  Budget  Russell
20  1872.0   113.3        0
21  1489.0   369.2        1
22  3290.0     NaN        0
```

To quickly compute some descriptive statistics for the *numerical* variables in the `DataFrame`, we use `describe()`:

[10]:
```
df.describe()
```

[10]:
```
          Founded      Students        Staff        Admin       Budget  \
count    23.000000     23.000000    20.000000    19.000000    22.000000
mean   1745.652174  24106.782609  3664.250000  3556.736842   768.609091
std     256.992149   9093.000735  2025.638038  1550.434342   608.234948
min    1096.000000   8984.000000  1086.000000  1489.000000   113.300000
25%    1589.000000  18776.500000  2294.250000  2193.500000   340.850000
50%    1826.000000  23247.000000  3307.500000  3485.000000   643.750000
75%    1941.500000  30801.500000  4439.750000  4347.500000  1023.500000
max    2004.000000  41180.000000  7913.000000  6199.000000  2450.000000

         Russell
count  23.000000
mean    0.739130
std     0.448978
min     0.000000
25%     0.500000
50%     1.000000
75%     1.000000
max     1.000000
```

Note that this automatically ignores the columns `Institution` and `Country` as they contain strings, and computing the mean, standard deviation, etc. of a string variable does not make sense.

For categorical data, we can use `value_counts()` to tabulate the number of unique values of a variable. For example, the following code shows the number of institutions by Russell Group membership:

[11]:
```
df['Russell'].value_counts()
```

[11]:
```
1    17
0     6
Name: Russell, dtype: int64
```

Lastly, to see low-level information about the data type used in each column and the number of non-missing observations, we call `info()`:

[12]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 23 entries, 0 to 22
Data columns (total 8 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Institution  23 non-null     object
 1   Country      23 non-null     object
 2   Founded      23 non-null     int64
 3   Students     23 non-null     int64
 4   Staff        20 non-null     float64
 5   Admin        19 non-null     float64
 6   Budget       22 non-null     float64
 7   Russell      23 non-null     int64
dtypes: float64(3), int64(3), object(2)
memory usage: 1.6+ KB
```

Pandas automatically discards missing information in computations. For example, the number of academic staff is missing for several universities, so the number of *non-null* entries reported in the table above is less than 23, the overall sample size.

6.4 Indexing

Pandas supports two types of indexing:

1. Indexing by position. This is basically identical to the indexing of other Python and NumPy containers.
2. Indexing by label, i.e., by the values assigned to the row or column index. These labels need not be integers in increasing order, as is the case for NumPy. We will see how to assign labels below.

Pandas indexing is performed either by using brackets `[]`, or by using `.loc[]` for label indexing, or `.iloc[]` for positional indexing.

Indexing via `[]` can be somewhat confusing:

- specifying `df['name']` returns the column name as a `Series` object.
- On the other hand, specifying a range such as `df[5:10]` returns the *rows* associated with the *positions* 5,. . . ,9.

Example: Selecting columns

[13]:
```
import pandas as pd

# Load sample data set of UK universities
df = pd.read_csv(f'{DATA_PATH}/universities.csv', sep=';')
df['Institution']                # select a single column
```

[13]:
```
0                University of Glasgow
1              University of Edinburgh
2             University of St Andrews
3               University of Aberdeen
4           University of Strathclyde
5                                  LSE
6                                  UCL
7              University of Cambridge
8                 University of Oxford
9                University of Warwick
10             Imperial College London
11               King’s College London
12            University of Manchester
13               University of Bristol
14            University of Birmingham
15     Queen Mary University of London
16                  University of York
17            University of Nottingham
18                University of Dundee
19                  Cardiff University
20               University of Stirling
21           Queen’s University Belfast
22                  Swansea University
Name: Institution, dtype: object
```

[14]:
```
df[['Institution', 'Students']]     # select multiple columns using a list
```

[14]:
```
                        Institution  Students
0             University of Glasgow     30805
1           University of Edinburgh     34275
2          University of St Andrews      8984
3            University of Aberdeen     14775
4         University of Strathclyde     22640
5                               LSE     11850
6                               UCL     41180
7           University of Cambridge     23247
8              University of Oxford     24515
9             University of Warwick     27278
10          Imperial College London     19115
11            King’s College London     32895
```

```
12         University of Manchester     40250
13            University of Bristol     25955
14         University of Birmingham     35445
15  Queen Mary University of London     20560
16               University of York     19470
17         University of Nottingham     30798
18             University of Dundee     15915
19               Cardiff University     25898
20           University of Stirling      9548
21       Queen's University Belfast     18438
22               Swansea University     20620
```

Example: Selecting rows by position

To return the rows at positions 1, 2 and 3 we use

```
[15]: df[1:4]
```

```
[15]:               Institution   Country  Founded  Students   Staff   Admin  \
      1   University of Edinburgh  Scotland     1583     34275  4589.0  6107.0
      2  University of St Andrews  Scotland     1413      8984  1137.0  1576.0
      3    University of Aberdeen  Scotland     1495     14775  1086.0  1489.0

         Budget  Russell
      1  1102.0        1
      2   251.2        0
      3   219.5        0
```

Pandas follows the Python convention that indices are 0-based, and the endpoint of a slice is not included.

6.4.1 Creating and manipulating indices

Pandas uses *labels* to index and align data. These can be integer values starting at 0 with increments of 1 for each additional element, which is the default, but they need not be. The three main methods to create/manipulate indices are:

- Create a new `Series` or `DataFrame` object with a custom index using the `index=` argument.
- `set_index(keys=['column1', ...])` uses the values of `column1` and optionally additional columns as indices, discarding the current index.
- `reset_index()` resets the index to its default value, a sequence of increasing integers starting at 0.

Creating custom indices

First, consider the following code with creates a `Series` with three elements `[10, 20, 30]` using the default index `[0,1,2]`:

```
[16]: import pandas as pd

      # Create Series with default integer index
      pd.Series([10, 20, 30])
```

```
[16]: 0    10
      1    20
      2    30
      dtype: int64
```

We can use the `index=` argument to specify a custom index, for example one containing the lower-case characters `a`, `b`, `c` as follows:

[17]:
```
# Create Series with custom index [a, b, c]
pd.Series([10, 20, 30], index=['a', 'b', 'c'])
```

[17]:
```
a    10
b    20
c    30
dtype: int64
```

Manipulating indices

To modify the index of an *existing* `Series` or `DataFrame` object, we use the methods `set_index()` and `reset_index()`. Note that these return a new object and leave the original `Series` or `DataFrame` unchanged. If we want to change the existing object, we need to pass the argument `inplace=True`.

For example, we can replace the row index and use the Roman lower-case characters `a`, `b`, `c`, … as labels instead of integers:

[18]:
```
import pandas as pd

# Read in university data
df = pd.read_csv(f'{DATA_PATH}/universities.csv', sep=';')

# Create list of lower-case letters which has same
# length as the number of observations.
index = [chr(97+i) for i in range(len(df))]     # len(df) returns number of obs.
index
```

[18]:
```
['a',
 'b',
 'c',
 'd',
 'e',
 'f',
 'g',
 'h',
 'i',
 'j',
 'k',
 'l',
 'm',
 'n',
 'o',
 'p',
 'q',
 'r',
 's',
 't',
 'u',
 'v',
 'w']
```

[19]:
```
df['index'] = index                             # create new column 'index'
df.set_index(keys=['index'], inplace=True)      # set letters as index!

# print first 3 rows using labels
df['a':'c']             # This is the same as df[:3]
```

[19]:
```
                    Institution   Country  Founded  Students   Staff   Admin  \
index
a          University of Glasgow  Scotland     1451     30805  2942.0  4003.0
```

```
b       University of Edinburgh  Scotland     1583     34275  4589.0  6107.0
c      University of St Andrews  Scotland     1413      8984  1137.0  1576.0

       Budget  Russell
index
a       626.5        1
b      1102.0        1
c       251.2        0
```

Note that when specifying a range in terms of labels, the last element *is* included! Hence the row with index c in the above example is shown.

We can reset the index to its default integer values using the `reset_index()` method:

```
[20]: # Reset index labels to default value (integers 0, 1, 2, ...) and print
      # first three rows
      df.reset_index(drop=True).head(3)
```

```
[20]:                 Institution   Country  Founded  Students   Staff   Admin  \
      0     University of Glasgow  Scotland     1451     30805  2942.0  4003.0
      1   University of Edinburgh  Scotland     1583     34275  4589.0  6107.0
      2  University of St Andrews  Scotland     1413      8984  1137.0  1576.0

         Budget  Russell
      0   626.5        1
      1  1102.0        1
      2   251.2        0
```

The `drop=True` argument tells pandas to throw away the old index values instead of storing them as a column of the resulting `DataFrame`.

6.4.2 Selecting elements

To more clearly distinguish between selection by label and by position, pandas provides the `.loc[]` and `.iloc[]` methods of indexing. To make your intention obvious, you should therefore adhere to the following rules:

1. Use `df['name']` only to select *columns* and nothing else.
2. Use `.loc[]` to select by label.
3. Use `.iloc[]` to select by position.

Selection by label

To illustrate, using `.loc[]` unambiguously indexes by label:

```
[21]: df.loc['d':'f', ['Institution', 'Students']]
```

```
[21]:                      Institution  Students
      index
      d          University of Aberdeen     14775
      e       University of Strathclyde     22640
      f                             LSE     11850
```

With `.loc[]` we can even perform slicing on column names, which is not possible with the simpler `df[]` syntax:

```
[22]: df.loc['d':'f', 'Institution':'Founded']
```

```
[22]:                      Institution   Country  Founded
      index
      d          University of Aberdeen  Scotland     1495
```

```
e      University of Strathclyde  Scotland     1964
f                            LSE   England     1895
```

This includes all the columns between `Institution` and `Founded`, where the latter is included since we are slicing by label.

Trying to pass in positional arguments will return an error for the given `DataFrame` since the index labels are a, b, c,... and not 0, 1, 2...

[23]:
```
df.loc[0:4]
```

```
TypeError: cannot do slice indexing on Index with these indexers [0] of type int
```

However, we can reset the index to its default value. Then the index labels are integers and coincide with their position, so that `.loc[]` works:

[24]:
```
df.reset_index(inplace=True, drop=True)     # reset index labels to integers,
                                            # drop original index
df.loc[0:4]
```

[24]:
```
                 Institution   Country  Founded  Students   Staff   Admin  \
0      University of Glasgow  Scotland     1451     30805  2942.0  4003.0
1    University of Edinburgh  Scotland     1583     34275  4589.0  6107.0
2   University of St Andrews  Scotland     1413      8984  1137.0  1576.0
3     University of Aberdeen  Scotland     1495     14775  1086.0  1489.0
4  University of Strathclyde  Scotland     1964     22640     NaN  3200.0

   Budget  Russell
0   626.5        1
1  1102.0        1
2   251.2        0
3   219.5        0
4   304.4        0
```

Again, the end point with label 4 is included because we are selecting by label.

Somewhat surprisingly, we can also pass boolean arrays to `.loc[]` even though these are clearly not labels:

[25]:
```
df.loc[df['Country'] == 'Scotland']
```

[25]:
```
                  Institution   Country  Founded  Students   Staff   Admin  \
0       University of Glasgow  Scotland     1451     30805  2942.0  4003.0
1     University of Edinburgh  Scotland     1583     34275  4589.0  6107.0
2    University of St Andrews  Scotland     1413      8984  1137.0  1576.0
3      University of Aberdeen  Scotland     1495     14775  1086.0  1489.0
4   University of Strathclyde  Scotland     1964     22640     NaN  3200.0
18       University of Dundee  Scotland     1967     15915  1410.0  1805.0
20     University of Stirling  Scotland     1967      9548     NaN  1872.0

    Budget  Russell
0    626.5        1
1   1102.0        1
2    251.2        0
3    219.5        0
4    304.4        0
18   256.4        0
20   113.3        0
```

Indexing via `.loc[]` supports a few more types of arguments, see the official documentation for details.

Selection by position

Conversely, if we want to select items exclusively by their position and ignore their labels, we use `.iloc[]`:

```
[26]: df.iloc[0:4, 0:2]           # select first 4 rows, first 2 columns
```

```
[26]:                  Institution   Country
      0     University of Glasgow  Scotland
      1   University of Edinburgh  Scotland
      2  University of St Andrews  Scotland
      3    University of Aberdeen  Scotland
```

Again, `.iloc[]` supports a multitude of other arguments, including boolean arrays. See the official documentation for details.

Boolean indexing

Similar to NumPy, pandas allows us to select a subset of rows in a `Series` or `DataFrame` if they satisfy some condition. The simplest use case is to create a column of boolean values, as shown earlier. This even works without explicitly using the `.loc[]` attribute:

```
[27]: # Read in university data
      df = pd.read_csv(f'{DATA_PATH}/universities.csv', sep=';')

      # Select universities NOT in Russell Group
      # (equivalent to df.loc[df['Russell'] != 1])
      df[df['Russell'] != 1]
```

```
[27]:                    Institution   Country  Founded  Students   Staff   Admin  \
      2     University of St Andrews  Scotland     1413      8984  1137.0  1576.0
      3       University of Aberdeen  Scotland     1495     14775  1086.0  1489.0
      4    University of Strathclyde  Scotland     1964     22640     NaN  3200.0
      18         University of Dundee  Scotland     1967     15915  1410.0  1805.0
      20       University of Stirling  Scotland     1967      9548     NaN  1872.0
      22           Swansea University     Wales     1920     20620     NaN  3290.0

          Budget  Russell
      2    251.2        0
      3    219.5        0
      4    304.4        0
      18   256.4        0
      20   113.3        0
      22     NaN        0
```

Multiple conditions can be combined using the `&` (logical and) or `|` (logical or) operators:

```
[28]: # Select universities NOT in Russell Group located in Wales
      df[(df['Russell'] != 1) & (df['Country'] == 'Wales')]
```

```
[28]:            Institution Country  Founded  Students  Staff   Admin  Budget  \
      22  Swansea University   Wales     1920     20620    NaN  3290.0     NaN

          Russell
      22        0
```

If we want to include rows where an observation takes on one of multiple values, the `isin()` method can be used:

```
[29]: # Select institutions located in Wales or Scotland
      df[df['Country'].isin(('Wales', 'Scotland'))]
```

[29]:
```
                  Institution   Country  Founded  Students   Staff   Admin  \
0       University of Glasgow  Scotland     1451     30805  2942.0  4003.0
1     University of Edinburgh  Scotland     1583     34275  4589.0  6107.0
2    University of St Andrews  Scotland     1413      8984  1137.0  1576.0
3      University of Aberdeen  Scotland     1495     14775  1086.0  1489.0
4   University of Strathclyde  Scotland     1964     22640     NaN  3200.0
18       University of Dundee  Scotland     1967     15915  1410.0  1805.0
19         Cardiff University     Wales     1883     25898  3330.0  5739.0
20     University of Stirling  Scotland     1967      9548     NaN  1872.0
22         Swansea University     Wales     1920     20620     NaN  3290.0

    Budget  Russell
0    626.5        1
1   1102.0        1
2    251.2        0
3    219.5        0
4    304.4        0
18   256.4        0
19   644.8        1
20   113.3        0
22     NaN        0
```

Finally, `DataFrame` implements a `query()` method which allows us to combine multiple conditions in a single string in an intuitive fashion. Column names can be used directly within this string to put restrictions on their values.

[30]:
```python
# Select institutions in England with more than 30000 students
df.query('Country == "England" & Students > 30000')
```

[30]:
```
                  Institution  Country  Founded  Students   Staff   Admin  \
6                         UCL  England     1826     41180  7700.0  5375.0
11       King's College London  England     1829     32895  5220.0  3485.0
12  University of Manchester  England     2004     40250  3849.0     NaN
14  University of Birmingham  England     1825     35445  4020.0     NaN
17  University of Nottingham  England     1798     30798  3495.0     NaN

    Budget  Russell
6   1451.1        1
11   902.0        1
12  1095.4        1
14   673.8        1
17   656.5        1
```

6.5 Aggregation, reduction and transformation

Similar to NumPy, pandas supports data aggregation and reduction functions such as computing sums or averages. Unlike NumPy, these operations can be applied to subsets of the data which have been grouped according to some criterion. For example, for the university data set we used earlier, it is straightforward to compute the average number of students *by country*.

6.5.1 Working with entire DataFrames

The simplest way to perform data reduction is to invoke the desired routine on the entire `DataFrame`:

[31]:
```python
import pandas as pd

df = pd.read_csv(f'{DATA_PATH}/universities.csv', sep=';')
df.mean(numeric_only=True)
```

```
[31]: Founded       1745.652174
      Students     24106.782609
      Staff         3664.250000
      Admin         3556.736842
      Budget         768.609091
      Russell          0.739130
      dtype: float64
```

Methods such as `mean()` are by default applied column-wise to each column. The `numeric_only=True` argument is used to discard all non-numeric columns (depending on the version of pandas, `mean()` will issue a warning otherwise).

One big advantage over NumPy is that missing values (represented by `np.nan`) are automatically ignored:

```
[32]: # mean() automatically drops 3 missing observations
      df['Staff'].mean()
```

```
[32]: 3664.25
```

6.5.2 Splitting and grouping

Applying aggregation functions to the entire `DataFrame` is similar to what we can do with NumPy. The added flexibility of pandas becomes obvious once we want to apply these functions to subsets of data, i.e., groups which we can define based on values or index labels.

For example, we can easily group our universities by country using `groupby()`:

```
[33]: import pandas as pd

      df = pd.read_csv(f'{DATA_PATH}/universities.csv', sep=';')

      # Group observations by country (Scotland, England, etc.)
      groups = df.groupby(['Country'])
```

Here `groups` is a special pandas objects which can subsequently be used to process group-specific data. To compute the group-wise averages, we can simply run

```
[34]: groups.mean(numeric_only=True)
```

```
[34]:                       Founded      Students        Staff        Admin  \
      Country
      England           1745.923077  27119.846154  4336.692308  4112.000000
      Northern Ireland  1810.000000  18438.000000  2414.000000  1489.000000
      Scotland          1691.428571  19563.142857  2232.800000  2864.571429
      Wales             1901.500000  23259.000000  3330.000000  4514.500000

                             Budget   Russell
      Country
      England           1001.700000  1.000000
      Northern Ireland   369.200000  1.000000
      Scotland           410.471429  0.285714
      Wales              644.800000  0.500000
```

Groups support column indexing: if we want to only compute the total number of students for each country in our sample, we can do this as follows:

```
[35]: groups['Students'].sum()
```

```
[35]: Country
      England             352558
      Northern Ireland     18438
      Scotland            136942
      Wales                46518
      Name: Students, dtype: int64
```

There are numerous routines to aggregate grouped data, for example:

- `mean()`: averages within each group
- `sum()`: sum values within each group
- `std()`, `var()`: within-group standard deviation and variances
- `size()`: number of observations in each group
- `first()`, `last()`: first and last elements in each group
- `min()`, `max()`: minimum and maximum elements within a group

Example: Number of elements within each group

```
[36]: groups.size()       # return number of elements in each group
```

```
[36]: Country
      England             13
      Northern Ireland     1
      Scotland             7
      Wales                2
      dtype: int64
```

Example: Return first element of each group

```
[37]: groups.first()      # return first element in each group
```

```
[37]:
                                     Institution  Founded  Students   Staff  \
      Country
      England                                LSE     1895     11850  1725.0
      Northern Ireland  Queen’s University Belfast     1810     18438  2414.0
      Scotland               University of Glasgow     1451     30805  2942.0
      Wales                       Cardiff University     1883     25898  3330.0

                         Admin  Budget  Russell
      Country
      England           2515.0   415.1        1
      Northern Ireland  1489.0   369.2        1
      Scotland          4003.0   626.5        1
      Wales             5739.0   644.8        1
```

We can create custom aggregation routines by calling `agg()` on the grouped object. To illustrate, we count the number of universities in each country that have more than 20,000 students:

```
[38]: import numpy as np

      groups['Students'].agg(lambda x: np.sum(x >= 20000))
```

```
[38]: Country
      England             10
      Northern Ireland     0
      Scotland             3
      Wales                2
      Name: Students, dtype: int64
```

Note that we called `agg()` only on the column `Students`, otherwise the function would be applied to every column separately, which is not what we want.

The most flexible aggregation method is `apply()` which calls a given function, passing the entire group-specific subset of data (including all columns) as an argument, and glues together the results.

Example: Aggregation with custom functions

If we want to compute the average budget per student (in pounds), we can do this as follows:

```
[39]: # Budget is in millions of pounds, rescale by 1.0e6 to get
      # pounds per student
      groups.apply(lambda x: x['Budget'].sum() / x['Students'].sum() * 1.0e6)
```

```
[39]: Country
      England             36936.050239
      Northern Ireland    20023.863760
      Scotland            20981.875539
      Wales               13861.301002
      dtype: float64
```

We couldn't have done this with `agg()`, since `agg()` never gets to see the entire chunk of data but only one column at a time.

It is possible to apply multiple functions in a single call by passing a list of functions. These can be passed as strings or as callables.

Example: Applying multiple functions at once

If we want to compute the mean and median number of students by country, we proceed as follows:

```
[40]: groups['Students'].agg(['mean', 'median'])
```

```
[40]:                         mean   median
      Country
      England           27119.846154  25955.0
      Northern Ireland  18438.000000  18438.0
      Scotland          19563.142857  15915.0
      Wales             23259.000000  23259.0
```

Note that we could have also specified these function by passing references to the corresponding NumPy functions instead:

```
[41]: groups['Students'].agg([np.mean, np.median])
```

```
[41]:                         mean   median
      Country
      England           27119.846154  25955.0
      Northern Ireland  18438.000000  18438.0
      Scotland          19563.142857  15915.0
      Wales             23259.000000  23259.0
```

Finally, a the following more advanced syntax allows us to create new column names using existing columns and some operation:

```
groups.agg(
    new_column_name1=('column_name1', 'operation1'),
    new_column_name2=('column_name2', 'operation2'),
    ...
)
```

Example: Applying multiple functions to multiple columns

The following code computes the average student numbers and the earliest year a university was founded in a single aggregation:

```
[42]: groups.agg(
          average_students=('Students', 'mean'),
          first_founded=('Founded', 'first')
      )
```

```
[42]:                   average_students  first_founded
      Country
      England               27119.846154           1895
      Northern Ireland      18438.000000           1810
      Scotland              19563.142857           1451
      Wales                 23259.000000           1883
```

This section provided only a first look at pandas's "split-apply-combine" functionality implemented via `groupby`. See the official documentation for more details.

6.5.3 Transformations

In the previous section, we combined grouping and reduction, i.e., data at the group level was reduced to a single statistic such as the mean. However, we can combine grouping with the `transform()` function which assigns the result of a computation to each observation within a group and consequently leaves the number of observations unchanged.

For example, for each observation we could compute the average number of students by the corresponding country:

```
[43]: df['Avg_Student'] = df.groupby('Country')[['Students']].transform('mean')

      # Print results for each institution
      df[['Institution', 'Country', 'Avg_Student']].head(10)
```

```
[43]:                   Institution   Country   Avg_Student
      0      University of Glasgow  Scotland  19563.142857
      1    University of Edinburgh  Scotland  19563.142857
      2   University of St Andrews  Scotland  19563.142857
      3     University of Aberdeen  Scotland  19563.142857
      4  University of Strathclyde  Scotland  19563.142857
      5                        LSE   England  27119.846154
      6                        UCL   England  27119.846154
      7    University of Cambridge   England  27119.846154
      8       University of Oxford   England  27119.846154
      9      University of Warwick   England  27119.846154
```

As you can see, instead of collapsing the `DataFrame` to only 4 observations (one for each country), the number of observations remains the same, and the number of average students is constant within each country.

When would we want to use `transform()` instead of aggregation? Such use cases arise whenever we want to perform computations that include the individual value as well as an aggregate statistic.

Example: Deviation from average budget per student

Assume that we want to compute how much each university's budget per student differs from the average budget per student in the corresponding country. We could compute this using `transform()` as follows:

[44]:
```
# Compute budget per student (multiply by 1e6 to get budget in pounds)
df['Budget_Student'] = df['Budget'] / df['Students'] * 1.0e6

# Compute deviation from country-average budget per student
df['Budget_Diff'] = df.groupby('Country')['Budget_Student'].transform(lambda x: x - np.
  ↪mean(x))

# Print relevant columns
df[['Institution', 'Country', 'Budget_Diff']].head()
```

[44]:
```
                  Institution   Country   Budget_Diff
0      University of Glasgow  Scotland    804.965611
1    University of Edinburgh  Scotland  12619.072157
2   University of St Andrews  Scotland   8428.177314
3     University of Aberdeen  Scotland  -4676.465947
4  University of Strathclyde  Scotland  -6087.412238
```

From the first row you see that the University of Glasgow has approximately 800 pounds per student more at its disposal than the average Scottish university.

6.6 Working with time series data

In economics and finance, we frequently work with time series data, i.e., observations that are associated with a particular point in time (time stamp) or a time period. pandas offers comprehensive support for such data, in particular if the time stamp or time period is used as the index of a `Series` or `DataFrame`. This section presents a few of the most important concepts, see the official documentation for a comprehensive guide.

To illustrate, let's construct a set of daily data for the first three months of 2022, i.e., the period 2022-01-01 to 2022-03-31 using the `date_range()` function (we use the data format `YYYY-MM-DD` in this section, but pandas also supports other date formats).

[45]:
```
import pandas as pd
import numpy as  np

# Create sequence of dates from 2022-01-01 to 2022-03-31
# at daily frequency
index = pd.date_range(start='2022-01-01', end='2022-03-31', freq='D')

# Use date range as index for Series with some artificial data
data = pd.Series(np.arange(len(index)), index=index)

# Print first 5 observations
data.head(5)
```

[45]:
```
2022-01-01    0
2022-01-02    1
2022-01-03    2
2022-01-04    3
2022-01-05    4
Freq: D, dtype: int64
```

6.6.1 Indexing with date/time indices

pandas implements several convenient ways to select observations associated with a particular date or a set of dates. For example, if we want to select one specific date, we can pass it as a string to `.loc[]`:

[46]:
```
# Select single observation by date
data.loc['2022-01-01']
```

[46]:
```
0
```

It is also possible to select a time period by passing a start and end point (where the end point is included, as usual with label-based indexing in pandas):

[47]:
```
# Select first 5 days
data.loc['2022-01-01':'2022-01-05']
```

[47]:
```
2022-01-01    0
2022-01-02    1
2022-01-03    2
2022-01-04    3
2022-01-05    4
Freq: D, dtype: int64
```

A particularly useful way to index time periods is a to pass a partial index. For example, if we want to select all observations from January 2022, we could use the range `'2022-01-01':'2022-01-31'`, but it is much easier to specify the partial index `'2022-01'` instead which includes all observations from January since no days were specified.

[48]:
```
# Select all observations from January 2022
data.loc['2022-01']
```

[48]:
```
2022-01-01     0
2022-01-02     1
2022-01-03     2
2022-01-04     3
2022-01-05     4
2022-01-06     5
2022-01-07     6
2022-01-08     7
2022-01-09     8
2022-01-10     9
2022-01-11    10
2022-01-12    11
2022-01-13    12
2022-01-14    13
2022-01-15    14
2022-01-16    15
2022-01-17    16
2022-01-18    17
2022-01-19    18
2022-01-20    19
2022-01-21    20
2022-01-22    21
2022-01-23    22
2022-01-24    23
2022-01-25    24
2022-01-26    25
2022-01-27    26
2022-01-28    27
2022-01-29    28
2022-01-30    29
2022-01-31    30
Freq: D, dtype: int64
```

6.6.2 Lags, differences, and other useful transformations

When working with time series data, we often need to create lags or leads of a variable (e.g., if we want to include lagged values in a regression model). In pandas, this is done using `shift()` which shifts the index by the desired number of periods (default: 1). For example, invoking `shift(1)` creates lagged observations of each column in the `DataFrame`:

```
[49]: # Lag observations by 1 period
data.shift(1)
```

```
[49]: 2022-01-01     NaN
2022-01-02     0.0
2022-01-03     1.0
2022-01-04     2.0
2022-01-05     3.0
              ...
2022-03-27    84.0
2022-03-28    85.0
2022-03-29    86.0
2022-03-30    87.0
2022-03-31    88.0
Freq: D, Length: 90, dtype: float64
```

Note that the first observation is now missing since there is no preceding observation which could have provided the lagged value.

Another useful method is `diff()` which computes the difference between adjacent observations (the period over which the difference is taken can be passed as a parameter).

```
[50]: # Compute 1-period difference
data.diff()
```

```
[50]: 2022-01-01    NaN
2022-01-02    1.0
2022-01-03    1.0
2022-01-04    1.0
2022-01-05    1.0
             ...
2022-03-27    1.0
2022-03-28    1.0
2022-03-29    1.0
2022-03-30    1.0
2022-03-31    1.0
Freq: D, Length: 90, dtype: float64
```

Note that `diff()` is identical to manually computing the difference with the lagged value like this:

```
data - data.shift()
```

Additionally, we can use `pct_change()` which computes the percentage change (the relative difference) over a given number of periods (default: 1).

```
[51]: # Compute percentage change vs. previous period
data.pct_change()
```

```
[51]: 2022-01-01         NaN
2022-01-02         inf
2022-01-03    1.000000
2022-01-04    0.500000
2022-01-05    0.333333
                 ...
2022-03-27    0.011905
```

```
2022-03-28    0.011765
2022-03-29    0.011628
2022-03-30    0.011494
2022-03-31    0.011364
Freq: D, Length: 90, dtype: float64
```

Again, this is just a convenience method that is a short-cut for manually computing the percentage change:

```
(data - data.shift()) / data.shift()
```

6.6.3 Resampling and aggregation

Another useful feature of the time series support in pandas is *resampling* which is used to group observations by time period and apply some aggregation function. This can be accomplished using the `resample()` method which in its simplest form takes a string argument that describes how observations should be grouped (`'A'` or `'Y'` for aggregation to years, `'Q'` for quarters, `'M'` for months, `'W'` for weeks, etc.).

For example, if we want to aggregate our 3 months of artificial daily data to monthly frequency, we would use `resample('M')`. This returns an object which is very similar to the one returned by `groupby()` we studied previously, and we can call various aggregation methods such as `mean()`:

[52]:
```
# Resample to monthly frequency, aggregate to mean of daily observations
# within each month
data.resample('M').mean()
```

[52]:
```
2022-01-31    15.0
2022-02-28    44.5
2022-03-31    74.0
Freq: M, dtype: float64
```

Similarly, we can use `resample('W')` to resample to weekly frequency. Below, we combine this with the aggregator `last()` to return the last observation of each week (weeks by default start on Sundays):

[53]:
```
# Return last observation of each week
data.resample('W').last()
```

[53]:
```
2022-01-02     1
2022-01-09     8
2022-01-16    15
2022-01-23    22
2022-01-30    29
2022-02-06    36
2022-02-13    43
2022-02-20    50
2022-02-27    57
2022-03-06    64
2022-03-13    71
2022-03-20    78
2022-03-27    85
2022-04-03    89
Freq: W-SUN, dtype: int64
```

6.7 Visualisation

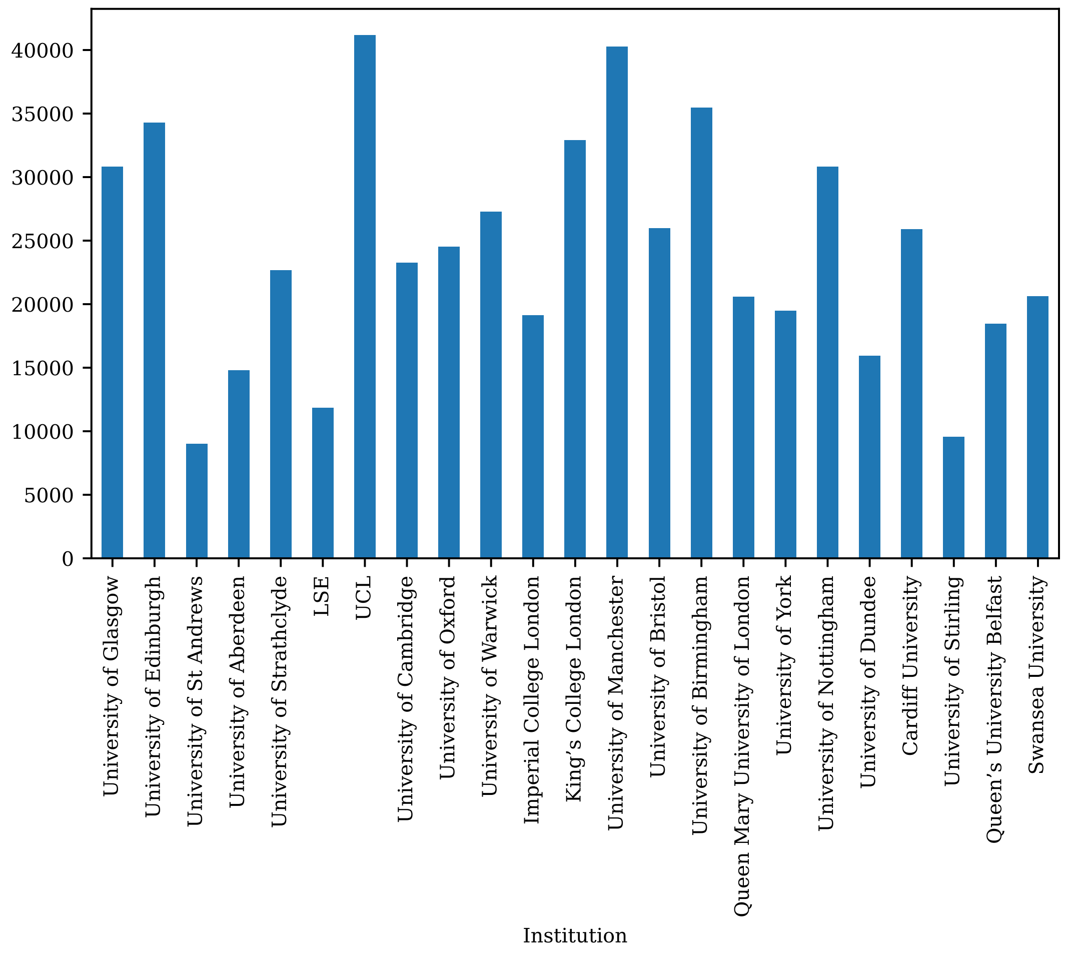
We covered plotting with Matplotlib in earlier units. Pandas itself implements some convenience wrappers around Matplotlib plotting routines which allow us to quickly inspect data stored in `DataFrames`. Alternatively, we can extract the numerical data and pass it to Matplotlib's routines manually.

Example: Creating bar charts

Let's return to our UK universities data. To plot student numbers as a bar chart, we can directly use pandas's `plot.bar()`:

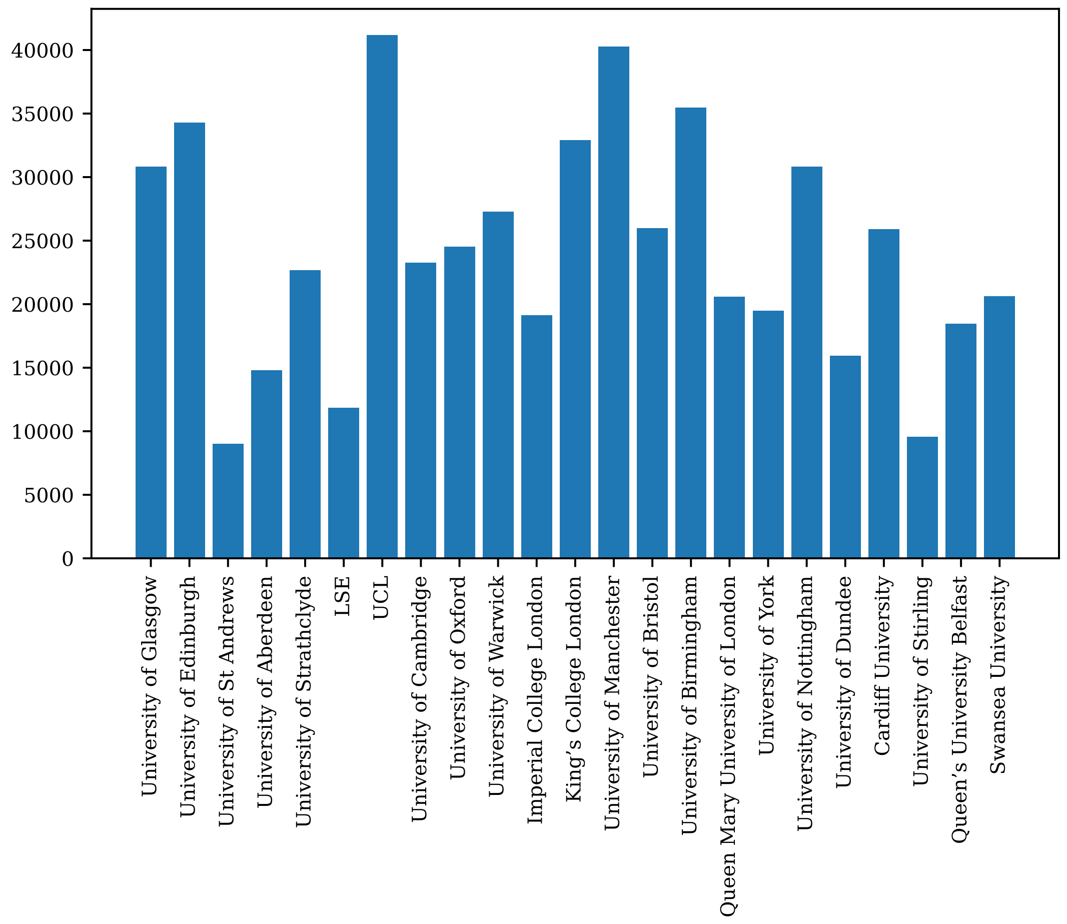
[54]:
```
import pandas as pd

# Uncomment this to use files in the local data/ directory
DATA_PATH = '../data'

# Uncomment this to load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/python-intro-PGR/main/data'

# Read universities data from CSV
df = pd.read_csv(f'{DATA_PATH}/universities.csv', sep=';')

# set institution as label so they automatically show up in plot
df2 = df.set_index(keys=['Institution'])

# Create bar chart. Alternatively, use df2['Students'].plot(kind='bar)
df2['Students'].plot.bar(figsize=(7, 4))
```

[54]:
```
<Axes: xlabel='Institution'>
```



Alternatively, we can construct the graph using Matplotlib ourselves:

[55]:
```python
import matplotlib.pyplot as plt

labels = df['Institution'].to_list()        # labels as list
values = df['Students'].to_numpy()          # data as NumPy array

# Create new figure with desired size
plt.figure(figsize=(7, 4))

# Create bar chart
plt.bar(labels, values)

# Rotate ticks
plt.tick_params(axis='x', labelrotation=90)
```



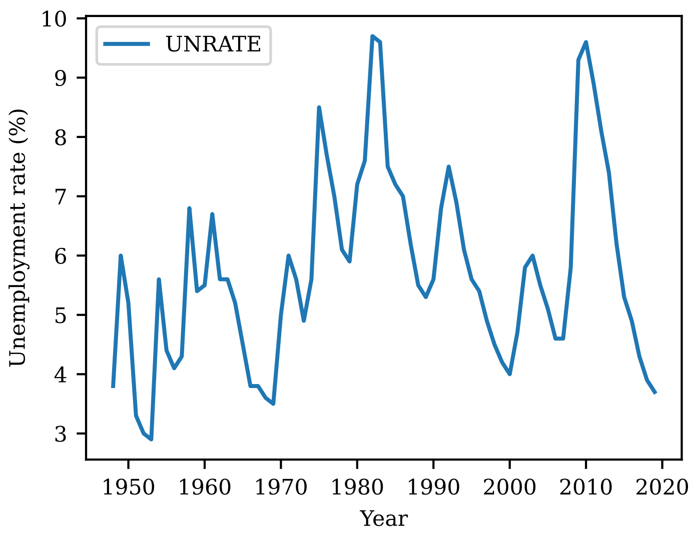
Sometimes Matplotlib's routines directly work with pandas's data structures, sometimes they don't. In cases where they don't, we can convert a `DataFrame` or `Series` object to a NumPy array using the `to_numpy()` method, or convert a `Series` to a Python list using `to_list()`, as illustrated in the example above.

Example: Plotting timeseries data

To plot timeseries-like data, we can use the `plot()` method which optionally accepts arguments to specify which columns should be used for the x-axis and which for the y-axis. We illustrate this using the US unemployment rate at annual frequency.

[56]:
```python
import numpy as np
import pandas as pd

# Path to FRED.csv; DATA_PATH variable was defined above!
filepath = f'{DATA_PATH}/FRED.csv'

# Read CSV data
df = pd.read_csv(filepath, sep=',')

# Plot unemployment rate by year
df.plot(x='Year', y='UNRATE', ylabel='Unemployment rate (%)')
```

[56]:
```
<Axes: xlabel='Year', ylabel='Unemployment rate (%)'>
```

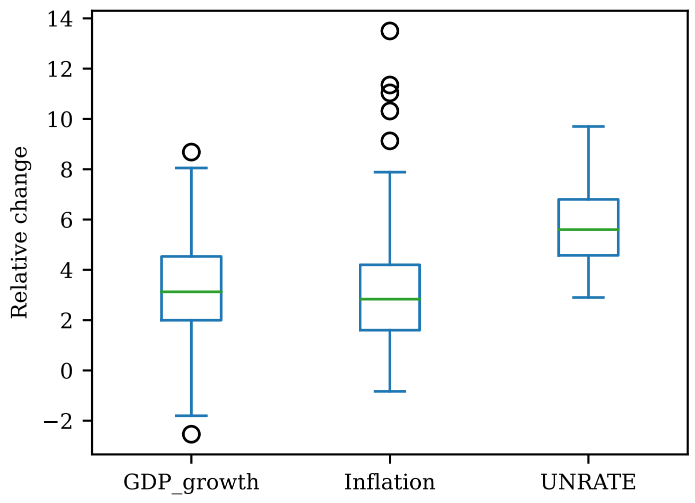


Example: Creating box plots

To quickly plot some descriptive statistics, we can use the `plot.box()` provided by pandas. We demonstrate this by plotting the distribution of post-ware GDP growth, inflation and the unemployment rate in the US:

[57]:
```python
import numpy as np
import pandas as pd

# Path to FRED.csv; DATA_PATH variable was defined above!
filepath = f'{DATA_PATH}/FRED.csv'

# Read CSV data
df = pd.read_csv(filepath, sep=',')

# Compute annual growth rates (in percent)
df['GDP_growth'] = df['GDP'].pct_change() * 100.0
df['Inflation'] = df['CPI'].pct_change() * 100.0

# Include only the following columns in plot
columns = ['GDP_growth', 'Inflation', 'UNRATE']

# Create box plot. Alternatively, use df.plot(kind='box')
df[columns].plot.box(ylabel='Relative change')
```

[57]:
```
<Axes: ylabel='Relative change'>
```

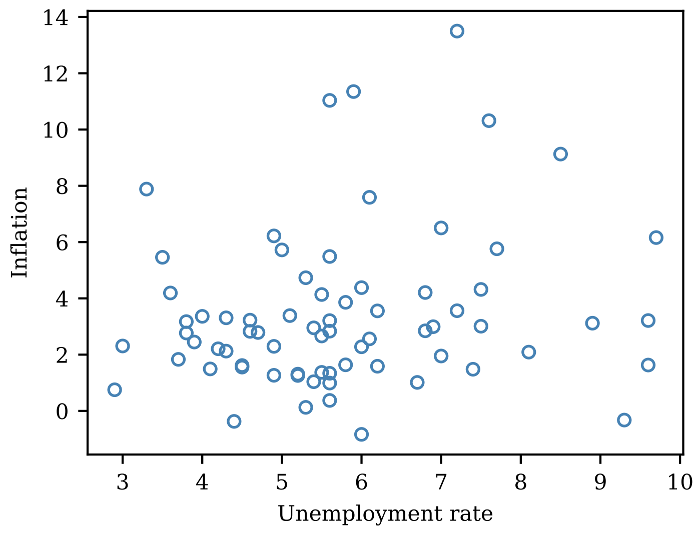


Example: Creating scatter plots

Similarly, we can generate scatter plots, plotting one column against another. To illustrate, we plot the US unemployment rate against inflation in any given year over the post-war period.

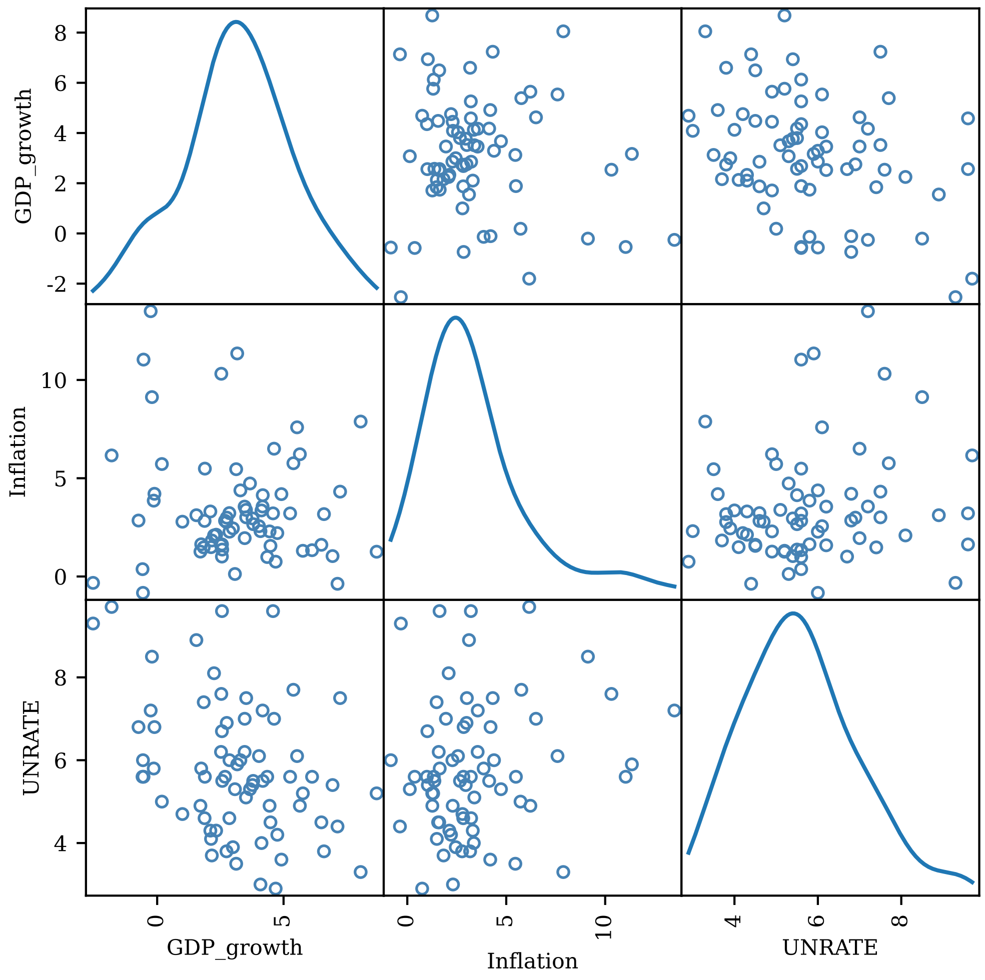
[58]:
```
# Path to FRED.csv; DATA_PATH variable was defined above!
filepath = f'{DATA_PATH}/FRED.csv'

# Read CSV data
df = pd.read_csv(filepath, sep=',')

# Compute annual inflation as growth rate of CPI
df['Inflation'] = df['CPI'].pct_change() * 100.0

df.plot.scatter(
    x='UNRATE', y='Inflation',
    color='none', edgecolor='steelblue',
    xlabel='Unemployment rate'
)
```

[58]:
```
<Axes: xlabel='Unemployment rate', ylabel='Inflation'>
```



Pandas also offers the convenience function `scatter_matrix()` which lets us easily create pairwise scatter plots for more than two variables:

[59]:
```python
from pandas.plotting import scatter_matrix

# Continue with DataFrame from previous example, compute GDP growth
df['GDP_growth'] = df['GDP'].pct_change() * 100.0

# Columns to include in plot
columns = ['GDP_growth', 'Inflation', 'UNRATE']

# Use argument diagonal='kde' to plot kernel density estimate
# in diagonal panels
axes = scatter_matrix(
    df[columns],
    figsize=(6, 6),
    diagonal='kde',            # plot kernel density along diagonal
    s=70,                      # marker size
    color='none',
    edgecolor='steelblue',
    alpha=1.0,
)
```



In general, the wrappers implemented in pandas are useful to get an idea how the data looks like. For reusable code or more complex graphs, we'll usually want to directly use Matplotlib and pass the data converted to NumPy arrays.

6.8 Optional exercises

The following exercises use data files from the `data/` folder.

Exercise 1: Basic data manipulations

In this exercise, we will perform some basic data manipulation and plot the results.

1. Load the CSV file `FRED_QTR.csv` (using `sep=','`). Set the columns `Year` and `Quarter` as (joint) indices.

   Hint: You can do this by specifying these column names in the `index_col` argument of `read_csv()`. Alternatively, you can cell `set_index()` once you have loaded the data.

2. This data comes at a quarterly frequency. Convert it to annual values by computing the average values for each year.

   Hint: Group the data by `Year` using the `groupby()` function and compute the mean on the grouped data.

3. Compute two new variables from the annualised data and add them to the `DataFrame`:

   - `Inflation`, defined as the growth rate of `CPI` (consumer price index)
   - `GDP_growth`, defined as the growth rate of `GDP`

4. Drop all rows with missing values (these show up as `NaN`).

   Hint: There is no need to manually filter out `NaN` values, you can use the `dropna()` method instead.

5. Plot the columns `GDP_growth`, `Inflation`, `UNRATE` (unemployment rate) and `LFPART` (labour force participation) using the pandas plotting routines. Use the option `subplots=True` and `layout=(2,2)` to create a 2×2 grid. See the documentation for `plot()` for details.

Exercise 2: Decade averages

Load the FRED data from the CSV file `FRED_QTR.csv` (using `sep=','`) and perform the following tasks:

1. Compute the quarterly GDP growth rate and inflation, similar to what you did in the previous exercise.
2. Add the column `Decade` which contains the decade for every observation. Use 1940 to code the 40s, 1950 for the 50s, etc.
3. We want to retain only observations for decades for which all 40 quarters are present:

   1. Group the data by `Decade` and count the number of observations using `count()`.
   2. A decade should be kept in the data set only if *all* variables have the full 40 observations.
   3. Drop all observations for which this is not the case.

4. With the remaining observations, compute the decade averages for quarterly GDP growth, inflation and the unemployment rate (`UNRATE`). Annualise the GDP growth and inflation figures by multiplying them by 4.
5. Create a bar chart that plots these three variables by decade.

Exercise 3: Group averages

Load the universities data from the CSV file `universities.csv` (using `sep=';'`) and perform the following tasks:

1. Group the data by Russell Group membership using the indicator variable `Russell`. For each group, compute the averages of the following ratios using `apply()`:

    - The ratio of academic staff (`Staff`) to students (`Students`)
    - The ratio of administrative staff (`Admin`) to students.
    - The budget (`Budget`) per student in pounds.

    Additionally, compute the number of universities is each group.

2. Repeat the task using a different approach:

    1. Compute the above ratios and add them as new columns to the initial `DataFrame`.
    2. Group the data by Russell Group membership.
    3. Compute the mean of each ratio using `mean()`.
    4. Compute the number of universities in each group using `count()`, and store the result in the column `Count` in the `DataFrame` you obtained in the previous step.

3. Create a bar chart, plotting the value for universities in and outside of the Russell Group for each of the four statistics computed above.

Exercise 4: Grouping by multiple dimensions

Load the universities data from the CSV file `universities.csv` (using `sep=';'`) and perform the following tasks:

1. Create an indicator `Pre1800` which is `True` for universities founded before the year 1800.

2. Group the data by `Country` and the value of `Pre1800`.

    Hint: You need to pass a list of column names to `groupby()`.

3. Compute the number of universities for each combination of `(Country, Pre1800)`.

4. Create a bar chart showing the number of pre- and post-1800 universities by country (i.e., create four groups of bars, each group showing one bar for pre- and one for post-1800).

5. Create a bar chart showing the number of universities by country by pre- and post-1800 period (i.e., create two groups of bars, each group showing four bars, one for each country.)

Exercise 5: Okun's law (advanced)

In this exercise, we will estimate Okun's law on quarterly data for each of the last eight decades.

Okun's law relates unemployment to the output gap. One version (see Jones: Macroeconomics, 2019) is stated as follows:

$$u_t - \overline{u}_t = \alpha + \beta \left(\frac{Y_t - \overline{Y}_t}{\overline{Y}_t} \right)$$

where u_t is the unemployment rate, $\overline{u}_t$ is the natural rate of unemployment, Y_t is output (GDP) and $\overline{Y}_t$ is potential output. We will refer to $u_t - \overline{u}_t$ as "cyclical unemployment" and to the term in parenthesis on the right-hand side as the "output gap." Okun's law says that the coefficient β is negative, i.e., cyclical unemployment is higher when the output gap is low (negative) because the economy is in a recession.

Load the FRED data from the CSV file `FRED_QTR.csv` (using `sep=','`) and perform the following tasks:

1. Compute the output gap and cyclical unemployment rate as defined above and add them as columns to the `DataFrame`.

2. Assign each observation to a decade as you did in previous exercises.
3. Write a function `regress_okun()` which accepts a `DataFrame` containing a decade-spefic subsample as the only argument, and estimates the coefficients α (the intercept) and β (the slope) of the above regression equation.

   This function should return a `DataFrame` of a single row and two columns which store the intercept and slope.

   Hint: Use NumPy's `lstsq()` to perform the regression. To regress the dependent variable `y` on regressors `X`, you need to call `lstsq(X, y)`. To include the intercept, you will manually have to create `X` such that the first column contains only ones.
4. Group the data by decade and call the `apply()` method, passing `regress_okun` you wrote as the argument.
5. Plot your results: for each decade, create a scatter plot of the raw data and overlay it with the regression line you estimated.

6.9 Solutions

These solutions illustrate *one* possible way to solve the exercises. Pandas is extremely flexible (maybe too flexible) and allows us to perform these tasks in many different ways, so your implementation might look very different.

The solutions are also provided as Python scripts in the lectures/solutions/unit06/ folder.

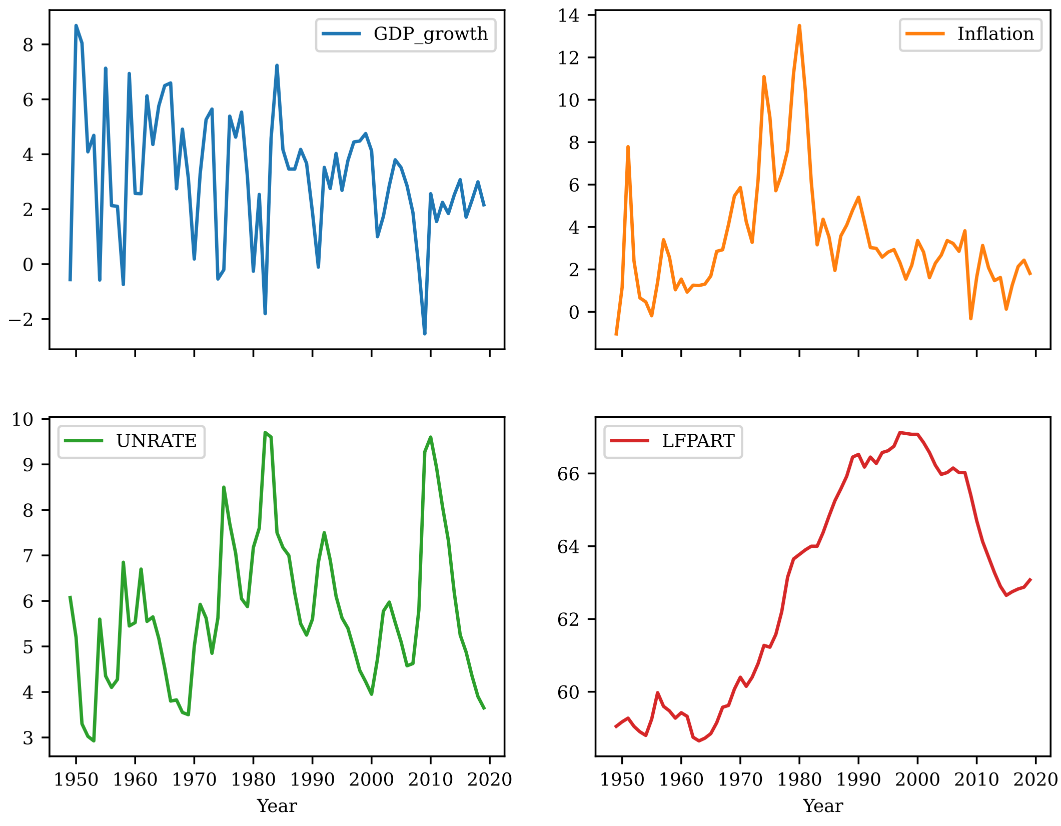
Solution for exercise 1

One possible implementation looks as follows:

[60]:
```python
import pandas as pd

# Use either local or remote path to data/ directory
# DATA_PATH = '../data'
DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/python-intro-PGR/main/data'
filepath = f'{DATA_PATH}/FRED_QTR.csv'

df = pd.read_csv(filepath, sep=',', index_col=['Year', 'Quarter'])
# Alternatively, set index columns later
# df = pd.read_csv(filepath, sep=',')
# df.set_index(keys=['Year', 'Quarter'], inplace=True)

# Convert to annual frequency
# Group by year
grp = df.groupby(['Year'])
# Compute annual data as mean of quarterly values
df_year = grp.mean()

# Alternative ways to perform the same aggregation:
# df_year = grp.agg('mean')
# df_year = grp.agg(np.mean)

# Compute CPI and GDP growth rates (in percent)
df_year['Inflation'] = df_year['CPI'].diff() / df_year['CPI'].shift() * 100.0
df_year['GDP_growth'] = df_year['GDP'].diff() / df_year['GDP'].shift() * 100.0

# Drop all rows that contain any NaNs
df_year = df_year.dropna(axis=0)

# Columns to plot
```

```
varnames = ['GDP_growth', 'Inflation', 'UNRATE', 'LFPART']
df_year.plot.line(y=varnames, subplots=True, layout=(2, 2),
                  sharex=True, figsize=(8, 6))

# Alternatively, we can call plot() directly, which
# defaults to generating a line plot:
#
# df_year.plot(y=varnames, subplots=True, layout=(2, 2),
#              sharex=True, figsize=(10, 10))
```

```
[60]: array([[<Axes: xlabel='Year'>, <Axes: xlabel='Year'>],
       [<Axes: xlabel='Year'>, <Axes: xlabel='Year'>]], dtype=object)
```



A few comments:

1. We can set the index column when loading a CSV file by passing the column names as `index_col`:

   ```
   df = pd.read_csv(filepath, sep=',', index_col=['Year', 'Quarter'])
   ```

   Alternatively, we can first load the CSV file and set the index later:

   ```
   df = pd.read_csv(filepath, sep=',')
   df.set_index(keys=['Year', 'Quarter'], inplace=True)
   ```

2. There are several ways to compute the means of grouped data:

   1. We can call `mean()` on the group object directly:

      ```
      df_year = grp.mean()
      ```

   2. Alternatively, we can call `agg()` and pass it the aggregation routine that should be applied:

      ```
      df_year = grp.agg('mean')
      df_year = grp.agg(np.mean)
      ```

Here we again have multiple options: pandas understands `'mean'` if passed as a string (which might not be the case for some other functions), or we pass an actual function such as `np.mean`.

3. The easiest way to compute differences between adjacent rows is to use the `diff()` method, which returns $x_t - x_{t-1}$. Pandas then automatically matches the correct values and sets the first observation to `NaN` as there is no preceding value to compute the difference.

   To compute a growth rate $(x_t - x_{t-1})/x_{t-1}$, we additionally need to lag a variable to get the correct period in the denominator. In pandas this is achieved using the `shift()` method (which defaults to shifting by 1 period).

Solution for exercise 2

This time we do not specify `index_cols` when reading in the CSV data since we need `Year` as a regular variable, not as the index.

We then compute the decade for each year, using the fact that // performs division with integer truncation. As an example, 1951 // 10 is 195, and (1951 // 10) * 10 = 1950, which we use to represent the 1950s.

```
[61]: import pandas as pd

# Use either local or remote path to data/ directory
# DATA_PATH = '../data'
DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/python-intro-PGR/main/data'
filepath = f'{DATA_PATH}/FRED_QTR.csv'

df = pd.read_csv(filepath, sep=',')

# Compute GDP growth rates, inflation (in percent)
df['GDP_growth'] = df['GDP'].diff() / df['GDP'].shift() * 100.0
df['Inflation'] = df['CPI'].diff() / df['CPI'].shift() * 100.0

# Assign decade using // to truncate division to
# integer part. So we have 194x // 10 = 194 for any x.
df['Decade'] = (df['Year'] // 10) * 10

grp = df.groupby(['Decade'])

# Print number of obs. by decade
print(grp.count())

# Create series that contains True for each
# decade if all variables have 40 observations.
use_decade = (grp.count() == 40).all(axis=1)
# Convert series to DataFrame, assign column name 'Keep'
df_decade = use_decade.to_frame('Keep')
# merge into original DataFrame, matching rows on value
# of column 'Decade'
df = df.merge(df_decade, on='Decade')
# Restrict data only to rows which are part of complete decade
df = df.loc[df['Keep'], :].copy()
# Drop 'Keep' column
del df['Keep']

# Compute average growth rates and unemployment rate by decade
grp = df.groupby(['Decade'])

df_avg = grp[['GDP_growth', 'Inflation', 'UNRATE']].mean()
# Convert to (approximate) annualised growth rates
df_avg['GDP_growth'] *= 4.0
df_avg['Inflation'] *= 4.0
```

```
        Year  Quarter  GDP  CPI  UNRATE  LFPART  GDPPOT  NROU  GDP_growth  \
Decade
1940       8        8    8    8       8       8       4     4           7
1950      40       40   40   40      40      40      40    40          40
1960      40       40   40   40      40      40      40    40          40
1970      40       40   40   40      40      40      40    40          40
1980      40       40   40   40      40      40      40    40          40
1990      40       40   40   40      40      40      40    40          40
2000      40       40   40   40      40      40      40    40          40
2010      40       40   40   40      40      40      40    40          40

        Inflation
Decade
1940            7
1950           40
1960           40
1970           40
1980           40
1990           40
2000           40
2010           40
```

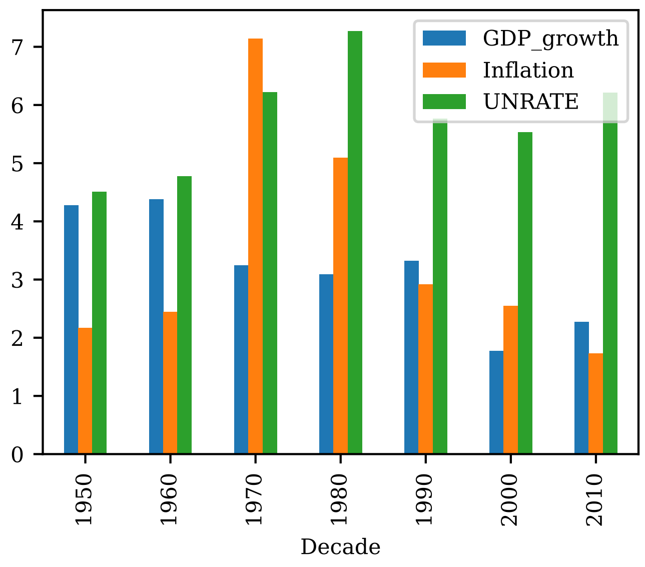
The tricky part is to keep only observations for "complete" decades that have 40 quarters of data. We see that this is not the case for the 1940s:

1. We group by `Decade` and use `count()` to determine the number of non-missing observations for each variable.
2. `count() == 40` evaluates to `True` for some variable if it has 40 observations.
3. We then use `all()` to aggregate across all variables, i.e., we require 40 observations for every variable to keep the decade.
4. Finally, we merge the indicator whether a decade should be kept in the data set using `merge()`, where we match on the value of the column `Decade`. Note that the argument to `merge()` must be a `DataFrame`, so we first have to convert our indicator data.
5. Finally, we keep only those observations which have a flag that is `True`.

The rest of the exercise is straightforward as it just repeats what we have done previously. You can create the bar chart directly with pandas as follows:

```
[62]: df_avg.plot.bar(y=['GDP_growth', 'Inflation', 'UNRATE'])
```

```
[62]: <Axes: xlabel='Decade'>
```



Solution for exercise 3

We first read in the CSV file, specifying `';'` as the field separator:

[63]:
```
import pandas as pd

# Use either local or remote path to data/ directory
# DATA_PATH = '../data'
DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/python-intro-PGR/main/data'

# Load CSV file
filepath = f'{DATA_PATH}/universities.csv'
df = pd.read_csv(filepath, sep=';')
```

For the first task we use `apply()` to create a new `Series` object for each ratio of interest.

We compute the ratios for each institution which will result in NaNs if either the numerator of denominator is missing. We thus use `np.nanmean()` to compute averages, ignoring any NaNs.

Finally, we combine all `Series` into a `DataFrame`. We do this by specifying the data passed to `DataFrame()` as a dictionary, since then we can specify the column names as keys.

[64]:
```
# Variant 1
# Compute means using apply()

grp = df.groupby(['Russell'])

# Create Series objects with the desired means
staff = grp.apply(lambda x: np.nanmean(x['Staff'] / x['Students']))
admin = grp.apply(lambda x: np.nanmean(x['Admin'] / x['Students']))
# Budget in millions of pounds
budget = grp.apply(lambda x: np.nanmean(x['Budget'] / x['Students']))
# Convert to pounds
budget *= 1.0e6
# Count number of institutions in each group.
# We can accomplish this by calling size() on the group object.
count = grp.size()

# Create a new DataFrame. Each column is a Series object.
df_all = pd.DataFrame({'Staff_Student': staff,
                       'Admin_Student': admin,
                       'Budget_Student': budget,
                       'Count': count})

df_all
```

[64]:
```
         Staff_Student  Admin_Student  Budget_Student  Count
Russell
0             0.096219       0.147762    16847.834366      6
1             0.155131       0.169079    35406.453649     17
```

For the second task, we first insert additional columns which contain the ratios of interest for each university.

We then drop all unused columns, group by the `Russell` indicator and compute the means by directly calling `mean()` on the group object.

[65]:
```
# Variant 2:
# Compute ratios first, apply aggregation later

# Create new variables directly in original DataFrame
df['Staff_Student'] = df['Staff'] / df['Students']
df['Admin_Student'] = df['Admin'] / df['Students']
```

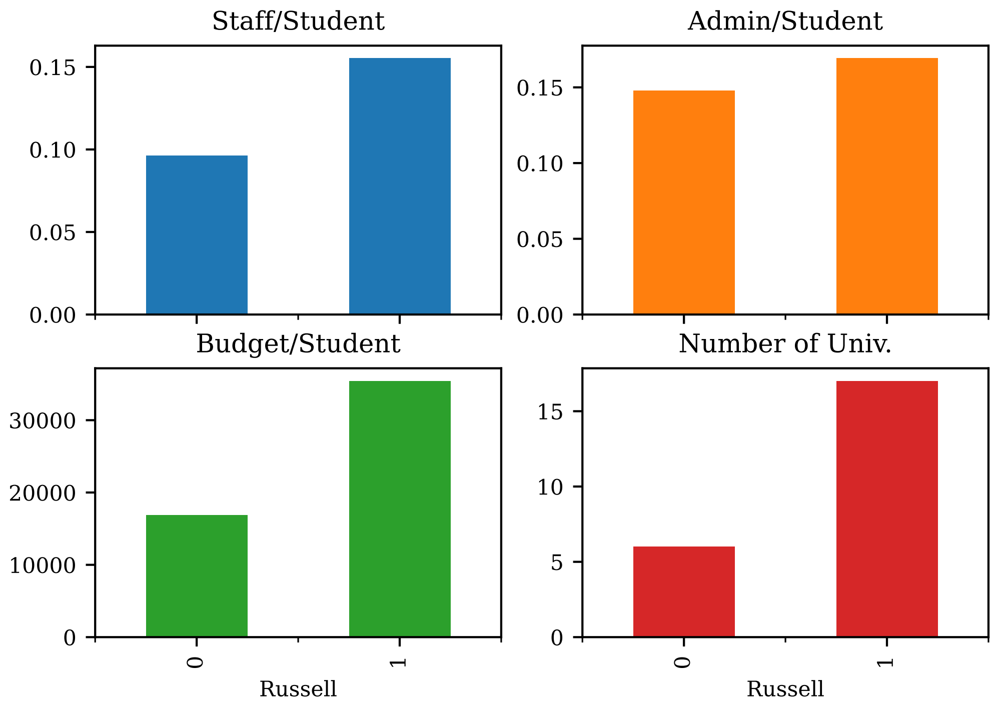
```
# Budget in pounds (original Budget is in million pounds)
df['Budget_Student'] = df['Budget'] / df['Students'] * 1.0e6

# Keep only newly constructed ratios
columns_keep = [name for name in df.columns
                if name.endswith('_Student')]
# Also keep Russell indicator
columns_keep += ['Russell']
df = df[columns_keep].copy()

# Aggregate by Russell indicator
grp = df.groupby(['Russell'])
# Count number of institutions in each group.
# We can accomplish this by calling size() on the group object.
count = grp.size()

df_all = grp.mean()
# Add counter
df_all['Count'] = count

df_all
```

[65]:
```
         Staff_Student  Admin_Student  Budget_Student  Count
Russell
0             0.096219       0.147762    16847.834366      6
1             0.155131       0.169079    35406.453649     17
```

We plot the results using pandas's `bar()` function. Since the data is of vastly different magnitudes, we specify `sharey=False` so that each panel will have its own scaling on the y-axis.

[66]:
```
# Plot results as bar charts, one panel for each variable

# Pretty titles
title = ['Staff/Student', 'Admin/Student', 'Budget/Student', 'Number of Univ.']
# Create bar chart using pandas's bar() function
df_all.plot.bar(sharey=False, subplots=True, layout=(2, 2), legend=False,
                title=title, figsize=(6, 4))
```

[66]:
```
array([[<Axes: title={'center': 'Staff/Student'}, xlabel='Russell'>,
        <Axes: title={'center': 'Admin/Student'}, xlabel='Russell'>],
       [<Axes: title={'center': 'Budget/Student'}, xlabel='Russell'>,
        <Axes: title={'center': 'Number of Univ.'}, xlabel='Russell'>]],
      dtype=object)
```



Solution for exercise 4

We create an indicator variable called `Pre1800` which is set to `True` whenever the founding year in column `Founded` is lower than 1800.

We then group the data by `Country` and `Pre1800` and count the number of universities in each group using `count()`.

[67]:
```
import pandas as pd

# Use either local or remote path to data/ directory
# DATA_PATH = '../data'
DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/python-intro-PGR/main/data'

# Load CSV file
filepath = f'{DATA_PATH}/universities.csv'
df = pd.read_csv(filepath, sep=';')

# Create mask for founding period
df['Pre1800'] = (df['Founded'] < 1800)

# Create group by country and founding period;
grp = df.groupby(['Country', 'Pre1800'])

# Number of universities by country and founding period.
# Since we are grouping by two attributes, this will create a
# Series with a multi-level (hierarchical) index
count = grp.size()

count
```

[67]:
```
Country           Pre1800
England           False      8
                  True       5
Northern Ireland  False      1
Scotland          False      3
```

```
                  True       4
Wales             False      2
dtype: int64
```

The resulting `Series` only contains values for those combinations that are actually present in the data. For example, the combination `(Wales, True)` does not show up because there are no Welsh universities founded before 1800 in our sample. We will have to "complete" the data and add zero entries in all such cases.

First, we create a `DataFrame` with countries in rows and the number of universities for the pre- and post-1800 periods in columns. To accomplish this, we need to pivot the second row index using the `unstack()` method. The `level=-1` argument tells it to use the last row index, and `fill_value=0` will assign zeros to all elements that were not present in the initial `DataFrame`, such as the combination `(Wales, True)`.

[68]:
```
# DataFrame with countries in rows, Pre-1800 indicator in columns

# Pivot inner index level to create separate columns for True/False
# values of Pre1800 indicator
df_count = count.unstack(level=-1, fill_value=0)

# Set name of column index to something pretty: this will
# be used as the legend title
df_count.columns.rename('Founding year', inplace=True)
# Rename columns to get pretty labels in legend
df_count.rename(columns={True: 'Before 1800', False: 'After 1800'},
                inplace=True)

df_count
```

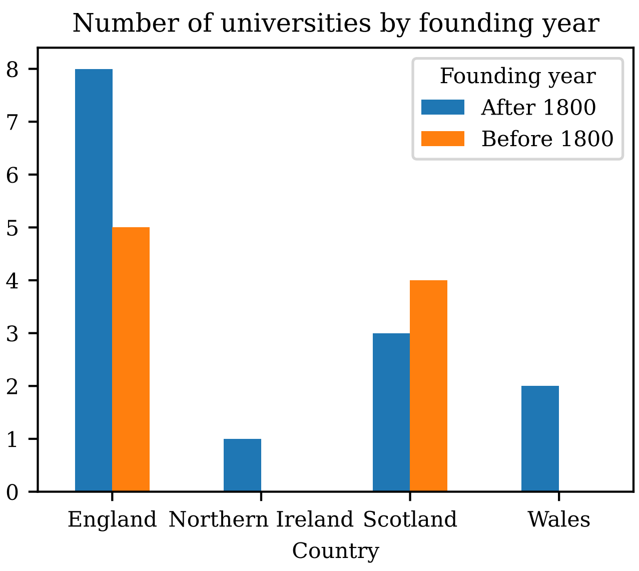
[68]:
```
Founding year     After 1800  Before 1800
Country
England                    8            5
Northern Ireland           1            0
Scotland                   3            4
Wales                      2            0
```

Whenever we use pandas's built-in plotting functions, these use index names and labels to automatically label the graph. We therefore first have to assign these objects "pretty" names.

We can then generate the bar chart as follows:

[69]:
```
# Create bar chart by country
title = 'Number of universities by founding year'
# pass rot=0 to undo the rotation of x-tick labels
# which pandas applies by default
df_count.plot.bar(xlabel='Country', rot=0, title=title)
```

[69]:
```
<Axes: title={'center': 'Number of universities by founding year'},
xlabel='Country'>
```



Note how the legend title is automatically set to the column index name and the legend labels use the column index labels.

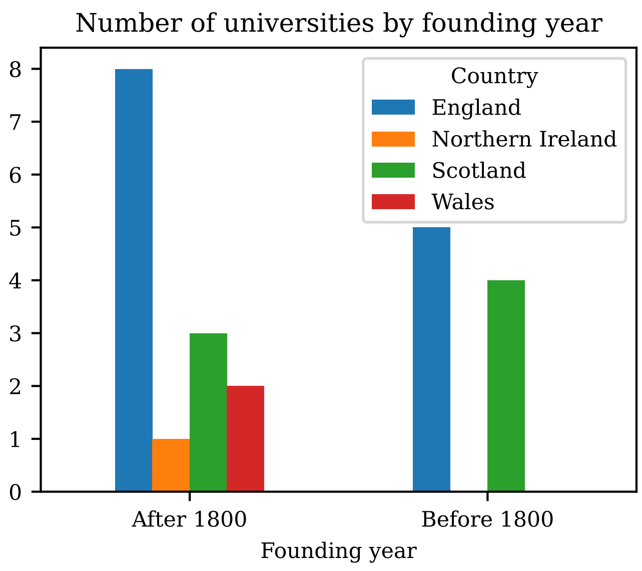
We create the second `DataFrame` with the founding period in rows and country names in columns in exactly the same way, but now call `unstack(level=0)` so that the first index level will be pivoted.

[70]:
```
# Pivot first row index level to create separate columns for each country
df_count = count.unstack(level=0, fill_value=0)

# Set index name to something pretty
df_count.index.rename('Founding year', inplace=True)
# Rename index labels to get pretty text in legend
df_count.rename(index={True: 'Before 1800', False: 'After 1800'},
                inplace=True)

df_count
```

[70]:
```
Country         England  Northern Ireland  Scotland  Wales
Founding year
After 1800            8                 1         3      2
Before 1800           5                 0         4      0
```

[71]:
```
# Create bar chart by founding year
# pass rot=0 to undo the rotation of x-tick labels
# which pandas applies by default
df_count.plot.bar(rot=0, title=title)
```

[71]:
```
<Axes: title={'center': 'Number of universities by founding year'},
xlabel='Founding year'>
```



Solution for exercise 5

This exercise is quite involved, so we will discuss it in parts. First, we write the function that will be called by `apply()` to process sub-sets of the data which belong to a single decade:

[72]:
```python
def regress_okun(x):
    # x is a DataFrame, restricted to rows for the current decade

    # Extract dependent and regressor variables
    outcome = x['unempl_gap'].to_numpy()
    GDP_gap = x['GDP_gap'].to_numpy()

    # Regressor matrix including intercept
    regr = np.ones((len(GDP_gap), 2))
    # overwrite second column with output gap
    regr[:,1] = GDP_gap

    # Solve least-squares problem (pass rcond=None to avoid a warning)
    coefs, *rest = np.linalg.lstsq(regr, outcome, rcond=None)

    # Construct DataFrame which will be returned to apply()
    # Convert data to 1 x 2 matrix
    data = coefs[None]
    columns = ['Const', 'GDP_gap']
    df_out = pd.DataFrame(data, columns=columns)

    return df_out
```

This function is passed in a single argument which is a `DataFrame` restricted to the sub-sample that is currently being processed.

- Our task is to perform the required calculations and to return the result as a `DataFrame`. `apply()` then glues together all decade-specific DataFrames to form the result of the operation.
- We first extract the relevant variables as NumPy arrays, and we create a regressor matrix which has ones in the first column. This column represents the intercept.
- We invoke `lstsq()` to run the regression. `lstsq()` returns several arguments which we mop up in the tuple `*rest` since we are only interested in the regression coefficients.

  Note that we wouldn't be using `lstsq()` to run OLS on a regular basis, but it's sufficient for this use case.

- Finally, we build the `DataFrame` to be returned by this function. It has only one row (since we ran only one regression) and two columns, one for each regression coefficient.

This was the hard part. We now need to perform some standard manipulations to prepare the data:

1. We construct the output gap (in percent), which we store in the column `GDP_gap`.
2. We construct the cyclical unemployment rate and store it in the column `unempl_gap`.
3. We determine the decade each observation belongs to using the same code as in previous exercises.
4. We then drop all unused variables from the `DataFrame` and also all observations which contain missing values.

Lastly, we can call `apply()` to run the regression for each decade.

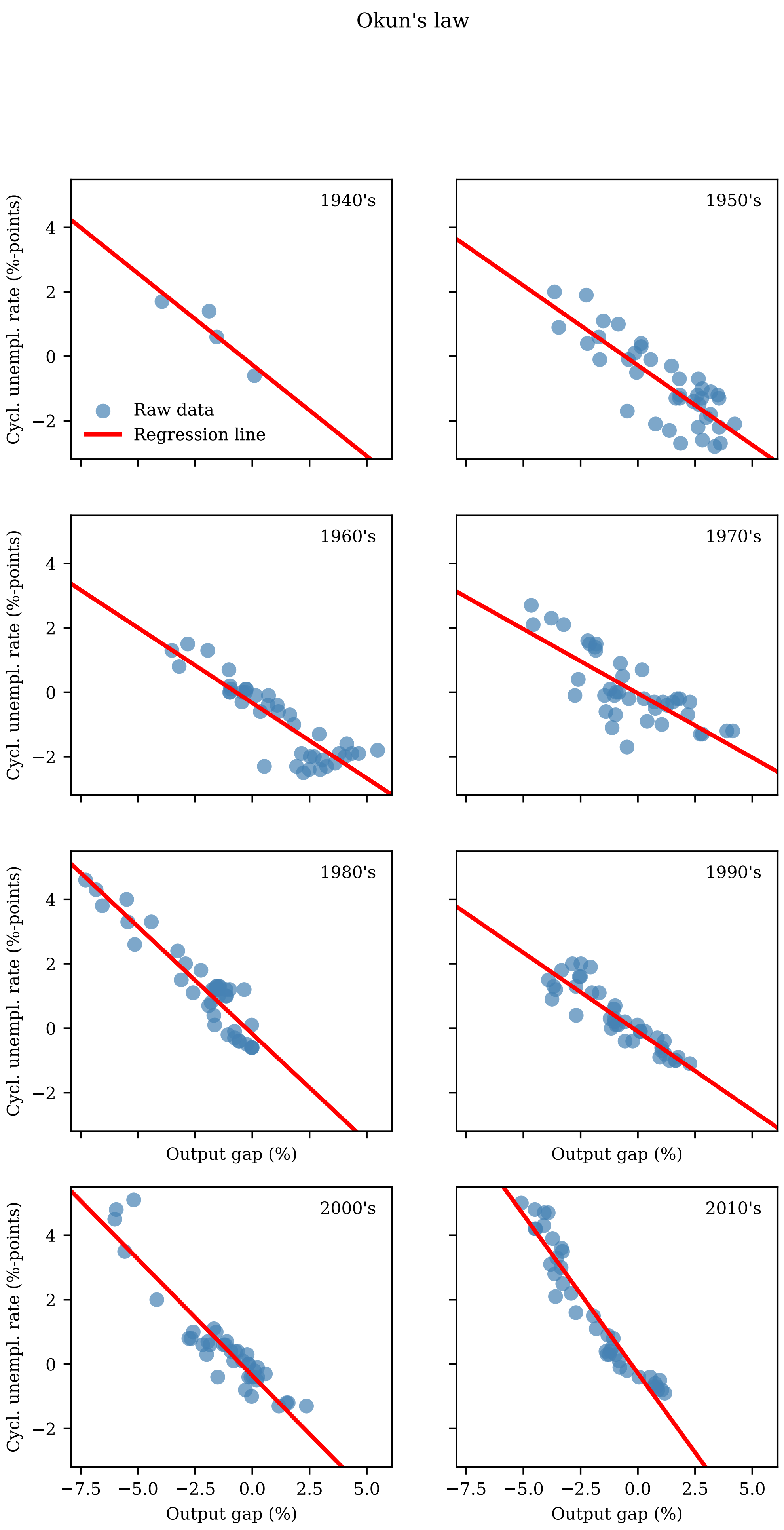
[73]:
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Use either local or remote path to data/ directory
# DATA_PATH = '../data'
DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/python-intro-PGR/main/data'

# Load CSV file
filepath = f'{DATA_PATH}/FRED_QTR.csv'
df = pd.read_csv(filepath, sep=',')

# Generate output gap (in percent)
df['GDP_gap'] = (df['GDP'] - df['GDPPOT']) / df['GDPPOT'] * 100.0

# Generate deviations of unempl. rate from natural unempl. rate
df['unempl_gap'] = df['UNRATE'] - df['NROU']

# Assign decade using // to truncate division to
# integer part. So we have 194x // 10 = 194 for any x.
df['Decade'] = (df['Year'] // 10) * 10

# Keep only variables of interest
df = df[['Decade', 'GDP_gap', 'unempl_gap']]
# Drop rows with any missing obs.
df = df.dropna(axis=0)

# Group by decade
grp = df.groupby(['Decade'])

# Apply regression routine to sub-set of data for each decade
df_reg = grp.apply(regress_okun)
# Get rid of second row index introduced by apply()
df_reg = df_reg.reset_index(level=-1, drop=True)

# Display intercept and slope coefficients
# estimated for each decade.
df_reg
```

[73]:
```
           Const   GDP_gap
Decade
1940   -0.259986 -0.567257
1950   -0.277104 -0.494637
1960   -0.331665 -0.467206
1970   -0.032063 -0.398751
1980   -0.178001 -0.666688
1990   -0.102465 -0.489427
2000   -0.355138 -0.723567
2010   -0.279333 -0.983768
```

The following code creates 8 panels of scatter plots showing the raw data and overlays a regression line for each decade.

[74]:
```
# Number of plots (= number of decades)
Nplots = len(df_reg)

# Fix number of columns, determine rows as needed
ncol = 2
nrow = int(np.ceil(Nplots / ncol))

fig, axes = plt.subplots(nrow, ncol, sharey=True, sharex=True,
                         figsize=(6, 11))

for i, ax in enumerate(axes.flatten()):

    # decade in current iteration
    decade = df_reg.index.values[i]
    # restrict DataFrame to decade-specific data
    dfi = df.loc[df['Decade'] == decade]
    # Scatter plot of raw data
    ax.scatter(dfi['GDP_gap'], dfi['unempl_gap'], color='steelblue',
               alpha=0.7, label='Raw data')
    # Extract regression coefficients
    const = df_reg.loc[decade, 'Const']
    slope = df_reg.loc[decade, 'GDP_gap']

    # plot regression line:
    # We need to provide one point and a slope to define the line to be plotted.
    ax.axline((0.0, const), slope=slope, color='red',
              lw=2.0, label='Regression line')

    # Add label containing the current decade
    ax.text(0.95, 0.95, f"{decade}'s", transform=ax.transAxes,
            va='top', ha='right')

    # Add legend in the first panel only
    if i == 0:
        ax.legend(loc='lower left', frameon=False)

    # Add x- and y-labels, but only for those panels
    # that are on the left/lower boundary of the figure
    if i >= nrow * (ncol - 1):
        ax.set_xlabel('Output gap (%)')
    if (i % ncol) == 0:
        ax.set_ylabel('Cycl. unempl. rate (%-points)')

fig.suptitle("Okun's law")
```

[74]:
```
Text(0.5, 0.98, "Okun's law")
```


Okun's law
1940's
1950's
1960's
1970's
1980's
1990's
2000's
2010's
Raw data
Regression line
Cycl. unempl. rate (%-points)
Output gap (%)
4
2
0
−2
−7.5
−5.0
−2.5
0.0
2.5
5.0

7 Data input and output

In this unit we discuss input and output, or I/O for short. We focus exclusively on I/O routines used to load and store data from files that are relevant for numerical computation and data analysis.

7.1 Input/output with NumPy

7.1.1 Loading text data

We have already encountered the most basic, and probably most frequently used NumPy I/O routine, `np.loadtxt()`. We often use files that store data as text files containing character-separated values (CSV) since virtually any application supports this data format. The most important I/O functions to process text data are:

- `np.loadtxt()`: load data from a text file.
- `np.genfromtxt()`: load data from a text file and handle missing data.
- `np.savetxt()`: save a NumPy array to a text file.

There are a few other I/O functions in NumPy, for example to write arrays as raw binary data. We won't cover them here, but you can find them in the official documentation.

Example: Load character-separated text data

Imagine we have the following tabular data from FRED, where the first two rows look as follows:

Year	GDP	CPI	UNRATE
1948	2118.5	24.0	3.8
1949	2106.6	23.8	6.0

These data are stored as character-separated values (CSV). To load this CSV file as a NumPy array, we use `loadtxt()`. As in the previous unit, it is advantageous to globally set the path to the `data/` directory that can point either to the local directory or to the `data/` directory on GitHub.

[1]:
```python
# Uncomment this to use files in the local data/ directory
DATA_PATH = '../data'

# Load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/python-intro-PGR/main/data'
```

[2]:
```python
import numpy as np

# Path to CSV file
file = f'{DATA_PATH}/FRED.csv'

# load CSV
data = np.loadtxt(file, skiprows=1, delimiter=',')

data[:2]        # Display first two rows
```

```
[2]: array([[1948. , 2118.5,   24. ,    3.8],
            [1949. , 2106.6,   23.8,    6. ]])
```

The default settings will in many cases be appropriate to load whatever CSV file we might have. However, we'll occasionally want to specify the following arguments to override the defaults:

- `delimiter`: Character used to separate individual fields (default: space).
- `skiprows=n`: Skip the first `n` rows. For example, if the CSV file contains a header with variable names, `skiprows=1` needs to be specified as NumPy by default cannot process these names.
- `dtype`: Enforce a particular data type for the resulting array.
- `encoding`: Set the character encoding of the input data. This is usually not needed, but can be required to import data with non-latin characters that are not encoded using Unicode.

While `loadtxt()` is simple to use, it quickly reaches its limits with more complex data sets. For example, when we try to load our sample of universities with `loadtxt()`, we get the following error:

```
[3]: import numpy as np

     file = f'{DATA_PATH}/universities.csv'

     # Try to load CSV data that contains strings
     # This will result in an error!
     data = np.loadtxt(file, delimiter=';', skiprows=1)
```

```
ValueError: could not convert string to float: '"University of Glasgow"'


The above exception was the direct cause of the following exception:

ValueError: could not convert string '"University of Glasgow"' to float64 at row 0, column 1.
 ↪
```

This code fails for two reasons:

1. The file contains strings and floats, and `loadtxt()` by default cannot load mixed data (e.g., strings and numerical data).
2. There are missing values (empty fields), which `loadtxt()` cannot handle either.

The simplest way to address these issues is to use pandas to load the data which we turn to in the next section.

7.1.2 Saving data to text files

To save a NumPy array to a CSV file, there is a logical counterpart to `np.loadtxt()` which is called `np.savetxt()`.

```
[4]: import numpy as np
     import os.path
     import tempfile

     # Generate three columns of 5 observations each
     data = np.linspace(0.0, 1.0, 15).reshape((3, 5))

     # create temporary directory
     d = tempfile.TemporaryDirectory()

     # path to CSV file
     file = os.path.join(d.name, 'data.csv')
```

```
# Print destination file - this will be different each time
print(f'Saving CSV file to {file}')

# Write NumPy array to CSV file. The fmt argument specifies
# that data should be saved as floating-point using a
# field width of 8 characters and 5 decimal digits.
np.savetxt(file, data, delimiter=';', fmt='%8.5f')
```

```
Saving CSV file to /tmp/tmp8hnv2edo/data.csv
```

The above code creates a 5×3 matrix of floats and stores these in the file `data.csv` using 5 significant digits.

We store the destination file in a temporary directory which we create as follows:

- Because we cannot know in advance on which system this code is run (e.g., the operating system and directory layout), we cannot hard-code a file path.
- Moreover, we do not know whether the code is run with write permissions in any particular folder.
- We work around this issue by asking the Python runtime to create a writeable temporary directory *for the system where the code is being run*.
- We use the routines in the `tempfile` module to create this temporary directory.

Of course, on your own computer you do not need to use a temporary directory, but can instead use any directory where your user has write permissions. For example, on Windows you could use something along the lines of

```
file = 'C:/Users/Path/to/file.txt'
np.savetxt(file, data, delimiter=';', fmt='%8.5f')
```

You can even use relative paths. To store a file in the current working directory it is sufficient to just pass the file name:

```
file = 'file.txt'
np.savetxt(file, data, delimiter=';', fmt='%8.5f')
```

7.2 Input/output with pandas

Pandas's I/O routines are more powerful than those implemented in NumPy:

- They support reading and writing numerous file formats.
- They support heterogeneous data without having to specify the data type in advance.
- They gracefully handle missing values.

For these reasons, it is often preferable to directly use pandas to process data instead of NumPy.

The most important routines are:

- `read_csv()`, `to_csv()`: Read or write CSV text files
- `read_fwf()`: Read data with fixed field widths, i.e., text data that does not use delimiters to separate fields.
- `read_excel()`, `to_excel()`: Read or write Excel spreadsheets
- `read_stata()`, `to_stata()`: Read or write Stata's `.dta` files.
- `read_pickle()`, `to_pickle()`: Read or write Python's binary pickle format, optionally using compression (see here for details). This should only be used to create temporary files, not to store data permanently.

For a complete list of I/O routines, see the official documentation.

To illustrate, we repeat the above examples using pandas's `read_csv()`. Since the FRED data contains only floating-point data, the result is very similar to reading in a NumPy array.

```
[5]: import pandas as pd

     # relative path to CSV file
     file = f'{DATA_PATH}/FRED.csv'

     df = pd.read_csv(file, sep=',')
     df.head(2)          # Display the first 2 rows of data
```

```
[5]:    Year     GDP   CPI  UNRATE
     0  1948  2118.5  24.0     3.8
     1  1949  2106.6  23.8     6.0
```

The difference between NumPy and pandas becomes obvious when we try to load our university data: this works out of the box:

```
[6]: import pandas as pd

     # relative path to CSV file
     file = f'{DATA_PATH}/universities.csv'

     df = pd.read_csv(file, sep=';')
     df.tail(3)      # show last 3 rows
```

```
[6]:                  Institution           Country  Founded  Students   Staff  \
     20      University of Stirling          Scotland     1967      9548     NaN
     21  Queen's University Belfast  Northern Ireland     1810     18438  2414.0
     22          Swansea University             Wales     1920     20620     NaN

          Admin  Budget  Russell
     20  1872.0   113.3        0
     21  1489.0   369.2        1
     22  3290.0     NaN        0
```

Note that missing values are correctly converted to `np.nan`.

Unlike NumPy, pandas can also process other popular data formats such as MS Excel files (or OpenDocument spreadsheets):

```
[7]: import pandas as pd

     # Excel file containing university data
     file = f'{DATA_PATH}/universities.xlsx'

     df = pd.read_excel(file, sheet_name='universities')
     df.head(3)
```

```
[7]:                Institution   Country  Founded  Students   Staff   Admin  \
     0     University of Glasgow  Scotland     1451     30805  2942.0  4003.0
     1   University of Edinburgh  Scotland     1583     34275  4589.0  6107.0
     2  University of St Andrews  Scotland     1413      8984  1137.0  1576.0

        Budget  Russell
     0   626.5        1
     1  1102.0        1
     2   251.2        0
```

The routine `read_excel()` takes the argument `sheet_name` to specify the sheet that should be read.

- Note that the Python package `openpyxl` needs to be installed in order to read files from Excel 2003 and above.
- To read older Excel files (`.xls`), you need the package `xlrd`.

Finally, we often encounter text files with fixed field widths, since this is a commonly used format in older applications (for example, fixed-width files are easy to create in Fortran). To illustrate, the fixed-width variant of our FRED data looks like this:

```
Year GDP    CPI  UNRATE
1948 2118.5   24     3.8
1949 2106.6 23.8       6
1950 2289.5 24.1     5.2
1951 2473.8   26     3.3
1952 2574.9 26.6       3
```

You see that the column `Year` occupies the first 5 characters, the `GDP` column the next 7 characters, and so on. To read such files, the width (i.e., the number of characters) has to be explicitly specified:

[8]:
```
import pandas as pd

# File name of FRED data, stored as fixed-width text
file = f'{DATA_PATH}/FRED-fixed.csv'

# field widths are passed as list to read_fwf()
df = pd.read_fwf(file, widths=[5, 7, 5, 8])
df.head(3)
```

[8]:
```
   Year     GDP   CPI  UNRATE
0  1948  2118.5  24.0     3.8
1  1949  2106.6  23.8     6.0
2  1950  2289.5  24.1     5.2
```

Here the `widths` argument accepts a list that contains the number of characters to be used for each field.

7.3 Retrieving macroeconomic / financial data from the web

7.3.1 Yahoo! Finance data

`yfinance` is a user-written library to access data from Yahoo! Finance using the public API (see the project's GitHub repository for detailed examples). This project is not affiliated with Yahoo! Finance and is intended for personal use only. Before using the library, it needs to be installed from PyPi as follows:

```
pip install yfinance
```

[9]:
```
# When running via Google Colab, uncomment and execute the following line
#! pip install yfinance
```

`yfinance` allows us to retrieve information for a single symbol via properties of the `Ticker` object, or for multiple ticker symbols at once.

Example: Retrieving data for a single symbol

We first use the API to retrieve data for a single symbol, in this case the S&P 500 index which has the (somewhat unusual) ticker symbol `^GSPS`. One can easily find the desired ticker symbol by searching for some stock, index, currency or other asset on Yahoo! Finance.

[10]:
```
import yfinance as yf

# Symbol for S&P 500 index
symbol = '^GSPC'

# Create ticker object
```

```
ticker = yf.Ticker(symbol)
```

We can now use the attributes of the `ticker` object to get all sorts of information. For example, we can get some meta data from the `info` attribute as follows:

[11]:
```
# Descriptive name and asset class
shortname = ticker.info['shortName']
quoteType = ticker.info['quoteType']

# 52-week low and high
low = ticker.info['fiftyTwoWeekLow']
high = ticker.info['fiftyTwoWeekHigh']

print(f'{shortname} is an {quoteType}')
print(f'{shortname} 52-week range: {low} - {high}')

# To see which keys are available, use the keys() method
# ticker.info.keys()
```

```
S&P 500 is an INDEX
S&P 500 52-week range: 3491.58 - 4325.28
```

We use the `history` attribute to get detailed price data. Unless we want all available data, we should select the relevant period using the `start=...` and `end=...` arguments.

[12]:
```
# Retrieve daily index values data for first quarter of this year
daily = ticker.history(start='2023-01-01', end='2023-03-31')

# Print first 5 rows
daily.head()
```

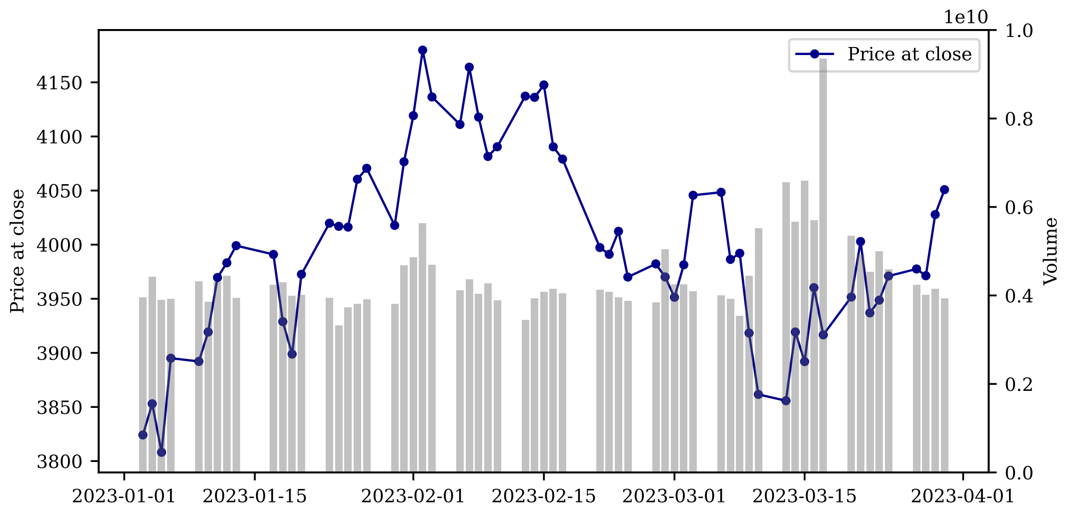
[12]:
```
                                  Open         High          Low        Close  \
Date
2023-01-03 00:00:00-05:00  3853.290039  3878.459961  3794.330078  3824.139893
2023-01-04 00:00:00-05:00  3840.360107  3873.159912  3815.770020  3852.969971
2023-01-05 00:00:00-05:00  3839.739990  3839.739990  3802.419922  3808.100098
2023-01-06 00:00:00-05:00  3823.370117  3906.189941  3809.560059  3895.080078
2023-01-09 00:00:00-05:00  3910.820068  3950.570068  3890.419922  3892.090088

                               Volume  Dividends  Stock Splits
Date
2023-01-03 00:00:00-05:00  3959140000        0.0           0.0
2023-01-04 00:00:00-05:00  4414080000        0.0           0.0
2023-01-05 00:00:00-05:00  3893450000        0.0           0.0
2023-01-06 00:00:00-05:00  3923560000        0.0           0.0
2023-01-09 00:00:00-05:00  4311770000        0.0           0.0
```

We can then use this data to plot the daily closing price and trading volume.

[13]:
```
import matplotlib.pyplot as plt

fix, ax = plt.subplots(1, 1, figsize=(7,3.5))

# Plot closing price
ax.plot(daily.index, daily['Close'], color='darkblue', marker='o', ms=3, lw=1)
ax.set_ylabel('Price at close')
ax.legend(['Price at close'], loc='upper right')

# Create secondary y-axis for trading volume
ax2 = ax.twinx()

# Plot trading volume as bar chart
```

```
ax2.bar(daily.index, daily['Volume'], color='#666666', alpha=0.4, zorder=-1, lw=0)
ax2.set_ylim((0.0, 1.0e10))
ax2.set_ylabel('Volume')
```

[13]:
```
Text(0, 0.5, 'Volume')
```



The above code uses `twinx()` to create a second (invisible) x-axis with an independent y-axis which allows us to plot the trading volume on a different scale.

Example: Retrieving data for multiple symbols

We can download trading data for multiple symbols at once using the `download()` function. Unlike the `Ticker` class, this immediately returns a `DataFrame` containing data similar to the `history` method we called previously, but now the column index contains an additional level for each ticker symbol. For example, to get the trading data for Amazon and Microsoft for the first 3 months of 2023, we proceed as follows:

[14]:
```
import yfinance as yf

# Get data for Amazon (AMZN) and Microsoft (MSFT) for first quarter of 2023
data = yf.download(('AMZN', 'MSFT'), start='2023-01-01', end='2023-03-31')
data.head()
```

```
[*********************100%***********************]  2 of 2 completed
```

[14]:
```
             Adj Close                Close                   High   \
                  AMZN        MSFT     AMZN        MSFT       AMZN
Date
2023-01-03   85.820000  238.460144  85.820000  239.580002  86.959999
2023-01-04   85.139999  228.029129  85.139999  229.100006  86.980003
2023-01-05   83.120003  221.270844  83.120003  222.309998  85.419998
2023-01-06   86.080002  223.878601  86.080002  224.929993  86.400002
2023-01-09   87.360001  226.058380  87.360001  227.119995  89.480003

                              Low                    Open              \
                  MSFT       AMZN        MSFT       AMZN        MSFT
Date
2023-01-03  245.750000  84.209999  237.399994  85.459999  243.080002
2023-01-04  232.869995  83.360001  225.960007  86.550003  232.279999
2023-01-05  227.550003  83.070000  221.759995  85.330002  227.199997
2023-01-06  225.759995  81.430000  219.350006  83.029999  223.000000
2023-01-09  231.240005  87.080002  226.410004  87.459999  226.449997
```

```
              Volume
                AMZN      MSFT
Date
2023-01-03  76706000  25740000
2023-01-04  68885100  50623400
2023-01-05  67930800  39585600
2023-01-06  83303400  43613600
2023-01-09  65266100  27369800
```

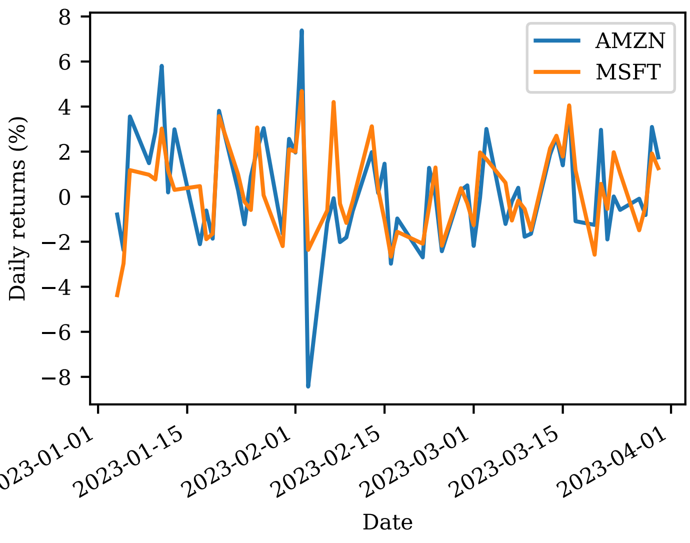
To extract data for a particular symbol, we have to take into account the hierarchical column index:

[15]:
```
# Use hierarchical indexing to get data for Amazon
data[('Close', 'AMZN')].head()
```

[15]:
```
Date
2023-01-03    85.820000
2023-01-04    85.139999
2023-01-05    83.120003
2023-01-06    86.080002
2023-01-09    87.360001
Name: (Close, AMZN), dtype: float64
```

[16]:
```
# Plot daily returns for both stocks
returns = data['Close'].pct_change() * 100.0
returns.plot(y=['AMZN', 'MSFT'], ylabel='Daily returns (%)')
```

[16]:
```
<Axes: xlabel='Date', ylabel='Daily returns (%)'>
```



7.3.2 Pandas Datareader

`pandas-datareader` is a Python library that fetches online data from multiple sources and returns them as pandas `DataFrame` objects. Despite its name, this library is not included in `pandas` and may need to be installed separately, e.g., by running

```
pip install pandas-datareader
```

The aim is to provide a uniform API to access data from multiple sources, including those covered in other sections in this unit. See the official documentation for supported data sources and how to access them.

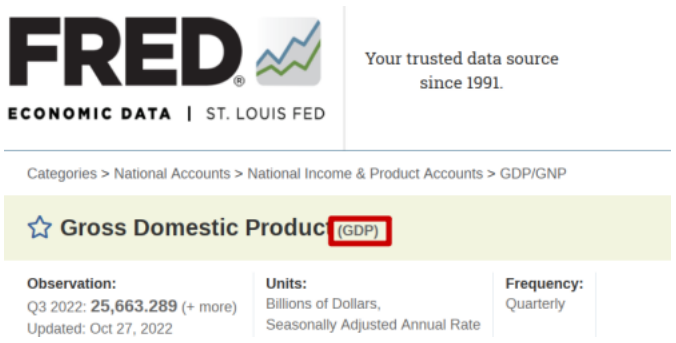
[17]:
```
# Uncomment and execute the following line if running in Google Colab
# ! pip install pandas-datareader
```

Example: Downloading data from FRED

As a first illustration, we fetch macroeconomic data from FRED, or Federal Reserve Economic Data. FRED is provided by the Federal Reserve of St. Louis and is one of the most important macroeconomic online databases (at least for US-centric data).

An alternative (but more complicated) way to access this data is via the `fredapi` library, which we examine below. With `pandas-datareader`, no API key is required to access FRED which makes using it a little simpler than `fredapi`.

In order to retrieve any data, we first need to identify the series name. This is easiest done by searching for the data on FRED using your browser and copying the series name, highlighted in red in the screenshot below.


FRED
ECONOMIC DATA | ST. LOUIS FED
Your trusted data source since 1991.
Categories > National Accounts > National Income & Product Accounts > GDP/GNP

Gross Domestic Product (GDP)

Observation:
Q3 2022: 25,663.289 (+ more)
Updated: Oct 27, 2022

Units:
Billions of Dollars,
Seasonally Adjusted Annual Rate

Frequency:
Quarterly

For example, if we want to retrieve the US Gross Domestic Product (GDP), the corresponding series name is `GDP`. The FRED web page contains additional useful information such as the time period for which the data is available, the data frequency (monthly, quarterly, annual) and whether it's seasonally adjusted.

[18]:
```
# The convention is to import this library as web
import pandas_datareader.data as web

# define start and end dates
start_date = "2000-01-01"
end_date = "2021-12-31"

# Specify series name as first and 'fred' data source as second argument
gdp = web.DataReader('GDP',
    data_source='fred',
    start=start_date,
    end=end_date
)

# Show first 3 observations
gdp.head(3)
```

[18]:
```
                  GDP
DATE
2000-01-01  10002.179
2000-04-01  10247.720
2000-07-01  10318.165
```

We can also fetch multiple series at the same time, for example the CPI (`CPIAUCSL`) and the unemployment rate (`UNRATE`).

[19]:
```
# without an explicit end argument, the latest available data
# will be retrieved
data = web.DataReader(['CPIAUCSL', 'UNRATE'],
    data_source='fred',
    start='2020-01-01'
)

data.head(3)
```

[19]:
```
            CPIAUCSL  UNRATE
DATE
2020-01-01   259.037     3.5
2020-02-01   259.248     3.5
2020-03-01   258.124     4.4
```

7.3.3 FRED: Federal Reserve Economic Data (optional)

FRED, provided by the Federal Reserve of St. Louis, is one of the most important macroeconomic online databases (at least for US-centric data). `fredapi` is a Python API for the FRED data which provides a wrapper for the FRED web service (see also the project's GitHub page).

Before accessing FRED, you need to install `fredapi` into your Python environment as follows:

```
pip install --no-deps fredapi
```

Important:

- The `--no-deps` argument might be required for Anaconda users as otherwise the conda-provided versions of `numpy` and `pandas` could be overwritten.
- Anaconda users should *not* use the `fredapi` package provided in `conda-forge` as at the time of this writing it is outdated and will not work.

To use FRED, you additional need an API key which can be requested at `https://fred.stlouisfed.org/docs/api/api_key.html`. Unlike with some other APIs we discuss below, it is not possible to make a request without a key. Once you have a key, you can specify it in several ways:

1. On your local machine, set the environment variable `FRED_API_KEY` to store the key and it will be picked up automatically. This only works if you run a Python environment locally.
2. Store it in a file and pass the file name when creating a `Fred` instance:

   ```
   from fredapi import Fred
   fred = Fred(api_key_file='path_to_file')
   ```

3. Pass the string containing the API key as a parameter:

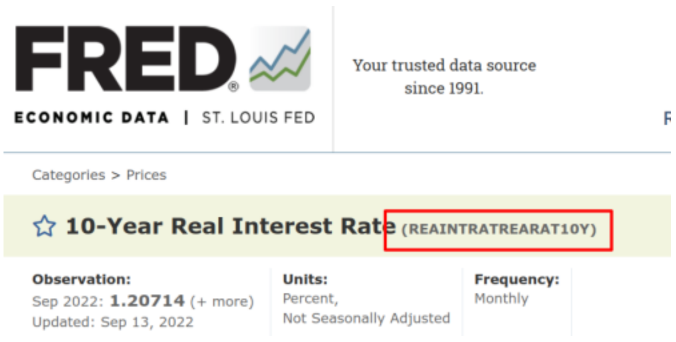
   ```
   from fredapi import Fred
   fred = Fred(api_key='INSERT API KEY HERE')
   ```

The following code assumes that the `FRED_API_KEY` variable has been set up and might not work in your environment if that is not the case.

Example: Retrieve the 10-year real interest rate

[20]:
```
# Uncomment the following to install fredapi in your local or
# cloud-hosted Python environment (e.g.,  Google Colab)

#! pip install --no-deps fredapi
```

In order to retrieve any data, we first need to identify the series name. This is easiest done by searching for the data on FRED using your browser and copying the series name, highlighted in red in the screenshot below.


Your trusted data source since 1991.

Categories > Prices

10-Year Real Interest Rate (REAINTRATREARAT10Y)

Observation: Sep 2022: **1.20714** (+ more) Updated: Sep 13, 2022

Units: Percent, Not Seasonally Adjusted

Frequency: Monthly

For example, if we want to retrieve the 10-year real interest rate, the corresponding series name is REAINTRATREARAT10Y. The FRED web page contains additional useful information such as the time period for which the data is available, the data frequency (monthly, quarterly, annual) and whether it's seasonally adjusted.

To download and plot the 10-year real interest rate, we proceed as follows:

[21]:
```python
from fredapi import Fred

# Create instance assuming API key is stored as environment variable
fred = Fred()

# or specify API key directly
# fred = Fred(api_key='INSERT API KEY HERE')

# Download observations starting from the year 2000 onward
series = fred.get_series('REAINTRATREARAT10Y',
    observation_start='2000-01-01'
)
```

[22]:
```python
# Print first 5 observations
series.head(5)
```

[22]:
```
2000-01-01    3.411051
2000-02-01    3.513343
2000-03-01    3.440347
2000-04-01    3.202967
2000-05-01    3.360531
dtype: float64
```

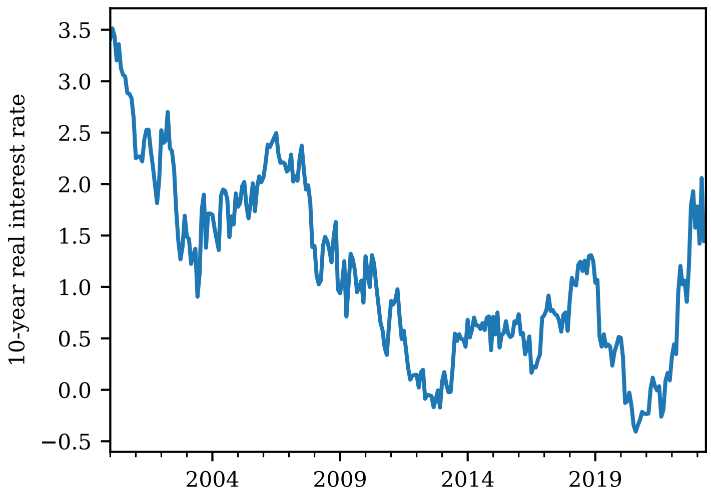
The data is returned as a pandas Series object with the corresponding dates set as the index.

[23]:
```python
series.info()
```

```
<class 'pandas.core.series.Series'>
DatetimeIndex: 281 entries, 2000-01-01 to 2023-05-01
Series name: None
Non-Null Count  Dtype
--------------  -----
281 non-null    float64
dtypes: float64(1)
memory usage: 4.4 KB
```

[24]:
```python
# Plot interest rate time series
series.plot(ylabel='10-year real interest rate')
```

[24]:
```
<Axes: ylabel='10-year real interest rate'>
```



Other popular time series available on FRED are the CPI, real GDP and the unemployment rate.

7.3.4 NASDAQ data API (optional)

The NASDAQ stock exchange provides an open-source Python library hosted on GitHub to access various types of financial data (not only those traded on NASDAQ), see here for details. The detailed API documentation can be found at here. This data API was formerly known as quandl which is no longer actively maintained but might still work.

Before using this service, you need to make sure that the Python package is installed. Depending on how you launched this notebook, you may need to execute the following code to install `nasdaq-data-link`:

```
pip install nasdaq-data-link
```

Various types of data are available via this service and can be found using the online search at `https://data.nasdaq.com/search`.

- Data come from various data provides. To select a data set, you usually have to specify a string of the form `'PROVIDER/SERIES'` where `'PROVIDER'` is the name of the provider (e.g., `'FRED'` or `'BOE'`) and `'SERIES'` is the name of the time series.
- Most of these data require a subscription or at least a free NASDAQ account. Once you have an account, you will need to get an API key and specify it when retrieving data. See the above links for details.
- Some commercial data series include sample data that can be used without a subscription but requires a free NASDAQ account.
- Some data series are freely available without a subscription or an account. These are often taken from other freely available data sets such as FRED or blockchain.com. We'll be using these to demonstrate how the API works.

  Important: Even for freely available data, NASDAQ imposes a cap of 50 web requests per day. You need to register to get around this.

The data is returned as pandas `DataFrame` object (or alternatively as an NumPy array).

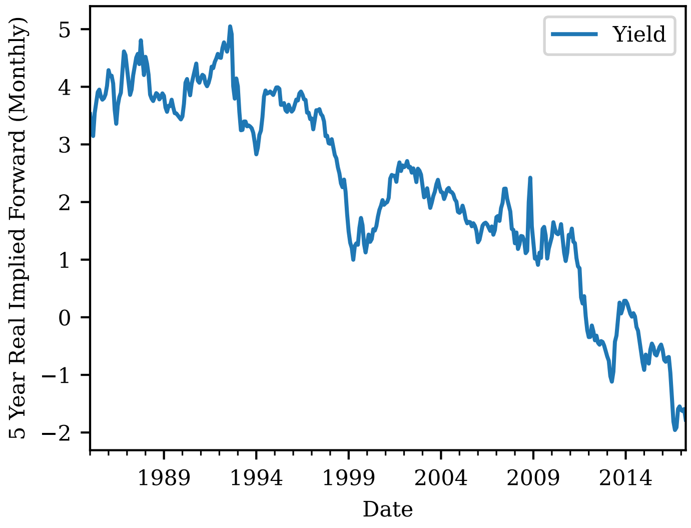
Example: Data from the Bank of England

Let's start by retrieving some macroeconomic times series from the Bank of England (BOE). It's not always straightforward to find the name of the time series one is looking for, but you can see some of the available time series here. The name will vary depending on the type of data (interest rate, exchange rate), the frequency and how it is aggregated (daily, last day of the month, monthly average) and a currency pair, if applicable.

[25]:
```
# When running via Google Colab, uncomment and execute the following line
#! pip install nasdaq-data-link
```

[26]:
```
# Retrieve 5-year real implied yield on UK government bonds
import nasdaqdatalink as ndl
df = ndl.get('BOE/IUMASRIF')

# Rename column which is always called 'Value'
df = df.rename(columns={'Value': 'Yield'})

# Plot time series
df.plot(ylabel='5 Year Real Implied Forward (Monthly)')
```

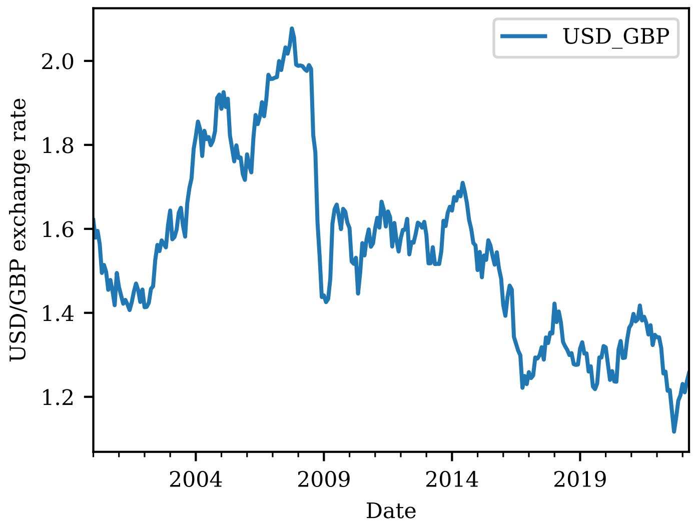
[26]:
```
<Axes: xlabel='Date', ylabel='5 Year Real Implied Forward (Monthly)'>
```



As another example, we retrieve the US dollar / Sterling exchange rate at a monthly frequency (this is determined by the name of the time series used where `ML` requests the monthly series, using the last observation for each month). Note that we can pass additional arguments, for example restricting the time period we want to retrieve using `start_date` and `end_date`.

[27]:
```
# Get USD / GDP exchange rate using the last observation for each month.
df = ndl.get('BOE/XUMLUSS', start_date='2000-01-31')
df = df.rename(columns={'Value': 'USD_GBP'})

# Plot USD/GBP time series
df.plot(ylabel='USD/GBP exchange rate')
```

[27]:
```
<Axes: xlabel='Date', ylabel='USD/GBP exchange rate'>
```

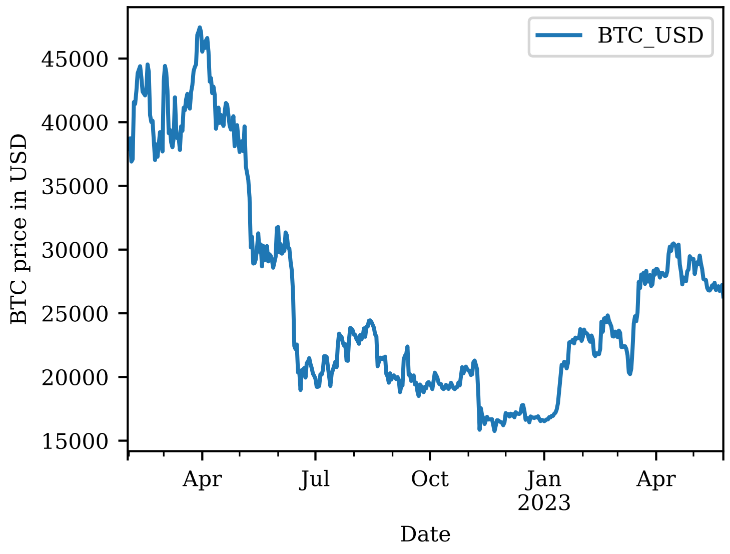


Example: Data from blockchain.com

The NASDAQ data link also supports retrieving data on cryptocurrencies. For example, there is a freely accessible time series for the price of Bitcoin in USD.

```
[28]: import nasdaqdatalink as ndl

      # Retrieve price of BTC in USD for 2022
      df = ndl.get('BCHAIN/MKPRU', start_date='2022-01-31')

      # Change column name to something more descriptive
      df = df.rename(columns={'Value': 'BTC_USD'})

      # Plot time series
      df.plot(ylabel='BTC price in USD')
```

```
[28]: <Axes: xlabel='Date', ylabel='BTC price in USD'>
```



Example: Historical stock data

As a final example, we obtain the trading data for the stock of Apple (ticker symbol AAPL) for the year 2001. Such data is often not available without a subscription or a login, but it works if the requested time period is sufficiently far in the past!

[29]:
```
# Retrieve stock data for Apple (ticker symbol AAPL)
df = ndl.get("WIKI/AAPL",
    start_date='2000-01-01',
    end_date='2000-12-31'
)
```

Unlike in the previous examples, this data contains not only a single value, but a whole range of variables including the opening and closing price, the trading volume, etc.:

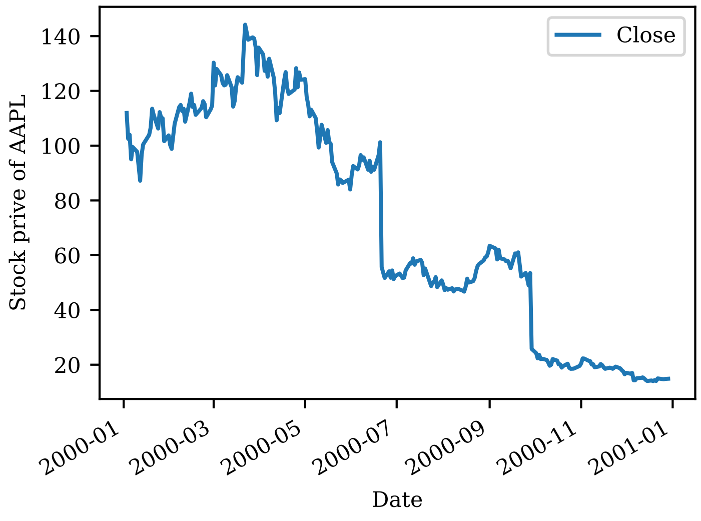
[30]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 252 entries, 2000-01-03 to 2000-12-29
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Open         252 non-null    float64
 1   High         252 non-null    float64
 2   Low          252 non-null    float64
 3   Close        252 non-null    float64
 4   Volume       252 non-null    float64
 5   Ex-Dividend  252 non-null    float64
 6   Split Ratio  252 non-null    float64
 7   Adj. Open    252 non-null    float64
 8   Adj. High    252 non-null    float64
 9   Adj. Low     252 non-null    float64
 10  Adj. Close   252 non-null    float64
 11  Adj. Volume  252 non-null    float64
dtypes: float64(12)
memory usage: 25.6 KB
```

To plot a specific column, we can use the `y=...` argument to `DataFrame.plot()`.

[31]:
```
df.plot(y='Close', ylabel='Stock prive of AAPL')
```

[31]:
```
<Axes: xlabel='Date', ylabel='Stock prive of AAPL'>
```



8 Random number generation and statistics

In this unit, we examine how to generate random numbers for various probability distributions in NumPy. Additionally, we take a look at SciPy's `stats` package which implements PDFs and other functions for numerous probability distributions.

8.1 Random number generators

Currently, there are several ways to draw random numbers in Python:

1. The *new* programming interface implemented in NumPy, introduced in version 1.17 (the current version as of this writing is 1.24) [official documentation].
2. The *legacy* programming interface implemented in NumPy [official documentation].

   While these functions have been superseded by the new implementation, they continue to work. If you are familiar with the legacy interface, you can read about what has changed in the new interface here.
3. The Python standard library itself also includes random number generators in the `random` module [official documentation].

   We won't be using this implementation at all, since for our purposes `numpy.random` is preferable as it supports NumPy arrays.

The programming interface for generating random numbers in NumPy changed substantially in release 1.17. We discuss the new interface in this unit since it offers several advantages, including faster algorithms for some distributions. Moreover, one would expect the legacy interface to be removed at some point in the future. However, most examples you will find in textbooks and on the internet are likely to use the old variant.

A note on random-number generation

Computers usually cannot draw truly random numbers, so we often talk about *pseudo-random number generators* (PRNG). Given an initial seed, these PRNGs will always produce the same sequence of "random" numbers, at least if run on the same machine, using the same underlying algorithm, etc. For scientific purposes this is actually desirable as it allows us to create reproducible results. For simplicity, we will nevertheless be using the terms "random number" and "random number generator" (RNG), omitting the "pseudo" prefix.

8.1.1 Simple random data generation

Before we can generate any random numbers using the new interface, we need to obtain an RNG instance. We can get the default RNG by calling `default_rng()` as follows:

[1]:
```python
# import function that returns the default RNG
from numpy.random import default_rng

# get an instance of the default RNG
rng = default_rng()
```

Let's begin with the most simple case, which uses the `random()` function to draw numbers that are uniformly distributed on the half-open interval $[0.0, 1.0)$.

```
[2]: from numpy.random import default_rng
     rng = default_rng()         # obtain default RNG implementation

     rng.random(5)               # return array of 5 random numbers
```

```
[2]: array([0.49063218, 0.40138645, 0.47361608, 0.92928662, 0.38783081])
```

Calling `random()` this way will return a different set of numbers each time (this might, for example, depend on the system time). To obtain the same draw each time, we can pass an initial *seed* when creating an instance of the RNG like this:

```
[3]: seed = 123
     rng = default_rng(seed)     # obtain default RNG implementation,
                                 # initialise seed

     rng.random(5)               # return array of 5 random numbers
```

```
[3]: array([0.68235186, 0.05382102, 0.22035987, 0.18437181, 0.1759059 ])
```

The `seed` argument needs to be an integer or an array of integers. This way, each call gives the same numbers as can easily be illustrated with a loop:

```
[4]: seed = 123
     for i in range(5):
         rng = default_rng(seed)
         print(rng.random(5))
```

```
[0.68235186 0.05382102 0.22035987 0.18437181 0.1759059 ]
[0.68235186 0.05382102 0.22035987 0.18437181 0.1759059 ]
[0.68235186 0.05382102 0.22035987 0.18437181 0.1759059 ]
[0.68235186 0.05382102 0.22035987 0.18437181 0.1759059 ]
[0.68235186 0.05382102 0.22035987 0.18437181 0.1759059 ]
```

You can remove the `seed` to verify that the set of number will then differ in each iteration.

Alternatively, we might want to draw random integers by calling `integers()`, which returns numbers from a “discrete uniform” distribution on a given interval:

```
[5]: rng.integers(2, size=5)         # vector of 5 integers from set {0, 1}
                                     # here we specify only the (non-inclusive)
                                     # upper bound 2
```

```
[5]: array([0, 1, 0, 1, 0])
```

Alternatively, we can specify the lower and upper bounds like this:

```
[6]: rng.integers(1, 10, size=5)     # specify lower and upper bound
```

```
[6]: array([3, 8, 8, 8, 9])
```

Following the usual convention in Python, the upper bound is not included by default. We can change this by additionally passing `endpoint=True`:

```
[7]: rng.integers(1, 10, size=5, endpoint=True)       # include upper bound
```

```
[7]: array([1, 6, 3, 3, 3])
```

We can create higher-order arrays by passing a list or tuple as the `size` argument:

```
[8]: rng.random(size=(2, 5))             # Create 2x5 array of floats
                                         # on [0.0, 1.0)
```

```
[8]: array([[0.21376296, 0.74146705, 0.6299402 , 0.92740726, 0.23190819],
            [0.79912513, 0.51816504, 0.23155562, 0.16590399, 0.49778897]])
```

```
[9]: rng.integers(2, size=(2,3,4))       # Create 2x3x4 array of integers {0,1}
```

```
[9]: array([[[1, 0, 1, 0],
             [0, 0, 0, 0],
             [0, 0, 1, 0]],

            [[1, 0, 1, 1],
             [1, 1, 1, 0],
             [0, 0, 1, 0]]])
```

Legacy interface

For completeness, let's look how you would accomplish the same using the *legacy* NumPy interface.

To draw floats on the unit interval, we use `random_sample()`:

```
[10]: from numpy.random import random_sample, randint, seed
      seed(123)
      random_sample(5)
```

```
[10]: array([0.69646919, 0.28613933, 0.22685145, 0.55131477, 0.71946897])
```

Random integers can be generated using `randint()`:

```
[11]: randint(2, size=5)      # draw random integers from {0,1}
```

```
[11]: array([1, 1, 0, 1, 0])
```

The legacy interface defines global functions `seed`, `random_sample`, etc. within the `numpy.random` module, which are implicitly associated with a global RNG object. This implicit association has been removed in the new programming model and you now have to obtain an RNG instance explicitly, for example by using the `default_rng()` function, as demonstrated above.

8.1.2 Drawing random numbers from distributions

Often we want to draw random numbers from a specific distribution such as the normal or log-normal distributions. The RNGs in `numpy.random` support a multitude of distributions, including:

- `binomial()`
- `exponential()`
- `normal()`
- `lognormal()`
- `multivariate_normal()`
- `uniform()`

and many others. For a complete list, see the official documentation.

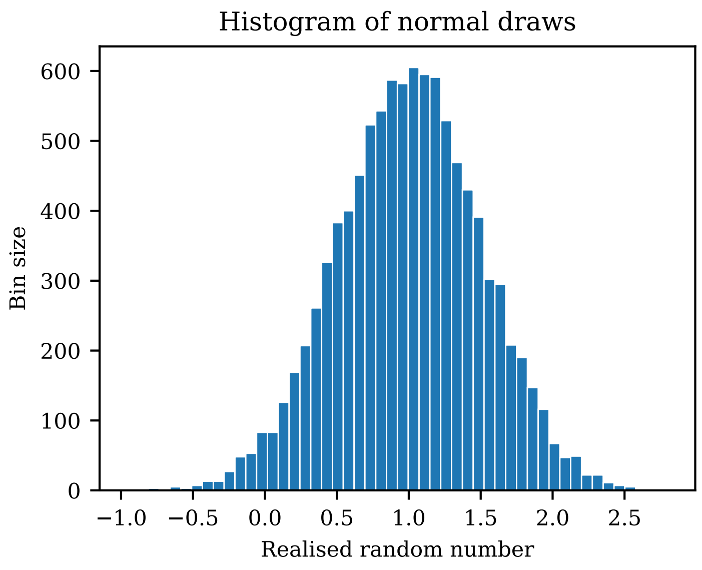
Example: Drawing from a normal distribution

We can draw from the normal distribution with mean $\mu = 1.0$ and standard deviation $\sigma = 0.5$ using `normal()` as follows:

```
[12]: from numpy.random import default_rng
      rng = default_rng(123)

      # location and scale parameters of normal distribution
```

```
mu = 1.0
sigma = 0.5

# Draw 10000 normal numbers;
# mean and std. are passed as loc and scale arguments
x = rng.normal(loc=mu, scale=sigma, size=10000)

# plot the results
import matplotlib.pyplot as plt
plt.hist(x, bins=50, linewidth=0.5, edgecolor='white')
plt.xlabel('Realised random number')
plt.ylabel('Bin size')
plt.title('Histogram of normal draws')
```

```
[12]: Text(0.5, 1.0, 'Histogram of normal draws')
```



Example: Drawing from a multivariate normal distribution

To draw from the multivariate normal, we need to specify a vector of means $\boldsymbol{\mu}$ and the variance-covariance matrix $\boldsymbol{\Sigma}$, which we set to

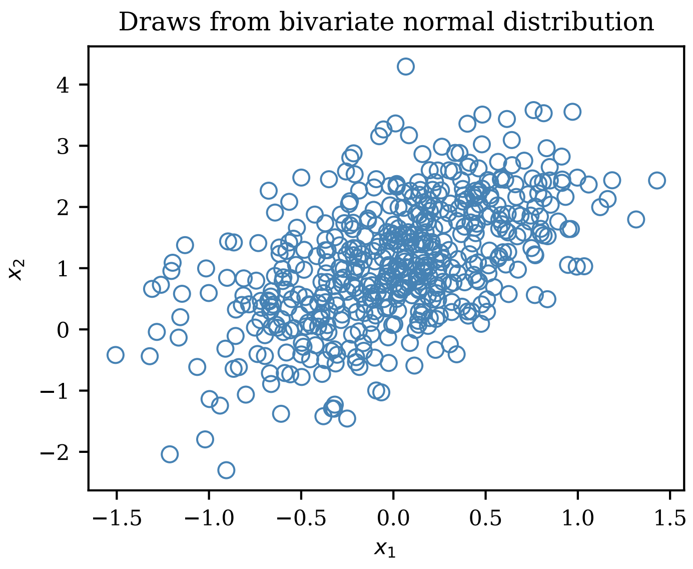
$$\boldsymbol{\mu} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \qquad \boldsymbol{\Sigma} = \begin{bmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{bmatrix}$$

with $\sigma_1 = 0.5$, $\sigma_2 = 1.0$ and $\rho = 0.5$. We call `multivariate_normal()` to draw a sample:

```
[13]: import numpy as np
      from numpy.random import default_rng
      import matplotlib.pyplot as plt

      rng = default_rng(123)

      mu = np.array((0.0, 1.0))        # vector of means
      sigma1 = 0.5                     # Std. dev. of first dimension
      sigma2 = 1.0                     # Std. dev. of second dimension
      rho = 0.5                        # Correlation coefficient

      # Compute covariance
      cov = rho * sigma1 * sigma2
```

```
# Create variance-covariance matrix
vcv = np.array([[sigma1**2.0, cov],
                [cov, sigma2**2.0]])

# Draw MVN random numbers:
# each row represents one sample draw.
x = rng.multivariate_normal(mean=mu, cov=vcv, size=500)

# Scatter plot of sample
plt.scatter(x[:, 0], x[:, 1], color='none', edgecolor='steelblue', lw=0.75)
plt.xlabel(r'$x_1$')
plt.ylabel(r'$x_2$')
plt.title('Draws from bivariate normal distribution')
```

```
[13]: Text(0.5, 1.0, 'Draws from bivariate normal distribution')
```



8.2 More functions for probability distributions

NumPy itself only implements distribution-specific RNGs. Frequently, we want to evaluate probability density functions (PDFs), cumulative distribution functions (CDFs) or compute some moments such as the mean of a random variable following some distribution. The SciPy project implements these functions for a wide range of discrete and continuous univariate distributions as well as for a few multivariate ones in the `scipy.stats` package.

The most useful functions include:

- `pdf()`: probability density function
- `cdf()`: cumulative distribution function
- `ppf()`: percent point function (inverse of `cdf`)
- `moment()`: non-central moment of some order *n*
- `expect()`: expected value of a function (of one argument) with respect to the distribution

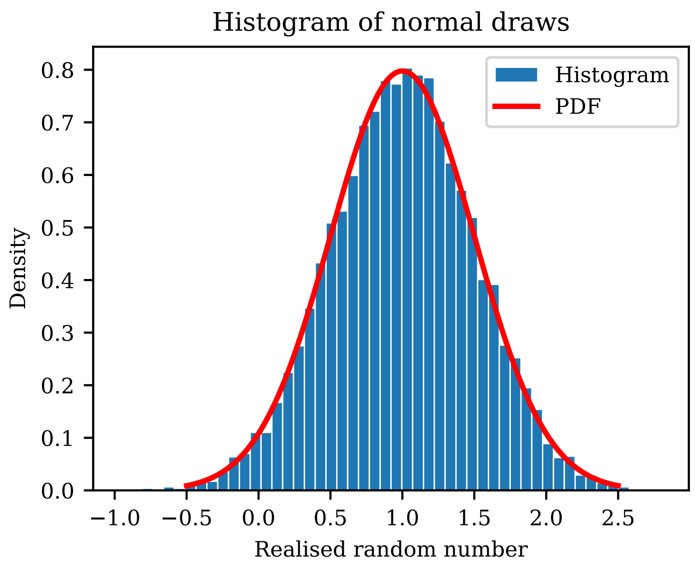
The parameters that need to be passed to these functions are distribution dependent. See the official documentation for details.

Example: Plotting a normal probability density function (PDF)

We can overlay the histogram of normal draws with the actual normal PDF using SciPy's `norm` distribution as follows:

[14]:
```
from numpy.random import default_rng
from scipy.stats import norm                # import normal distribution
import matplotlib.pyplot as plt
rng = default_rng(123)

# location and scale parameters of normal distribution
mu = 1.0
sigma = 0.5

# Draw 10000 normal numbers
x = rng.normal(loc=mu, scale=sigma, size=10000)     # mean and std. are passed as
                                                    # loc and scale arguments

# plot histogram
plt.hist(x, bins=50, density=True, linewidth=0.5, edgecolor='white',
         label='Histogram')

# Create x-values for PDF plot, using mean +/- 3 std.
xvalues = np.linspace(mu - 3*sigma, mu + 3*sigma, 100)
# Compute PDF of normal distr. at given x-values
pdf = norm.pdf(xvalues, loc=mu, scale=sigma)
# Plot PDF
plt.plot(xvalues, pdf, linewidth=2.0, color='red', label='PDF')
plt.xlabel('Realised random number')
plt.ylabel('Density')
plt.title('Histogram of normal draws')
plt.legend()
```

[14]:
```
<matplotlib.legend.Legend at 0x7f5f46b42b90>
```



In the above example we pass `density=True` to Matplotlib's `hist()` plotting function so that the result is rescaled to be comparable to the actual PDF.

Example: Computing the expectation of a function (advanced)

Sometimes we need to compute the expectation of a function $g(x)$ with respect to a given distribution with PDF $f(x)$ on some interval (a, b):

$$\mathbb{E}[g(x)] = \int_a^b g(x) f(x) dx$$

For example, we might want to know the mean of a *truncated* normal with parameters $\mu = 0, \sigma = 1.0$ with support on $(-\infty, 0)$, i.e.,

$$\mathbb{E}[x | x \leq 0] = \int_{-\infty}^{0} x \frac{f(x)}{F(0)} dx$$

where $f(x)$ and $F(x)$ are the PDF and CDF of the standard normal. We can compute it as follows:

```
[15]: from scipy.stats import norm
      import numpy as np

      lb = -np.inf            # integration lower bound
      ub = 0.0                # integration upper bound

      mu = 0.0                # mean of the (untruncated) normal
      sigma = 1.0             # std. dev. of the (untruncated) normal

      cdf0 = norm.cdf(0.0, loc=mu, scale=sigma)       # CDF at 0

      # Compute conditional expected value
      norm.expect(lambda x: x/cdf0, loc=mu, scale=sigma, lb=lb, ub=ub)
```

```
[15]: -0.7978845608028654
```

Here we define the function to be integrated as $g(x) = \frac{x}{F(0)}$, and we pass it to `expect()` as a lambda expression.

We can alternatively let `expect()` do the conditioning automatically by specifying `conditional=True`, and then we don't even need to apply the scaling factor $\frac{1}{F(0)}$:

```
[16]: norm.expect(lambda x: x, loc=mu, scale=sigma, lb=lb, ub=ub,
                  conditional=True)
```

```
[16]: -0.7978845608028654
```

8.3 Statistics functions

In the previous section, we examined functions associated with specific distributions. Additionally, there are numerous routines to process *sample* data which are spread across NumPy and SciPy.

In NumPy, the most useful routines include:

- `np.mean()`, `np.average()`: sample mean; the latter variant can also compute weighted means.
- `np.std()`, `np.var()`: sample standard deviation and variance
- `np.percentile()`, `np.quantile()`: percentiles or quantiles of a given array
- `np.corrcoef()`: Pearson correlation coefficient
- `np.cov()`: sample variance-covariance matrix
- `np.histogram()`: histogram of data. This only bins the data, as opposed to Matplotlib's `hist()` which plots it.

In addition, there are the variants `np.nanmean()`, `np.nanstd()`, `np.nanvar()` `np.nanpercentile()` and `np.nanquantile()` which ignore `NaN` values. You can find the full list of routines in the official documentation.

On top of that, the `scipy.stats` package contains functions to compute all sorts of descriptive statistics and statistical hypothesis tests. Many of these routines are too specific to be listed here, so have a look at the official documentation if you need to perform statistical analysis of your sample data.

Example: Pairwise correlations

To compute the pairwise correlations of a sample drawn from a multivariate normal distribution we proceed as follows:

```
[17]: import numpy as np
      from numpy.random import default_rng

      rng = default_rng(123)

      # vector of multivariate normal means
      mu = np.array([-1.0, 1.0])

      sigma1 = 0.5                    # Std. dev. of first dimension
      sigma2 = 1.0                    # Std. dev. of second dimension
      rho = 0.5                       # Correlation coefficient

      # Compute covariance
      cov = rho * sigma1 * sigma2

      # variance-covariance matrix
      vcv = np.array([[sigma1**2.0, cov],
                      [cov, sigma2**2.0]])

      # Draw some multivariate-normal random numbers
      x = rng.multivariate_normal(mean=mu, cov=vcv, size=1000)

      # Compute correlation coefficient (function expects each row to
      # contain one variable)
      np.corrcoef(x.T)
```

```
[17]: array([[1.        , 0.51768322],
             [0.51768322, 1.        ]])
```

Depending on the sample size, the correlation coefficient reported in the off-diagonal elements might or might not be close to the ρ used to draw the random data.

Example: Descriptive statistics

In the next example, we demonstrate how to compute some descriptive statistics using SciPy's `describe()` for a sample drawn from a 3-dimensional multivariate normal distribution:

```
[18]: import scipy.stats
      import numpy as np
      from numpy.random import default_rng

      rng = default_rng(123)

      # vector of multivariate normal means
      mu = np.array([-1.0, 0.0, 1.0])

      # variance-covariance matrix
      vcv = np.array([[1.0, 0.5, 0.2],
                      [0.5, 2.0, 0.7],
                      [0.2, 0.7, 0.5]])

      # Draw some multivariate-normal random numbers
      x = rng.multivariate_normal(mean=mu, cov=vcv, size=100)

      # Compute some descriptive statistics
      nobs, minmax, mean, variance, skewness, kurtosis = scipy.stats.describe(x)
```

```
# print array of means
mean
```

[18]:
```
array([-0.98486214, -0.0719401 ,  0.99084898])
```

[19]:
```
# array of variances
variance
```

[19]:
```
array([0.80017787, 1.96834418, 0.37118602])
```

Example: Test for normality (advanced)

To illustrate how to use one of the many tests implemented in `scipy.stats`, we compute the Jarque-Bera test statistic using `jarque_bera()`. This is a goodness-of-fit test to assess whether a sample has zero skewness and excess kurtosis and could thus be normally distributed.

[20]:
```
from scipy.stats import jarque_bera
from numpy.random import default_rng

rng = default_rng(123)

# Draw from univariate normal
x = rng.normal(loc=1.0, scale=2.0, size=10000)

# Compute Jarque-Bera test statistic
jb_stat, pvalue = jarque_bera(x)
print(f'Test statistic: {jb_stat:.3f}, p-value: {pvalue:.3f}')
```

```
Test statistic: 3.472, p-value: 0.176
```

With a p-value of about 0.18 we cannot reject the null hypothesis of zero skewness and zero excess kurtosis.

8.4 Optional exercises

The following exercises are considerably longer than those in previous units. The reason is that they incorporate everything we have covered so far, and we are finally able to use larger data sets (albeit only randomly generated ones) instead of just calling `np.arange(5)` all the time. In this sense, the exercises are starting to resemble (simplified) real-world applications.

Exercise 1: Histograms for increasing sample sizes

In this exercise, we plot histograms against the actual PDF of a standard-t distributed random variable for increasing sample sizes.

Consider the standard-t distribution with 20 degrees of freedom (this is the only parameter of this distribution):

- To draw samples from this distribution, use NumPy RNG's `standard_t()` method.
- To plot the PDF of this distribution, use the `t` distribution from `scipy.stats`. You can import it as follows

  ```
  from scipy.stats import t as standard_t
  ```

  It is a good idea to assign more descriptive names to imported symbols than a `t`.

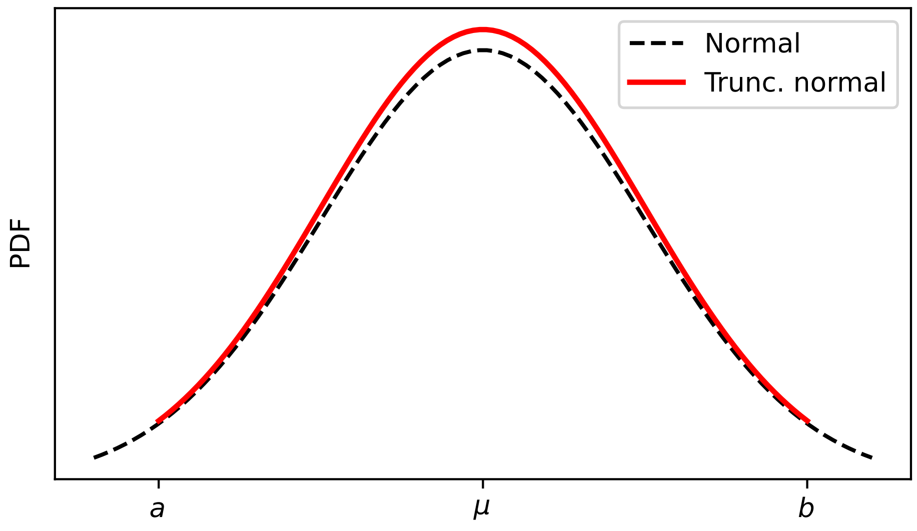
Perform the following tasks:

1. Draw random samples from the standard-t distribution for a sequence of increasing sample sizes of 50, 100, 500, 1000, 5000 and 10000.
2. Create a single figure with 6 panels in which you plot a histogram of the samples you have drawn. Use matplotlib's `hist()` function to do this, and pass the argument `bins = 50` so that each panel uses the same number of bins.
3. Add the actual PDF of the standard-t distribution to each panel. To evaluate the PDF, use the `pdf()` method of the `t` distribution you imported from `scipy.stats`.

Exercise 2: Moments of truncated normal

In this exercise, you are asked to compute the second non-central moment of a truncated normal distribution.

Consider a truncated normal distribution with support on the interval $[a, b]$ with $a = \mu - 2\sigma$ and $b = \mu + 2\sigma$. Assume the underlying (untruncated) normal distribution has mean $\mu = 0$ and variance $\sigma^2 = 1$. Compared to the untruncated normal PDF, the truncated PDF is rescaled upwards so that it integrates to 1, as illustrated in the following figure:



Compute the second non-central moment $\mathbb{E}[X^2]$ in four different ways:

1. Use the fact that $Var(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2$. Call the methods `mean()` and `var()` of the truncated normal implemented in SciPy to compute the (squared) mean and variance.

   Hint: Import the truncated normal as follows:

   ```
   from scipy.stats import truncnorm
   ```

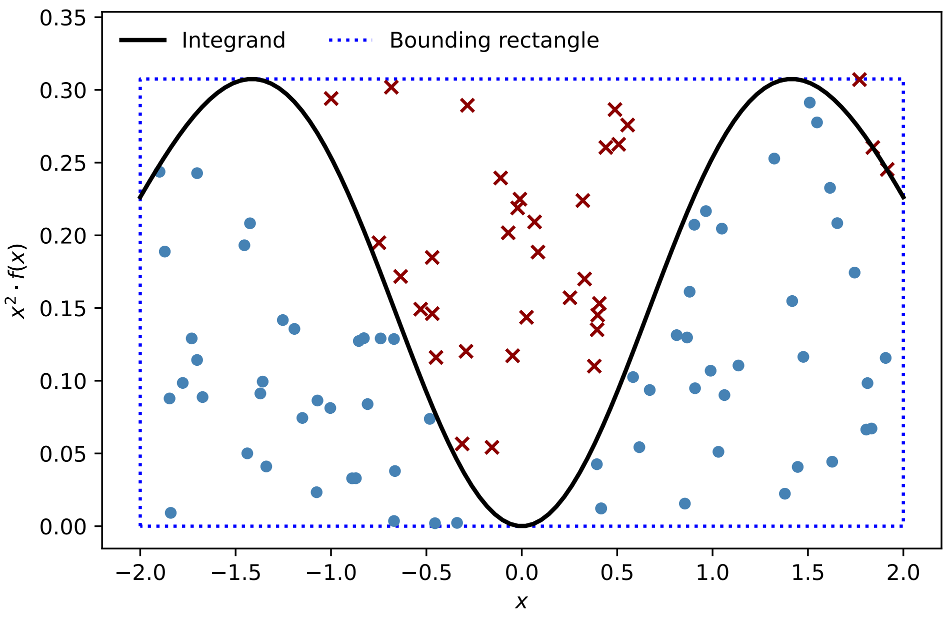
2. Use the `moment()` method of SciPy's truncated normal to directly compute the desired moment.

3. Use the `expect()` method of SciPy's truncated normal to compute the expectation $\mathbb{E}[X^2]$.

4. Use Monte Carlo integration to compute the expectation $\mathbb{E}[X^2]$. There are numerous ways to do MC integration. In this exercise, we use a variant which draws random samples from a 2-dimensional uniform distribution to compute an area under the integrand:

   - To do this, define the integrand as $x^2 \cdot f_t(x; a, b, \mu, \sigma)$ where f_t is the PDF of the truncated normal with parameters a, b, μ and σ.
   - Draw random numbers in the rectangle which has length $b - a$ and a height which is the maximum of the integrand on the integration interval $[a, b]$.
   - Determine the fraction of sampled points that are below the integral, and use this to compute the area under the integrand.

The following figure illustrates this approach to integration. The blue dots are included in the integral whereas the red crosses are not:



This may not be the most practical way to do MC integration, and we will examine a more common approach in the next exercise.

Note: SciPy's truncated normal expects *normalised* boundaries a and b. Whenever the underlying distribution is *not* standard normal, you have to pass $z_a = (a - \mu)/\sigma$ and $z_b = (b - \mu)/\sigma$ instead of a and b to all of `truncnorm`'s methods.

Exercise 3: Multi-period asset returns

Consider an investor with initial assets a and a 2-period investment horizon (we assume the investor does not change the asset position after the first period).

Denote by R the total gross return over two periods, so that the terminal wealth is given by $W = a \cdot R$. The total gross return is the product of the period-1 and period-2 gross returns, $R = R_1 \cdot R_2$. We impose that per-period log returns $r_t = \log R_t$ are jointly normally distributed with mean $\mathbb{E}[r_t] = \mu$, variance $Var(r_t) = \sigma^2$ for $t \in \{1, 2\}$ with a correlation coefficient $Corr(r_1, r_2) = \rho$. Let $a = 1$, $\mu = 0.04$, $\sigma = 0.16$ and $\rho = 0.5$.

1. Derive the analytical expression for the expected total gross return after 2 periods.

   Hint:

   - Remember that since (r_1, r_2) are jointly normally distributed, so is their sum, $\log R = r_1 + r_2$.
   - Moreover, if $\log R$ is normally distributed with mean μ_R and variance σ_R^2, then R is log-normally distributed and has the expected value

   $$\mathbb{E}[R] = \exp\left(\mu_R + \frac{1}{2}\sigma_R^2\right)$$

2. Compute the expected terminal wealth after 2 periods using Monte Carlo simulation. To do this,

   1. Draw N samples of multivariate normally distributed vectors (r_{1i}, r_{2i}) using NumPy's multi-variate_normal().

2. Compute the terminal wealth for each draw i: $W_i = a \exp(r_{1i}) \exp(r_{2i})$.
3. Compute the expected wealth as the sample average:

$$\mathbb{E}[W] \approx \overline{W} = \frac{1}{N} \sum_{i=1}^{N} W_i$$

Make sure you get approximately the same result as in part 1 (you may need to increase the sample size if this is not the case).

3. Plot a histogram of the simulated total gross returns, and overlay it with the PDF of the log-normal distribution you derived in the first part.

Exercise 4: Standard error and increasing sample size

Consider a setting in which we draw multiple samples indexed by k such that these samples are increasing in the sample size N_k, given by N_k = 10, 50, 100, 500, 1000, 5000, 10000, 50000 and 100000.

The data for the k-th sample are $(x_{ik})_{i=1}^{N_k}$ where i indexes some draw within the k-th sample. Assume that the data are log-normally distributed such that the underlying normal distribution has mean $\mu = 0.5$ and variance $\sigma^2 = 1.5^2$.

1. For each sample size N_k, use NumPy's `lognormal()` to draw a log-normally distributed sample.
2. For each sample, compute the sample mean and its standard error. As a reminder, the standard error of the k-th sample mean $\overline{x}_k$ is defined as

$$se(\overline{x}_k) = \sqrt{\frac{\widehat{\sigma}_k^2}{N_k}}$$

where $\widehat{\sigma}_k^2$ is the variance of residuals $u_{ik} = x_{ik} - \overline{x}_k$ for each sample k,

$$\widehat{\sigma}_k^2 = \frac{1}{N_k - 1} \sum_{i=1}^{N_k} u_{ik}^2$$

3. Plot the sample means for all samples, using the sample size on the x-axis and the estimated mean on the y-axis. Use the $\log_{10}$ scale on the x-axis.

   Hint: You can activate log scaling by calling `xscale('log')`, or `set_xscale('log')` when using the object-oriented plotting interface.
4. Add a horizontal line showing the true mean (which is the same for all sample sizes).
5. Add confidence intervals (CI) for each sample size: use the interval $\overline{x}_k \pm 2 \times se(\overline{x}_k)$ to get a CI of approximately 95%.

Exercise 5: The jackknife

We continue with the setting from exercise 4, but instead of computing the standard error as above, we now use a resampling technique known as the jackknife to get the sample mean and its standard error.

1. For each sample k, compute N_k sample means $\overline{x}_{-i,k}$ which are obtained by leaving out the i-th observation:

$$\overline{x}_{-i,k} = \frac{1}{N_k - 1} \sum_{j=1, j \neq i}^{N_k} x_{jk} \qquad i = 1, \dots, N_k$$

where x_{jk} is the j-th draw in the k-th sample.

2. Compute the jackknife estimate of the sample mean as the average of these sub-sample means:

$$\overline{x}_{k,jack} = \frac{1}{N_k} \sum_{i=1}^{N_k} \overline{x}_{-i,k}$$

For the special case of a sample mean, it is straightforward to show that this is just the regular sample mean computed on the whole sample, $\overline{x}_{k,jack} = \overline{x}_k$, where

$$\overline{x}_k = \frac{1}{N_k} \sum_{i=1}^{N_k} x_{ik}$$

3. The jackknife estimate of the error variance for sample size N_k is then given by

$$\begin{aligned}\widehat{var}(\overline{x}_k) &= \frac{N_k - 1}{N_k} \sum_{i=1}^{N_k} \left(\overline{x}_{-i,k} - \overline{x}_{jack,k}\right)^2 \\ &= \frac{1}{N_k(N_k-1)} \sum_{i=1}^{N_k} (x_{ik} - \overline{x}_k)^2\end{aligned}$$

where the second line again follows as a special case if we are estimating the sample mean. The standard error of the sample mean is thus

$$se(\overline{x}_k) = \sqrt{\widehat{var}(\overline{x}_k)}$$

4. Recreate the plot from exercise 4, but now use the jackknife estimate of the standard error instead.

Exercise 6: The bootstrap

We continue with the setting from exercises 4 and 5, but now we use the bootstrap to estimate the confidence intervals of the mean estimate.

1. For each sample size N_k proceed as follows:
    1. Draw an initial sample of size N_k as before and compute the sample mean.
    2. Resample N_k observations by drawing from this initial sample *with replacement* using NumPy's `choice()` function.
    3. For each "resample", compute the sample mean. Say we have the j-th resample which consists of the draws $(x_{ik}^j)_{i=1}^{N_k}$, so we compute the j-th mean

    $$\overline{x}_k^j = \frac{1}{N_k} \sum_{i=1}^{N_k} x_{ik}^j$$

    4. Repeat steps 2 and 3 $N_{bs} = 999$ times.
    5. Use these N_{bs} means to find the 2.5% and 97.5% percentiles of the mean estimate distribution, $\overline{x}_k^{p2.5}$ and $\overline{x}_k^{p97.5}$.
    6. The bootstrapped 95% confidence interval is then given by $\left[\overline{x}_k^{p2.5}, \overline{x}_k^{p97.5}\right]$.
2. Recreate the same plot as in exercises 4 and 5, but this time use the bootstrapped 95% confidence interval you computed for each sample size.
3. For each sample size, store all bootstrapped means and use these to create a histogram of sample means. You will thus have to create 9 histograms. Use vertical lines to indicate the 95% confidence interval.

8.5 Solutions

The solutions are also provided as Python scripts in the lectures/solutions/unit08/ folder.

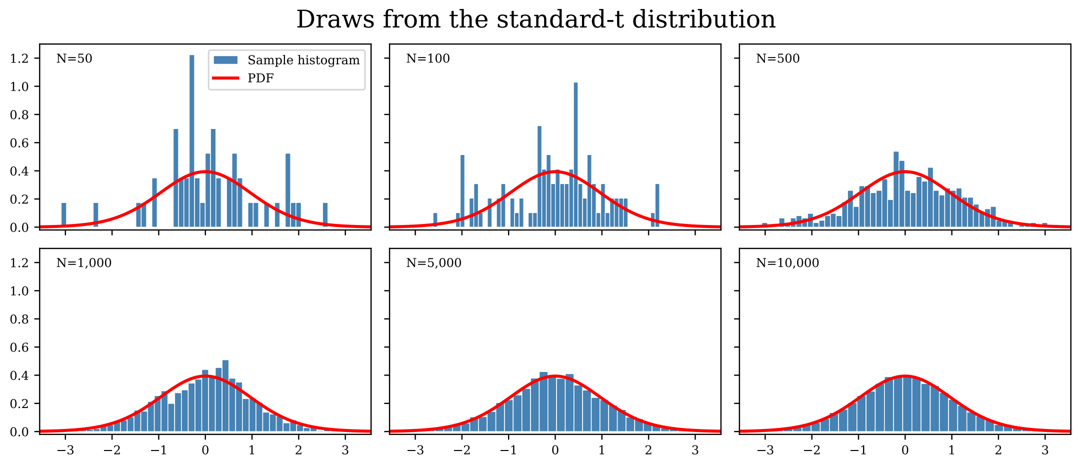
Solution for exercise 1

In the following solution, we create a figure with six panels (axes) and iterate over these axes. In each iteration, we

1. draw a random sample for the given (increasing) size;
2. plot the histogram using the current axes object; and
3. overlay the actual PDF.

[21]:
```
import matplotlib.pyplot as plt
import numpy as np
from numpy.random import default_rng
from scipy.stats import t as standard_t

# Sample sizes
Nobs = np.array((50, 100, 500, 1000, 5000, 10000))

# degrees of freedom
df = 20

# Determine xlims such that we cover the (0.1, 99.9) percentiles
# of the distribution.
xmin, xmax = standard_t.ppf((0.001, 0.999), df=df)

xvalues = np.linspace(xmin, xmax, 100)
pdf = standard_t.pdf(xvalues, df=df)

fig, ax = plt.subplots(2, 3, sharex=True, sharey=True, figsize=(10, 4))

# initialize default RNG
rng = default_rng(123)

for i, axes in enumerate(ax.flatten()):
    # Sample size to be plotted in current panel
    N = Nobs[i]
    # Draw sample of size N
    data = rng.standard_t(df=df, size=N)

    # plot histogram of given sample
    axes.hist(data, bins=50, linewidth=0.5, edgecolor='white',
              color='steelblue', density=True, label='Sample histogram')

    # overlay actual PDF
    axes.plot(xvalues, pdf, color='red', lw=2.0, label='PDF')

    # create text with current sample size
    axes.text(0.05, 0.95, f'N={N:,d}', transform=axes.transAxes, va='top')

    axes.set_xlim((xmin, xmax))
    axes.set_ylim((-0.02, 1.3))

    # plot legend only for the first panel
    if i == 0:
        axes.legend(loc='upper right')

# compress space between individual panels
fig.tight_layout()
# Add overall title
fig.suptitle('Draws from the standard-t distribution', fontsize=16, y=1.05)
```

[21]:
```
Text(0.5, 1.05, 'Draws from the standard-t distribution')
```



A few comments on how we create the x-values and the plot range for these graphs:

1. In principle, we could draw arbitrarily large or small realised values, but we want to restrict the plot limits to a reasonable interval.
2. To find such an interval, we compute the percentiles corresponding to 0.1% and 99.9%, which will cover almost any point we'd want to plot.
3. Moreover, we need to compute the x-values and evaluate the PDF at these points only once since these will remain unchanged for all sample sizes.

Note also that the `subplots()` function returns a 2-dimensional array (since we requested a 2×3 layout). We iterate over the *flattened* array of axes objects instead of writing two nested loops over rows and columns.

Solution for exercise 2

Computing the second non-central moment using the first three methods is straightforward. All you need to do is to make sure that you pass the correct parameters to SciPy's `truncnorm` methods:

[22]:
```python
import numpy as np
from scipy.stats import truncnorm

# Parameters
mu = 0.0
sigma = 1.0
a = mu - 2*sigma
b = mu + 2*sigma

# Standardised boundaries if underlying non-truncated distr. is
# NOT standard normal
za = (a-mu)/sigma
zb = (b-mu)/sigma

# Method 1: Compute from \mathbb{E}[X^2] = Var(X) + \mathbb{E}[X]^2
var = truncnorm.var(za, zb, loc=mu, scale=sigma)
mean = truncnorm.mean(za, zb, loc=mu, scale=sigma)
m2_var_mean = var + mean ** 2.0

# Method 2: Compute using moment()
m2 = truncnorm.moment(2, za, zb, loc=mu, scale=sigma)

# Method 3: Compute moment using expect() function
m2_expect = truncnorm.expect(lambda x: x**2.0, args=(za, zb),
                             loc=mu, scale=sigma)

print(f'Second non-central moment, var + mean^2: {m2_var_mean:.5e}')
print(f'Second non-central moment, moment(): {m2:.5e}')
print(f'Second non-central moment, expect(): {m2_expect:.5e}')
```

```
Second non-central moment, var + mean^2: 7.73741e-01
Second non-central moment, moment(): 7.73741e-01
Second non-central moment, expect(): 7.73741e-01
```

The fourth method is more involved. We first define the integrand as follows:

[23]:
```
# Function to compute integrand
def f_integrand(x, a, b, mu, sigma):
    # Transform to boundaries required by SciPy's truncnorm
    za = (a - mu) / sigma
    zb = (b - mu) / sigma
    # Evaluate truncated normal PDF
    pdf = truncnorm.pdf(x, za, zb, loc=mu, scale=sigma)
    # Compute integrand x^2 * f(x)
    result = x ** 2.0 * pdf
    return result
```

The remainder of the Monte Carlo code look as follows:

[24]:
```
from numpy.random import default_rng

# Initialise RNG
rng = default_rng(123)
# Sample size for Monte Carlo integration
Nsample = 50000

# x-values should be uniformly distributed on [a, b]
xsample = rng.uniform(a, b, size=Nsample)
# Alternatively we can also just use equidistant x-values, in
# low-dimensional problems it makes no difference.
# xsample = np.linspace(a, b, Nsamples)

# Evaluate integrand at sampled x-values
integrand = f_integrand(xsample, a, b, mu, sigma)

# Compute size of bounding rectangle:
# the height is taken as the largest realisation of the integrand.
length = b - a
height = np.amax(integrand)
area = height * length
# draw y-values from uniform distribution on [0, height]
ysample = rng.uniform(0, height, size=Nsample)
# Compute fraction of points that are underneath the PDF
frac = np.mean(ysample <= integrand)
integral_MC = frac * area

print(f'Second non-central moment, MC integration: {integral_MC:.5e}')
```

```
Second non-central moment, MC integration: 7.72828e-01
```

You might have noticed that MC integration is not the fastest to converge, but using 50000 draws is sufficient to get somewhat close to the other three methods.

In this case we do not actually need Monte Carlo methods, because the dimensionality of the problem is so low. We could just as well have used a dense deterministic rectangular grid instead of randomly-drawn points.

The entire Python script which also generates the graphs displayed in the exercise can be found in the solutions folder

Solution for exercise 3

Part 1

The first part is purely analytical. We use it to verify that the code in part 2 yields the correct result.

Since (r_1, r_2) are jointly normal, a standard result is that their sum $r_1 + r_2$ is normally distributed with mean and variance given by

$$\mu_{rr} = \mathbb{E}[r_1 + r_2] = 2\mu$$

$$\sigma_{rr}^2 = Var(r_1 + r_2) = Var(r_1) + Var(r_2) + 2 \cdot Cov(r_1, r_2) = 2\sigma^2 + 2\rho\sigma^2$$

This is even simpler than the usual formulas since both per-period log returns have the same mean and variance.

Moreover, since $\log R = r_1 + r_2$ is normally distributed, R is log-normally distributed and has the expected value

$$\mathbb{E}[R] = \exp\left(\mu_{rr} + \frac{1}{2}\sigma_{rr}^2\right) = \exp\left(2\mu + (1+\rho)\sigma^2\right)$$

Since $a = 1$, this is also the expected value of terminal wealth W.

We can plug in the parameter values to compute the numerical value:

[25]:
```
import numpy as np

# Parameters
a = 1.0                         # Initial assets
mu = np.array((0.04, 0.04))     # average log returns
sigma = 0.16                    # std. dev. of log returns
rho = 0.5                       # serial correlation

# Exact expectation
var_rr = 2.0 * sigma ** 2.0 + 2.0 * rho * sigma ** 2.0
sigma_rr = np.sqrt(var_rr)
mu_rr = np.sum(mu)

exp_exact = a * np.exp(mu_rr + sigma_rr ** 2.0 / 2.0)

print(f'Expected portfolio value (exact): {exp_exact:.4f}')
```

```
Expected portfolio value (exact): 1.1257
```

Parts 2 and 3

To perform the Monte Carlo simulation, we need to define the vector of means and the variance-covariance matrix which we can pass to NumPy's `multivariate_normal()` to sample returns (r_1, r_2):

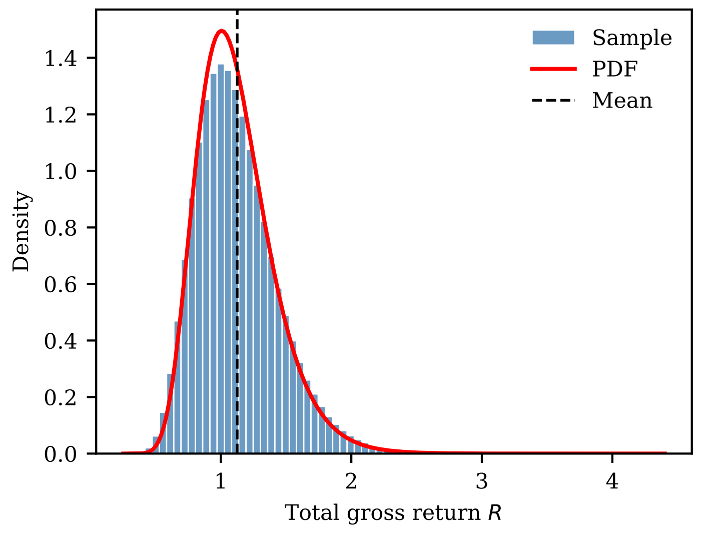
[26]:
```
import numpy as np
from numpy.random import default_rng
from scipy.stats import lognorm

# Parameters
a = 1.0                         # Initial assets
mu = np.array((0.04, 0.04))     # average log returns
sigma = 0.16                    # std. dev. of log returns
rho = 0.5                       # serial correlation

# Covariance
cov = rho*sigma**2.0
# variance-covariance matrix
vcv = np.array([[sigma**2.0, cov],
                [cov, sigma**2.0]])

Nsample = 5000000
rng = default_rng(123)
# Draw MV normal samples: each row corresponds to one draw
samples = rng.multivariate_normal(mean=mu, cov=vcv, size=Nsample)

# Evaluate total gross return at sampled points:
#   R = exp(r_1) * exp(r_2)
returns = np.prod(np.exp(samples), axis=1)
# Sampled terminal wealth after 2 periods
wealth = a * returns
# Expected terminal wealth
exp_MC = np.mean(wealth)

print(f'Expected portfolio value (MC): {exp_MC:.4f}')
```

```
Expected portfolio value (MC): 1.1256
```

Finally, we use the sampled points and the `pdf()` method of SciPy's lognorm to plot the histogram and the true PDF.

[27]:
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 1)

ax.hist(returns, bins=75, density=True, color='steelblue', lw=0.5,
        edgecolor='white', alpha=.8, label='Sample')

# Plot log-normal PDF of total gross return
xmin, xmax = np.amin(returns), np.amax(returns)
xvalues = np.linspace(xmin, xmax, 200)
pdf = lognorm.pdf(xvalues, s=sigma_rr, loc=mu_rr)
ax.plot(xvalues, pdf, c='red', lw=1.5, label='PDF')

# Add line with true expected value
ax.axvline(exp_exact, lw=1.0, color='black', ls='--', label='Mean')

ax.set_xlabel('Total gross return $R$')
ax.set_ylabel('Density')
ax.legend(loc='upper right', frameon=False)
```

[27]:
```
<matplotlib.legend.Legend at 0x7f5f4621d120>
```



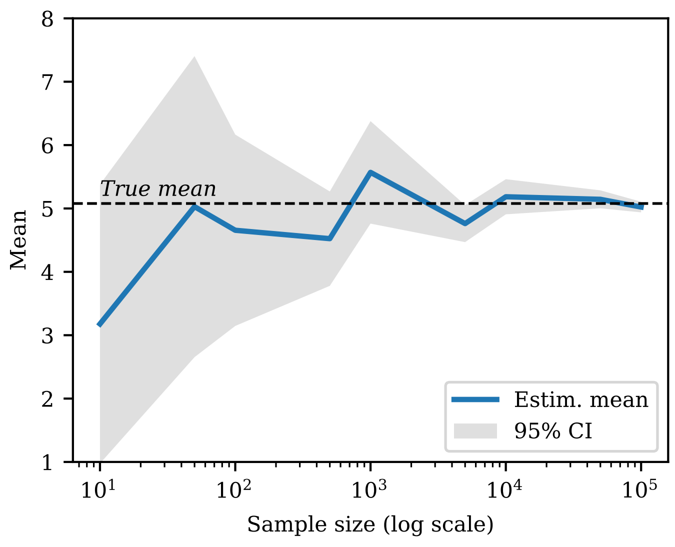
The dashed black line shows the analytically derived expected total gross return.

Solution for exercise 4

We solve the exercise by iterating over all sample sizes, drawing a new log-normal sample and computing the sample mean and standard error. These are stored in the arrays `mean_hat` and `std_err`, which we then use the generate the plot.

[28]:
```
import numpy as np
from numpy.random import default_rng
import matplotlib.pyplot as plt

sample_sizes = np.array([10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000])
# initialize default RNG
rng = default_rng(123)

# Parameters of underlying normal distribution
mu = 0.5
sigma = 1.5

# Array to store estimated mean for each sample size
mean_hat = np.zeros(len(sample_sizes))
# Array to store std. error for each sample size
std_err = np.zeros_like(mean_hat)

for k, N in enumerate(sample_sizes):
    data = rng.lognormal(mean=mu, sigma=sigma, size=N)
    # sample mean
    x_k = np.mean(data)
    # residuals around mean
    resid = data - x_k
    # Residual variance
    var_resid = np.sum(resid**2.0) / (N-1)
    # std. error of mean estimate
    se_k = np.sqrt(var_resid / N)

    # store sample estimates in array
    mean_hat[k] = x_k
    std_err[k] = se_k

# Plot results
plt.plot(sample_sizes, mean_hat, lw=2.0, label='Estim. mean')

# Add line indicating true mean of log-normal
mean = np.exp(mu + sigma**2.0 / 2.0)
plt.axhline(mean, lw=1.0, color='black', ls='--')
plt.text(sample_sizes[0], mean + 0.05, 'True mean', va='bottom',
         fontstyle='italic', fontfamily='serif')

plt.fill_between(sample_sizes, mean_hat - 2*std_err, mean_hat + 2*std_err,
                 color='grey', alpha=0.25, zorder=-1, lw=0.0,
                 label='95% CI')

plt.xscale('log')
plt.legend(loc='lower right')
plt.xlabel('Sample size (log scale)')
plt.ylabel('Mean')
# Use identical y-lims across ex. 4-6
plt.ylim((1.0, 8.0))
```

[28]:
```
(1.0, 8.0)
```



Solution for exercise 5

Much of this solution proceeds in the very same way as in exercise 4. Additionally,

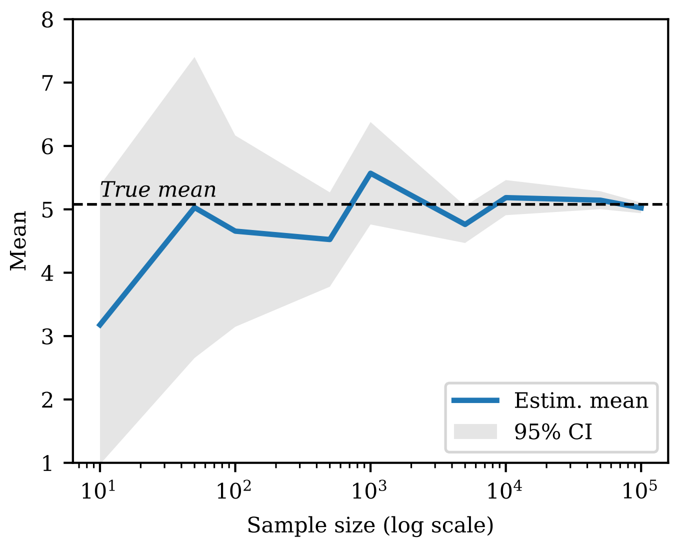
- For each sample, we now have to loop over all observations, create a sub-sample which omits a particular observation and calculate the mean of this sub-sample.
- We compute the estimate of the sample mean as the average of all these means.

The code is substantially slower than in exercise 4 as it takes considerable time to loop over all observations in the larger samples.

Note that the jackknife is rarely used these days as it has been replaced by other resampling methods such as the bootstrap. The resulting confidence intervals look identical to the ones in exercise 4 since we have established earlier that for the sample mean the jackknife estimate of the variance is in fact identical to the standard approach.

[29]:
```python
import numpy as np
from numpy.random import default_rng
import matplotlib.pyplot as plt


sample_sizes = np.array([10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000])
# initialize default RNG
rng = default_rng(123)

# Parameters of underlying normal distribution
mu = 0.5
sigma = 1.5

# Array to store estimated mean for each sample size
mean_hat = np.zeros(len(sample_sizes))
# Array to store std. errors for each sample size
std_err = np.zeros_like(mean_hat)

for k, N in enumerate(sample_sizes):
    data = rng.lognormal(mean=mu, sigma=sigma, size=N)

    mean_subsample = np.zeros_like(data)

    # Iterate over all elements, leaving out one element
    # and computing the mean of the resulting sub-sample
    for j in range(N):
        # Initial boolean mask: include all elements
        mask = np.ones_like(data, dtype=bool)
        # leave out j-th observation
        mask[j] = False
        subsample = data[mask]

        x_j = np.mean(subsample)
        mean_subsample[j] = x_j

    # compute sample jackknife mean estimate as average of
    # sub-sample means
    x_k = np.mean(mean_subsample)

    # Compute variance of jackknife mean estimate
    var = (N-1)/N * np.sum((mean_subsample - x_k) ** 2.0)
    # jackknife std. err. of mean estimate
    se = np.sqrt(var)

    # store sample estimates in array
    mean_hat[k] = x_k
    std_err[k] = se

# Plot results
plt.plot(sample_sizes, mean_hat, lw=2.0, label='Estim. mean')

# Add line indicating true mean of log-normal
mean = np.exp(mu + sigma ** 2.0 / 2.0)
plt.axhline(mean, lw=1.0, color='black', ls='--')
plt.text(sample_sizes[0], mean + 0.05, 'True mean', va='bottom',
         fontstyle='italic', fontfamily='serif')

plt.fill_between(sample_sizes, mean_hat - 2*std_err, mean_hat + 2*std_err,
                 color='grey', alpha=0.2, zorder=-1, lw=0.0,
                 label='95% CI')
plt.xscale('log')
plt.legend(loc='lower right')
plt.xlabel('Sample size (log scale)')
plt.ylabel('Mean')
# Use identical y-lims across ex. 4-6
plt.ylim((1.0, 8.0))
```

[29]:
```
(1.0, 8.0)
```



Solution for exercise 6

We first define a function `bootstrap_means()` which takes as given the initial sample, and

1. Resamples the desired number of times
2. For each resample, computes the sample mean
3. Returns all sample means in an array.

```
[30]: import numpy as np

      def boostrap_mean(x, Nrounds):
          means = np.zeros(Nrounds)
          # Sample size
          N = len(x)

          for j in range(Nrounds):
              # Resample with replacement
              resampled = np.random.choice(x, size=N, replace=True)

              # Compute mean of bootstrapped sample
              m = np.mean(resampled)

              means[j] = m

          return means
```

We use the function `np.random.choice()` to sample from the initial sample with replacement (passing the argument `replace=True`).

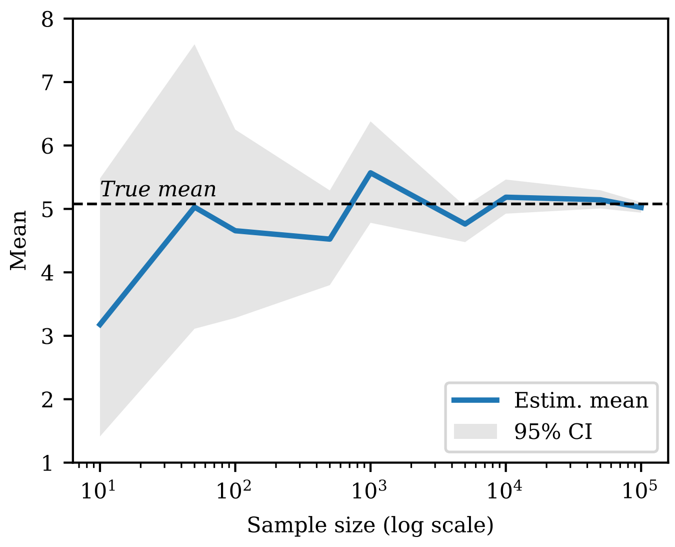
We can then use these bootstrapped means to compute the P2.5 and P97.5 percentiles using the `np.percentile()` function. These represent the bounds of the 95% confidence interval.

The remainder of the implementation looks almost identical to the previous exercises. We additionally store all bootstrapped means in the list `means_all` which we use below to create the histograms.

```
[31]: from numpy.random import default_rng
      import matplotlib.pyplot as plt

      sample_sizes = np.array([10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000])
      # Number of bootstraps
      Nbs = 999

      # initialize default RNG
      rng = default_rng(123)

      # Parameters of underlying normal distribution
      mu = 0.5
      sigma = 1.5

      # Array to store estimated mean for each sample size
      mean_hat = np.zeros(len(sample_sizes))
      # Array to store CI lower and upper bounds
      bounds = np.zeros((len(sample_sizes), 2))

      # Collect all bootstrapped means for each sample size
      # to create histograms at the end
      means_all = []

      for k, N in enumerate(sample_sizes):
          data = rng.lognormal(mean=mu, sigma=sigma, size=N)

          # Mean of original sample
          x_k = np.mean(data)
```

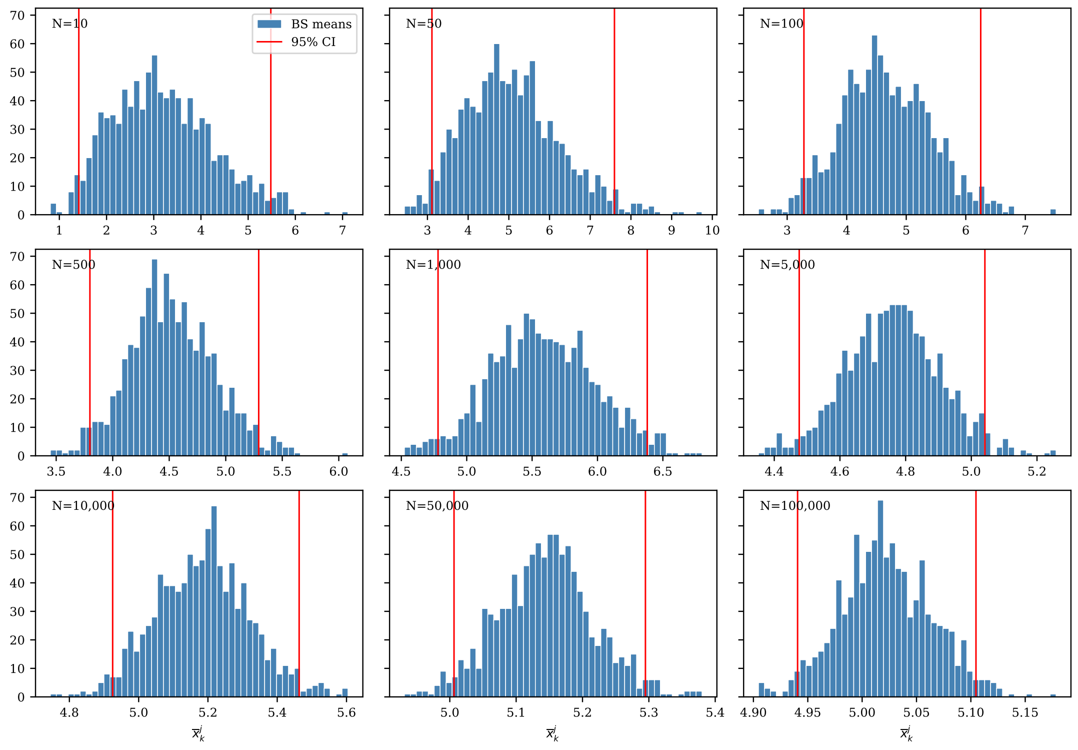
```
    mean_hat[k] = x_k

    # bootstrap means
    mean_bs = boostrap_mean(data, Nbs)

    # CI lower and upper bounds at (p2.5, p97.5)
    rank = 2.5, 97.5
    bnd = np.percentile(mean_bs, q=rank)
    bounds[k] = bnd

    # Store in list of all bootstrapped means
    # so we can plot histogram later
    means_all.append(mean_bs)


# Plot results
plt.plot(sample_sizes, mean_hat, lw=2.0, label='Estim. mean')

# Add line indicating true mean of log-normal
mean = np.exp(mu + sigma**2.0 / 2.0)
plt.axhline(mean, lw=1.0, color='black', ls='--')
plt.text(sample_sizes[0], mean + 0.05, 'True mean', va='bottom',
         fontstyle='italic', fontfamily='serif')

plt.fill_between(sample_sizes, bounds[:, 0], bounds[:, 1],
                 color='grey', alpha=0.2, zorder=-1, lw=0.0,
                 label='95% CI')
plt.xscale('log')
plt.legend(loc='lower right')
plt.xlabel('Sample size (log scale)')
plt.ylabel('Mean')
# Use identical y-lims across ex. 4-6
plt.ylim((1.0, 8.0))
```

[31]:
```
(1.0, 8.0)
```



Finally, the code below creates the histograms for each sample size. Note the different plot limits on the *x*-axis: the bootstrapped means for larger sample sizes are much closer together.

[32]:
```
fig, axes = plt.subplots(3, 3, sharex=False, sharey=True, figsize=(10, 7))

for k, ax in enumerate(axes.flatten()):
    N_k = sample_sizes[k]
    means_k = means_all[k]
    ax.hist(means_k, bins=50, color='steelblue', lw=0.5,
            edgecolor='white', label='BS means')
    ax.axvline(bounds[k, 0], color='red', lw=1.0, label='95% CI')
    ax.axvline(bounds[k, 1], color='red', lw=1.0)

    # add sample size
    ax.text(0.05, 0.95, f'N={N_k:,d}', transform=ax.transAxes, va='top')

    if k == 0:
        ax.legend(loc='upper right')

    if k > 5:
        ax.set_xlabel(r'$\overline{x}_{k}^j$')

fig.tight_layout()
```



As an aside, the use of the bootstrap in this scenario is somewhat nonsensical and only serves to practice doing data analysis with Python. We use bootstrapping in settings where we don't know the underlying distribution, so we are forced to use the data we have as the population from which we resample with replacement. Here, we of course know that the data is log-normally distributed, so we could just draw new samples from that distribution directly.

9 Introduction to unsupervised learning

9.1 Overview of machine learning (ML) algorithms

Broadly speaking, we can categorize machine learning algorithms into three groups:

1. *Supervised learning*

   Models in this group use both input data (often called independent variables, features, covariates, predictors, or X variables) and the corresponding output data (dependent variable, outcome, target, or y variable) to establish some relationship $y = f(X)$ within a training data set. We can then use this relationship to make predictions about outputs in new data.

   Subcategories within this group include:

   - Regression, where output data is allowed to take on continuous values; and
   - Classification, where output data is restricted to a few values, often called categories or labels.

2. *Unsupervised learning*

   In this scenario, machine learning algorithms operate on unlabelled data, i.e., there is no explicit outcome variable. We can, however, use machine learning to structure or reduce this data, for example by

   - Clustering, where we organise data into meaningful subgroups (clusters); or
   - Dimensionality reduction, where possibly high-dimensional data is compressed into fewer dimensions while preserving relevant information.

     One of the most widely used examples of dimensionality reduction is principal component analysis (PCA) which we study in more detail below.

3. *Reinforcement learning*

   We won't be concerned with ML algorithms that fall into this category in this course.

9.2 Principal component analysis

Principal component analysis (PCA) is one of the most widely used dimensionality reduction techniques. Assume we have a dataset consisting of $i = 1, \dots, N$ observations of $k = 1, \dots, K$ variables (or features) $\mathbf{x}_k$. For simplicity, assume that all $\mathbf{x}_k$ are *centred*, i.e., the have been transformed so that they have zero means. We could image two ways to reduce the dimensionality of this dataset:

1. Discard some of the variables and keep only a number $J \ll K$.
2. Create a collection of alternative variables $\mathbf{p}_j$ which are linear combinations of $\mathbf{x}_k$. The dimensionality reduction arises if we form only $J \ll K$ such variables.

The variables $\mathbf{p}_j$ are called principle components if we construct them in a particular way. Let $\mathbf{p}_1$ be the first such component which is a linear combination of all $\{\mathbf{x}_k\}$,

$$\mathbf{p}_1 = v_{11}\mathbf{x}_1 + v_{21}\mathbf{x}_2 + \cdots + v_{K1}\mathbf{x}_K = \sum_{k=1}^{K} v_{k1}\mathbf{x}_k$$

where the v_{1k} are called *coefficients* or *loadings* (note that the term "loadings" is not used consistently in the literature and sometimes refers to a rescaled version of v). We want to pick these coefficients in some optimal fashion, which in this case is by requiring that they maximise the sample variance of the first principle component subject to the constraint that $\sum_{k=1}^{K} v_{k1}^2 = 1$. Therefore, these coefficients are the solution to the following maximisation problem:

$$\max_{v_{11},v_{21},\dots,v_{K1}} \left\{ \frac{1}{n} \sum_{i=1}^{N} p_{i1}^2 \right\} = \left\{ \frac{1}{n} \sum_{i=1}^{N} \left(\sum_{k=1}^{K} v_{k1} x_{ik} \right)^2 \right\} \quad \text{subject to} \quad \sum_{k=1}^{K} v_{k1}^2 = 1$$

We never solve the above maximisation problem by hand to find the coefficients but instead use an algorithm from linear algebra called the *singular value decomposition*.

9.2.1 Singular value decomposition and principal components

To compute the principal components, we often use a matrix factorisation technique known as singular value decomposition (SVD). Alternatively (and equivalently), principal components can be computed from the eigenvalues and eigenvectors of the data's covariance matrix. We ignore this second approach in this unit.

SVD is a matrix factorisation that is commonly used in econometrics and statistics. For example, we can use it to implement PCA, principal component regression, OLS or Ridge regression (which we cover in the next unit).

Let $\mathbf{X} \in \mathbb{R}^{N\times K}$ be a matrix of data with N observations (in rows) of K variables (in columns) with $N \geq K$. The (compact) SVD of $\mathbf{X}$ is given by

$$\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^\top$$

where $\mathbf{U} \in \mathbb{R}^{N\times K}$ and $\mathbf{V} \in \mathbb{R}^{K\times K}$ are orthogonal matrices, and $\mathbf{\Sigma} \in \mathbb{R}^{K\times K}$ is a diagonal matrix

$$\mathbf{\Sigma} = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_K \end{bmatrix}$$

The elements σ_k are called singular values of $\mathbf{X}$, and $\mathbf{\Sigma}$ is arranged such that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_K$. Since $\mathbf{U}$ is not necessarily square, it's not truly orthogonal, but its columns are still orthogonal to each other.

These matrices satisfy the following useful properties:

$$\begin{aligned} \mathbf{U}^\top\mathbf{U} &= \mathbf{I}_n \\ \mathbf{V}^\top\mathbf{V} = \mathbf{V}\mathbf{V}^\top &= \mathbf{I}_n \\ \mathbf{V}^\top &= \mathbf{V}^{-1} \end{aligned}$$

where $\mathbf{I}_n$ is the $n \times n$ identity matrix.

Once we have obtained the singular value decomposition, we can project the data $\mathbf{X}$ onto the coordinate system underlying the principal components to obtain the transformed data $\mathbf{P}$:

$$\mathbf{P} = \mathbf{X}\mathbf{V}$$

Intuitively, the j-th column of $\mathbf{V}$ defines how the j-th column in $\mathbf{P}$ is obtained as a linear combination of the columns of $\mathbf{X}$. If you recall our definition of the first principle component from above,

$$\mathbf{p}_1 = v_{11}\mathbf{x}_1 + v_{21}\mathbf{x}_2 + \cdots + v_{K1}\mathbf{x}_K$$

you can immediately see that the coefficients v_{k1} are stored in the first column of $\mathbf{V}$, and the first column of $\mathbf{P}$ corresponds to the first principal component $\mathbf{p}_1$.

The power of PCA comes from the fact that we don't need to use all columns in $\mathbf{V}$ so that the dimension of $\mathbf{P}$ is lower than the dimension of the original data $\mathbf{X}$.

In Python, we compute the SVD using the `svd()` function from `numpy.linalg`.

9.2.2 Example: Bivariate normal sample

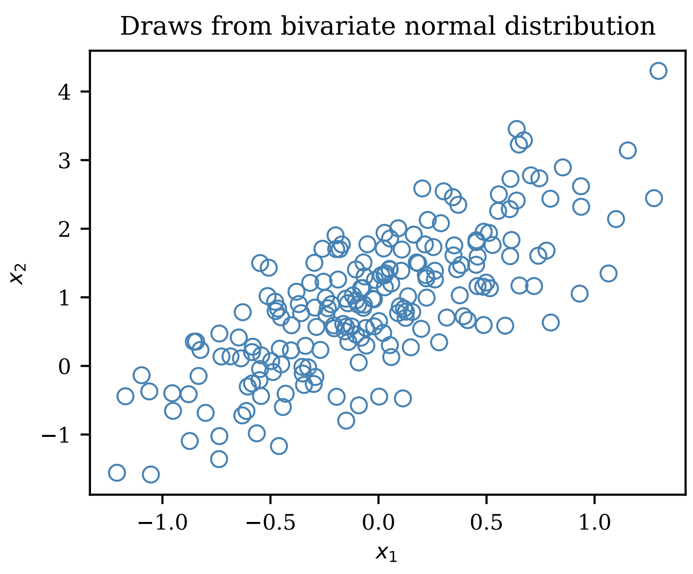
Imagine we construct $\mathbf{X}$ as 200 random draws from a bivariate normal:

[1]:
```
import numpy as np
from numpy.random import default_rng

# Draw a bivariate normal sample using the function we defined above
mu = np.array([0.0, 1.0])            # Vector of means
sigma = np.array([0.5, 1.0])         # Vector of standard deviations
rho = 0.75                           # Correlation coefficient
Nobs = 200                           # Sample size

# Correlation matrix
corr = np.array([
    [1.0, rho],
    [rho, 1.0]
])

# Create variance-covariance matrix
vcv = sigma[:, None] * corr * sigma

# Draw multivariate normal random numbers:
# each row represents one sample draw.
rng = default_rng(123)
X = rng.multivariate_normal(mean=mu, cov=vcv, size=Nobs)

# Split into variables X1, X2
x1, x2 = X.T
```

We use a scatter plot to visualise the random draws:

[2]:
```
import matplotlib.pyplot as plt

# Scatter plot of sample
plt.scatter(x1, x2, linewidths=0.75, c='none', edgecolors='steelblue')
plt.xlabel(r'$x_1$')
plt.ylabel(r'$x_2$')
plt.title('Draws from bivariate normal distribution')
```

[2]:
```
Text(0.5, 1.0, 'Draws from bivariate normal distribution')
```



Performing PCA manually using SVD

In a first step, we compute the principal components manually using SVD. Later on, we will examine how we do the same task using `scikit-learn`, one of the most widely used Python libraries for ML.

Before performing PCA, it is recommended to standardise the variables, i.e., transform them so that they have zero mean and unit variance. For this example, we will only demean the data but ignore the variance.

[3]:
```
# Demean variables

# Mean of each column
Xmean = np.mean(X, axis=0)

# Matrix Xcen stores the centred (demeaned) columns of X
Xcen = (X - Xmean)
```

We can now use the SVD factorisation to compute the principal components. Once we have computed the matrix $\mathbf{V}$, the data is transformed using the matrix multiplication

$$\mathbf{P} = \mathbf{X}\mathbf{V}$$

where $\mathbf{X}$ now denotes the standardised values.

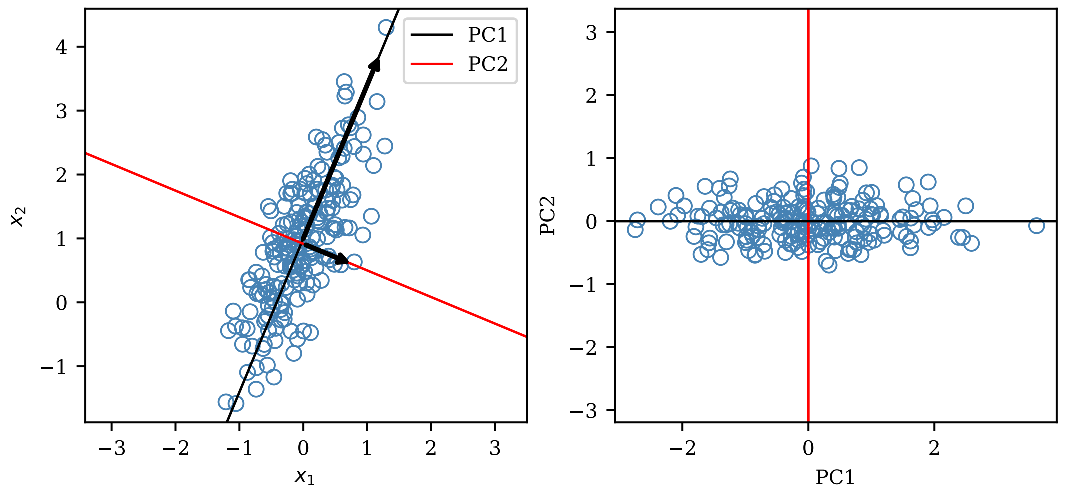
[4]:
```
from numpy.linalg import svd

# Apply SVD to standardised values. IMPORTANT: use full_matrices=False,
# otherwise SVD can take a long time and consume lots of memory!
U, S, Vt = svd(Xcen, full_matrices=False)

# Project onto principal components coordinates
PC = Xcen @ Vt.T

# Variance is highest for first component
var_PC = np.var(PC, axis=0, ddof=1)
print(f'Principal component variances: {var_PC}')
```

```
Principal component variances: [1.17607859 0.09444617]
```

We next plot the principal components in the original data space (left panel). Moreover, the right panel shows the data rotated and rescaled so that each axes corresponds to a principal component. Most of the variation clearly occurs along the first axis.

[5]:
```
# Plot principal components

# Scatter plot of sample
fig, (ax0, ax1) = plt.subplots(1, 2, figsize=(7,3))
ax0.scatter(X[:, 0], X[:, 1], linewidths=0.75, c='none', edgecolors='steelblue')
ax0.axis('equal')
ax0.set_xlabel(r'$x_1$')
ax0.set_ylabel(r'$x_2$')
ax0.axline(Xmean, Xmean + Vt[0], label='PC1', lw=1.0, c='black', zorder=1)
ax0.axline(Xmean, Xmean + Vt[1], label='PC2', lw=1.0, c='red', zorder=1)

# Plot arrows pointing along axes of individual components
PC_arrows = Vt * np.sqrt(var_PC[:, None])
for v in PC_arrows:
    # Scale up arrows by 3 so that they are visible!
    ax0.annotate('', Xmean + v*3, Xmean, arrowprops=dict(arrowstyle='->', linewidth=2))

ax0.legend()

# Plot in principal component coordinate system
ax1.scatter(PC[:, 0], PC[:, 1], linewidths=0.75, c='none', edgecolors='steelblue')
ax1.set_xlabel('PC1')
ax1.set_ylabel('PC2')
ax1.axis('equal')
ax1.axvline(0.0, lw=1.0, c='red')
ax1.axhline(0.0, lw=1.0, c='black')
```

[5]:
```
<matplotlib.lines.Line2D at 0x7f1a4b710c10>
```



Performing PCA using scikit-learn

In real applications we of course don't need to manually compute the principal components but can use a library such as `scikit-learn` to do it for us (see the documentation for `PCA` and the section on PCA in the official user guide for details).

Most models implemented in `scikit-learn` follow the same paradigm:

1. We create an instance of a class that presents the model we want to fit to the data. In our present case, the class is called `PCA`.

In many cases, we specify arguments that govern how a model is fit to the data, e.g., the number of principal components to use.

2. We fit the model (on the training data set) by calling the `fit()` method.
3. Frequently, we can use the `transform()` method to transform any other data (e.g., the test or validation sample) using the fitted model.

Note that in the case of PCA, `scikit-learn` automatically demeans the input data (but does not normalise the variance to 1), so we don't need to do it manually.

[6]:
```
from sklearn.decomposition import PCA

# Use same data X as before

# Create PCA with 2 components (which is the max, since we have only two
# variables)
pca = PCA(n_components=2)

# Perform PCA on input data
pca.fit(X)
```

[6]:
```
PCA(n_components=2)
```

To obtain the transformed data, we call the `transform()` method. Note that we could have called `fit_transform()` instead to perform the two previous steps in a single call.

[7]:
```
# Obtain data projected onto principal components
PC_skl = pca.transform(X)

# Check that these are identical to PC we obtained manually
assert np.all(np.abs(PC_skl - PC) < 1.0e-10)
```

The principal component coefficients (used to create the transformed data) are stored in the attribute `components_` and are identical to the matrix $\mathbf{V}^\top$ we computed above.

[8]:
```
# The attribute components_ can be used to retrieve the V' matrix
print("components_ (matrix V'):")
print(pca.components_)

# Check that these are identical to the matrix V' we computed via SVD
assert np.all(np.abs(pca.components_ - Vt) < 1.0e-10)
```

```
components_ (matrix V'):
[[ 0.38420018  0.92324981]
 [ 0.92324981 -0.38420018]]
```

The fitted `pca` object contains other useful attributes (see the documentation for a full list). For example,

- `explained_variance_` stores the variances of all principal components; and
- `explained_variance_ratio_` stores the fraction of the variance "explained" by each component.

[9]:
```
# The attribute explained_variance_ stores the variances of all PCs
print(f'Variance of each PC: {pca.explained_variance_}')

# Fraction of variance explained by each component:
print(f'Fraction of variance of each PC: {pca.explained_variance_ratio_}')
```

```
Variance of each PC: [1.17607859 0.09444617]
Fraction of variance of each PC: [0.92566365 0.07433635]
```

From the above output, we see that the first principal component captures about 92% of the variance in the data.

Finally, it is often interesting to examine how much any of the original variables in $\mathbf{X}$ contribute to each principal component. These contributions are called *loadings* (again, note that the term "loadings" is not used consistently) and can be computed as follows:

```
[10]: loadings = pca.components_.T * np.sqrt(pca.explained_variance_)

      # Use pandas DataFrame to tabulate loadings
      import pandas as pd
      pd.DataFrame(loadings, columns=['PC1', 'PC2'], index=['X1', 'X2'])
```

```
[10]:          PC1       PC2
      X1  0.416654  0.283734
      X2  1.001238 -0.118073
```

The output tells us that the first principal component (which corresponds to the first column in the above matrix) loads more heavily on the second column of $\mathbf{X}$, which can also be seen from the previous graph.

9.2.3 Example: Higher-dimensional data

Creating highly correlated inputs

The previous example was meant as an introduction but did not really illustrate the dimension reduction of PCA. After all, with only two dimensions there was not much to be reduced! Consider now a higher-dimensional (but still artificial) example with 10 dimensions. For this purpose, we create highly correlated data as follows:

1. We draw N independent samples from a bivariate normal distribution,

$$\mathbf{z}_i \overset{\text{iid}}{\sim} N\left(\mathbf{0}, \begin{bmatrix}\sigma_1^2 & 0 \\ 0 & \sigma_2^2\end{bmatrix}\right)$$

   and stack them in the matrix $\mathbf{Z} \in \mathbb{R}^{N\times 2}$.
2. For some $2 \times K$ matrix $\mathbf{A}$ with $K \gg 2$, we compute

$$\mathbf{X} = \mathbf{Z}\mathbf{A}$$

   which gives us the higher-dimensional matrix $\mathbf{X} \in \mathbb{R}^{N\times K}$.

For illustrative purposes, we draw the elements of $\mathbf{A}$ from a normal distribution but we could have picked almost any other coefficients. The point of the example is that we take two independent variables in $\mathbf{Z}$ and create $K \gg 2$ variables in $\mathbf{X}$ which are linear combinations of $\mathbf{Z}$. Intuitively, many of the columns of $\mathbf{X}$ will be highly correlated since they were created from the same variation in $\mathbf{Z}$.

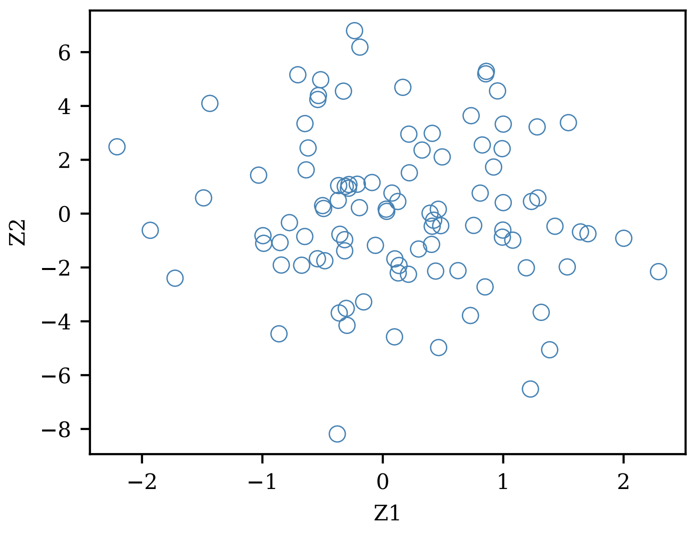
```
[11]: from numpy.random import default_rng
      import pandas as pd

      rng = default_rng(123)

      K = 10          # Number of columns in matrix X
      N = 100         # Number of observations

      # Std. dev. of columns in Z
      sigma = np.array([1.0, 3.0])

      # Draw 2 independent, normally distributed random variables and rescale
      # their variances
```

```
Z = rng.normal(size=(N, 2)) * sigma
```

We can plot the columns of **Z** against each other to verify that they don't seem to exhibit any particular dependence structure.

[12]:
```
import matplotlib.pyplot as plt

plt.scatter(Z[:, 0], Z[:, 1], color='none', edgecolors='steelblue', lw=0.5)
plt.xlabel('Z1')
plt.ylabel('Z2')
```

[12]:
```
Text(0, 0.5, 'Z2')
```



Next, we create the transformation matrix **A**. You can do this in various ways, so we simply choose to draw the elements of **A** from a standard normal distribution.

[13]:
```
A = rng.normal(size=(Z.shape[1], K))

# Print first three columns of A in transposed form
A.T[:3]
```

[13]:
```
array([[-0.93706677,  2.62894657],
       [-0.80933814,  0.45287643],
       [-0.41213169,  0.23403931]])
```

The above coefficients show us how the columns of **X** are formed: the first column is obtained as

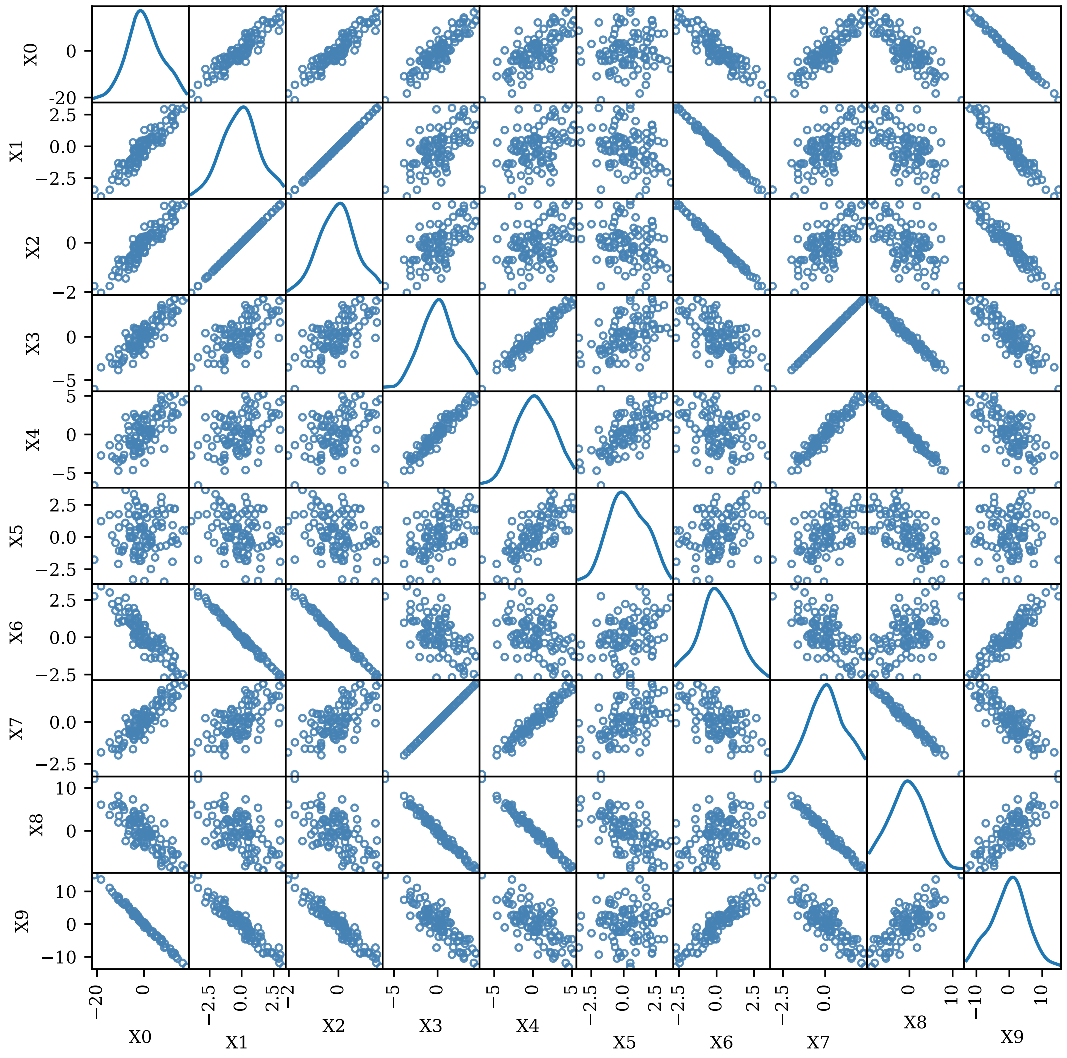
```
X[:, 0] = -0.937 * Z[:, 0] + 2.629 * Z[:, 1]
```

and so on.

[14]:
```
# Compute X as linear combinations of Z
X = Z @ A
```

By construction, the columns of **X** are highly correlated. We can illustrate this using pairwise scatter plots as follows:

[15]:
```
from pandas.plotting import scatter_matrix

df = pd.DataFrame(X, columns=[f'X{k}' for k in range(K)])

axes = scatter_matrix(df, figsize=(8, 8),
    alpha=0.9, color='none', edgecolor='steelblue',
    diagonal='kde')
```



Moreover, we can compute the pairwise correlation coefficients using pandas's `corr()` method which computes the correlation matrix between all columns of a `DataFrame`.

[16]:
```
# Print only three decimal places
pd.set_option('display.precision', 3)

# Compute and print correlation matrix
df.corr()
```

[16]:
```
       X0     X1     X2     X3     X4     X5     X6     X7     X8     X9
X0  1.000  0.931  0.933  0.893  0.739  0.067 -0.900  0.891 -0.824 -0.997
X1  0.931  1.000  1.000  0.667  0.443 -0.301 -0.997  0.665 -0.561 -0.955
X2  0.933  1.000  1.000  0.671  0.448 -0.295 -0.997  0.669 -0.565 -0.956
X3  0.893  0.667  0.671  1.000  0.963  0.510 -0.608  1.000 -0.991 -0.858
X4  0.739  0.443  0.448  0.963  1.000  0.722 -0.372  0.964 -0.991 -0.689
X5  0.067 -0.301 -0.295  0.510  0.722  1.000  0.373  0.513 -0.621  0.004
X6 -0.900 -0.997 -0.997 -0.608 -0.372  0.373  1.000 -0.605  0.495  0.929
X7  0.891  0.665  0.669  1.000  0.964  0.513 -0.605  1.000 -0.991 -0.857
X8 -0.824 -0.561 -0.565 -0.991 -0.991 -0.621  0.495 -0.991  1.000  0.782
X9 -0.997 -0.955 -0.956 -0.858 -0.689  0.004  0.929 -0.857  0.782  1.000
```

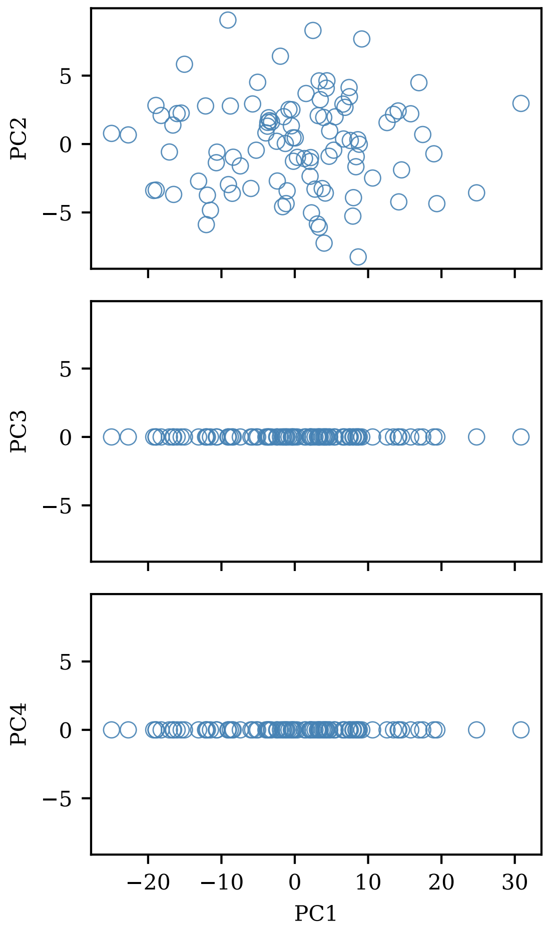
PCA with manually selected number of principle components

We now use `scikit-learn`'s `PCA` to perform the principal component analysis just as we did in the bivariate case.

[17]:
```
from sklearn.decomposition import PCA

# Create PCA using max. available components
pca = PCA()

# Perform PCA on input data
PC = pca.fit_transform(X)
```

To get some intuition for the transformed data, we plot the first principal component (which captures most of the variation by construction) against some of the other principal components. As you can see in the code below, there is some variation left in the 2nd principal component, while for the 3rd and 4th components the data along these dimension is basically constant (this is also the case for the remaining principal components).

[18]:
```
fig, axes = plt.subplots(3, 1, figsize=(3, 5), sharex=True, sharey=True)

# Dictionary of common keyword arguments for scatter() function
kw = dict(color='none', alpha=0.9, edgecolor='steelblue', lw=0.5)

# Indices of PCs to plot against 1st PC
yi = [1, 2, 3]

for i, k in enumerate(yi):
    axes[i].scatter(PC[:, 0], PC[:, k], **kw)
    axes[i].set_ylabel(f'PC{k+1}')

axes[-1].set_xlabel('PC1')

fig.tight_layout()
```

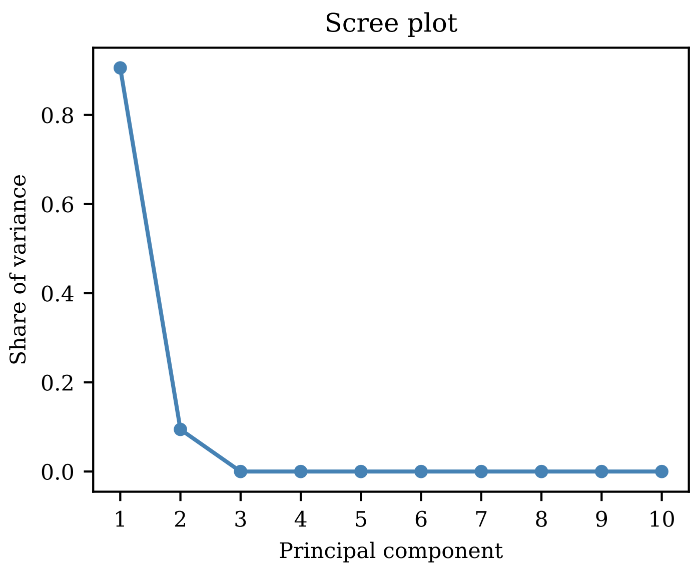


To see what's going on, we create a graph that plots the share of total variance captured by each component. This type of plot is called a *scree plot* and is occasionally used to visually pin down the number of principal components to use.

The plot below shows that for principal components beyond the 2nd one, this share is zero. This should come as no surprise since we generated the higher-dimensional data in **X** from only two dimensions of independent data.

[19]:
```python
import matplotlib.pyplot as plt

xvalues = np.arange(K)
plt.plot(xvalues, pca.explained_variance_ratio_, color='steelblue',
    marker='o', ms=4)
plt.xticks(xvalues, xvalues + 1)
plt.xlabel('Principal component')
plt.ylabel('Share of variance')
plt.title('Scree plot')
```

[19]:
```
Text(0.5, 1.0, 'Scree plot')
```



PCA with automatically selected number of principal components

Previously, we manually selected the number of principle components when constructing an instance of `PCA` (or we used the default, which takes the minimum of the number of rows and columns of $\mathbf{X}$). Alternatively, we can tell `scikit-learn` to automatically determine the number of components for us. For example, when the argument `n_components` is a floating-point number in (0, 1), `scikit-learn` interprets this as the minimum fraction of variance that should be explained and chooses the required number of principal components accordingly. To illustrate, let's perform PCA and request that the number of components should capture at leat 90% of the variance:

[20]:
```
# Perform PCA, select components to capture 90% of variance
pca = PCA(n_components=0.9)

# Perform PCA on input data
PC = pca.fit_transform(X)
```

This selects only the first principal component which is what we would suspect when looking at the previous graph.

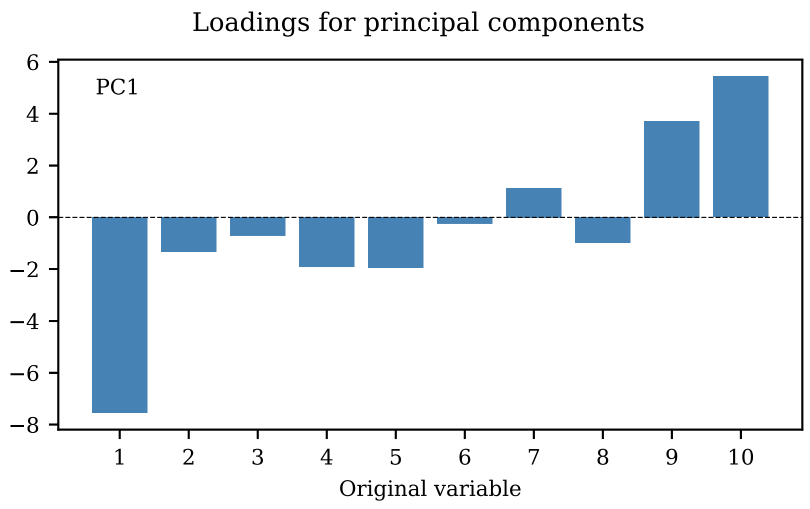
[21]:
```
print(f'Number of components: {pca.n_components_}')
```

```
Number of components: 1
```

We conclude this section by plotting the loadings for each principal component (which happens to be only one in this case). Since the data was generated randomly, this plot is not particularly insightful but will be much more useful with real data.

[22]:
```
# Compute loadings for all PCs
loadings = pca.components_.T * np.sqrt(pca.explained_variance_)

# Number of selected PCs
Ncomp = pca.n_components_

fig, axes = plt.subplots(Ncomp, 1, figsize=(5, 2.5 * Ncomp),
    sharex=True, sharey=True)

# Plot loadings for each PC
xvalues = np.arange(K)
for i, ax in enumerate(np.atleast_1d(axes)):
    ax.bar(xvalues, loadings[:, i], color='steelblue')
    ax.set_xticks(xvalues, xvalues+1)
    ax.set_xlabel('Original variable')
    ax.text(0.05, 0.95, f'PC{i+1}', transform=ax.transAxes,
        ha='left', va='top')
    ax.axhline(0.0, lw=0.5, ls='--', c='black')

fig.suptitle(f'Loadings for principal components')
```

```
[22]: Text(0.5, 0.98, 'Loadings for principal components')
```



9.3 Optional exercises

Exercise 1: PCA of uncorrelated variables

Imagine we have data $\mathbf{X}$ with $N = 1000$ observations of a randomly drawn sample of $K = 5$ variables where each variable is standard-normally distributed. Assume that all variables are uncorrelated.

Since these variables are uncorrelated and identically distributed, we can simply draw an array of shape (N, K) from the univariate normal distribution as follows instead of using the multivariate normal distribution:

```
[23]: import numpy as np

rng = np.random.default_rng(123)

K = 5
Nobs = 1000

X = rng.normal(size=(Nobs, K))
```

Using this random sample, perform the following tasks:

1. Create pairwise scatter plots of all variables in $\mathbf{X}$.
2. Perform a principal component analysis on this sample using `scikit-learn`'s `PCA` implementation.
3. Create pairwise scatter plots of all principal components. Do these plots look any different from the ones you created earlier? Explain your finding!
4. Generate a scree plot showing the fraction of variance captured by each principal component.
5. Is it possible to reduce the dimensionality of this data without sacrificing too much information? Why or why not?

Exercise 2: PCA of quarterly business cycle indicators

In this exercise, we use the quarterly data from FRED on GDP, CPI and the unemployment rate (UNRATE) in the United States stored in the CSV file `FRED_QTR.csv` in the `data/` folder. To load the data, proceed as follows:

[24]:
```python
# Uncomment this to use files in the local data/ directory
DATA_PATH = '../data'

# Load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/python-intro-PGR/main/data'

import pandas as pd

file = f'{DATA_PATH}/FRED_QTR.csv'
df = pd.read_csv(file, sep=',')
```

Generate the following variables which serve as indicators for the business cycle (the cyclical state of the economy):

1. Compute GDP growth as the percentage change of GDP between two periods.
2. Compute inflation as the percentage change of the CPI between two periods.
3. Compute differences in the unemployment rate between two periods (this is measured in percentage points).

Note that you will need to drop the first row with missing observations. This can be done using `dropna()`.

Perform the following tasks using the data you just created:

1. Compute and report the correlation matrix for all three business cycle indicators. Which variables are particularly highly correlated?
2. Create pairwise scatter plots of all three business cycle indicators.
3. Perform a principal component analysis using `scikit-learn`'s `PCA` implementation.
4. Tabulate the loadings for each component.
5. Create a scree plot showing the fraction of variance explained by each component. How many components would one need to capture 90% of the sample variance?

9.4 Solutions

Solution for exercise 1

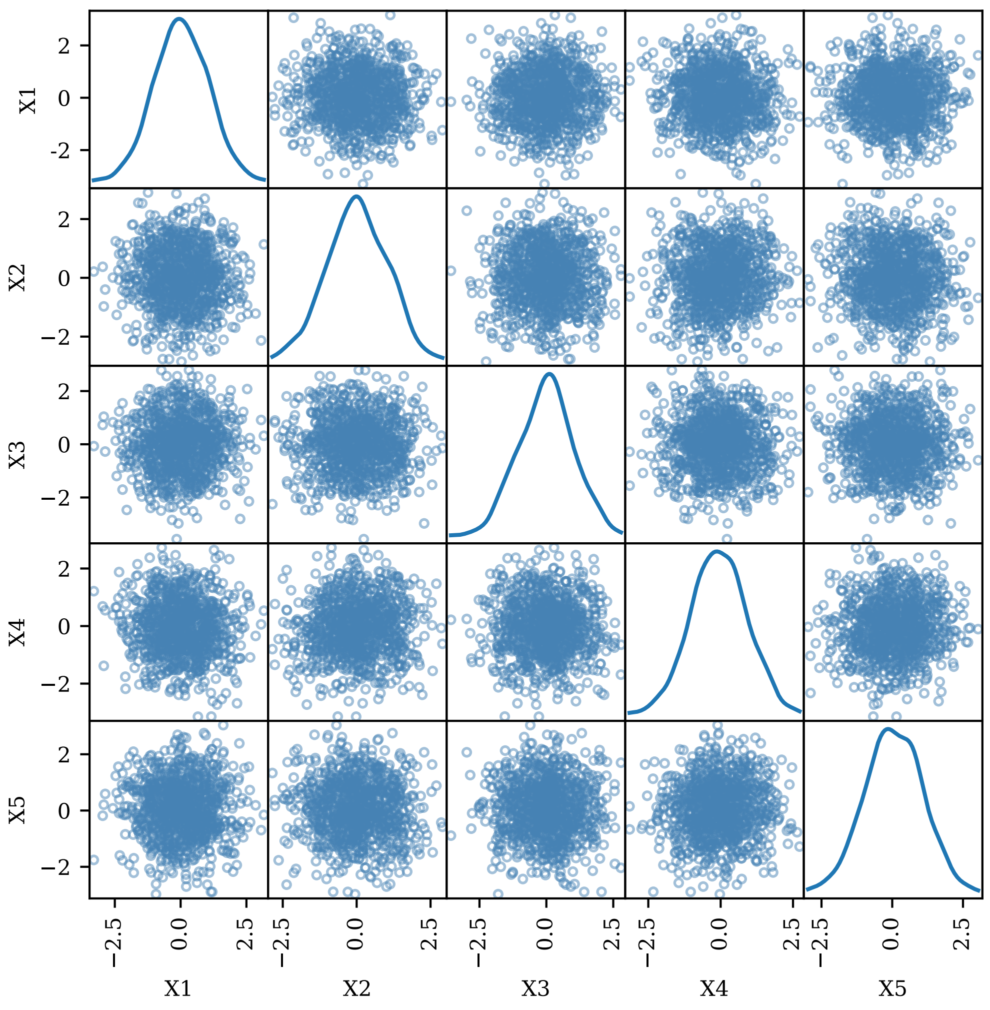
We generate the random sample as instructed:

[25]:
```python
import numpy as np

rng = np.random.default_rng(123)

K = 5
Nobs = 1000

X = rng.normal(size=(Nobs, K))
```

We use the function `scatter_matrix()` provided in `pandas.plotting` to create pairwise scatter plots for all 5 variables. As can be seen, these variables look uncorrelated since this is how we constructed them in the first place.

[26]:
```python
from pandas.plotting import scatter_matrix

# Create temporary DataFrame for scatter plots
```

```
df = pd.DataFrame(X, columns=[f'X{i}' for i in range(1, K+1)])

_ = scatter_matrix(df, figsize=(6,6), diagonal='kde',
    edgecolor='steelblue', color='none', alpha=0.5)
```

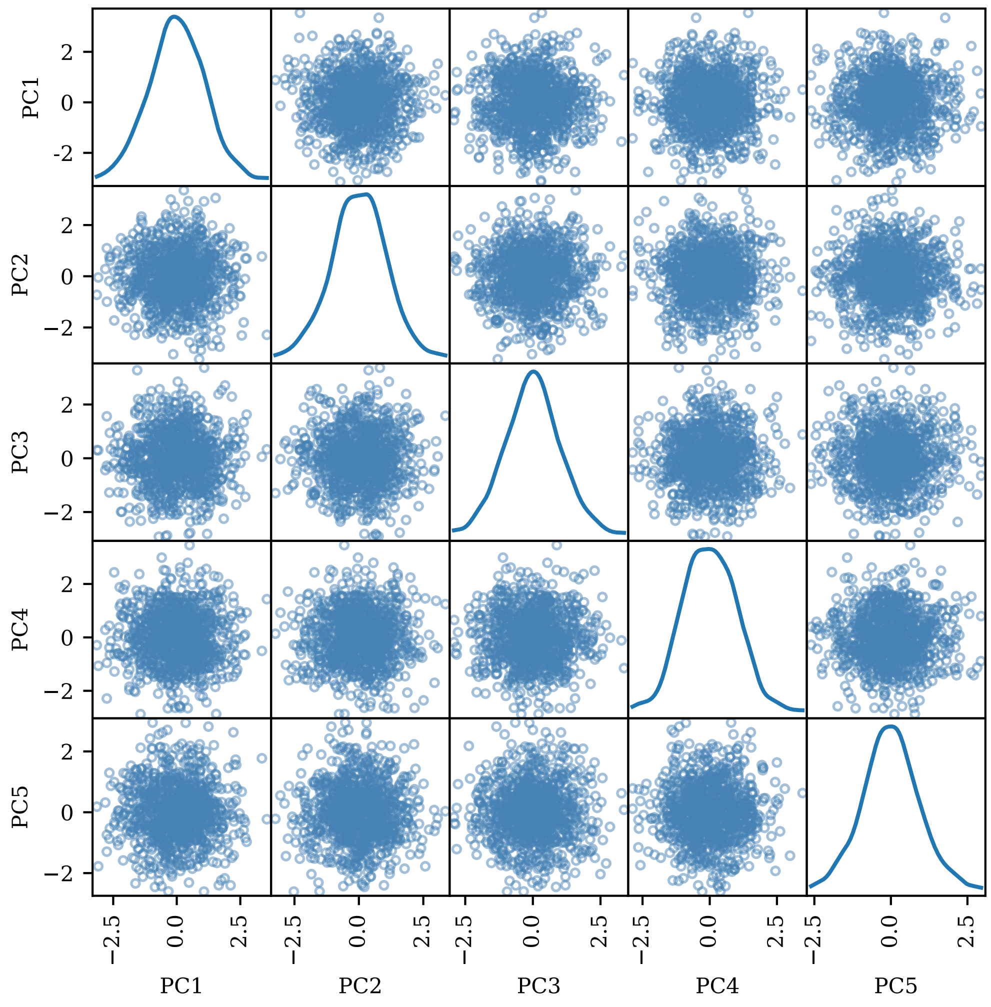


To perform the principal component analysis, we use the `sklearn.decomposition.PCA` implementation. Note that `PCA.fit()` automatically centres the data (which is not needed here since it is already mean zero).

```
[27]: from sklearn.decomposition import PCA

pca = PCA()

# Fit principal components, apply transformation to generate PCs
PC = pca.fit_transform(X)
```

As before, we use the `scatter_matrix()` function to create pairwise scatter plots of all principal components. Note that the principal components are uncorrelated by construction, even if the original data was correlated (which was not the case here).

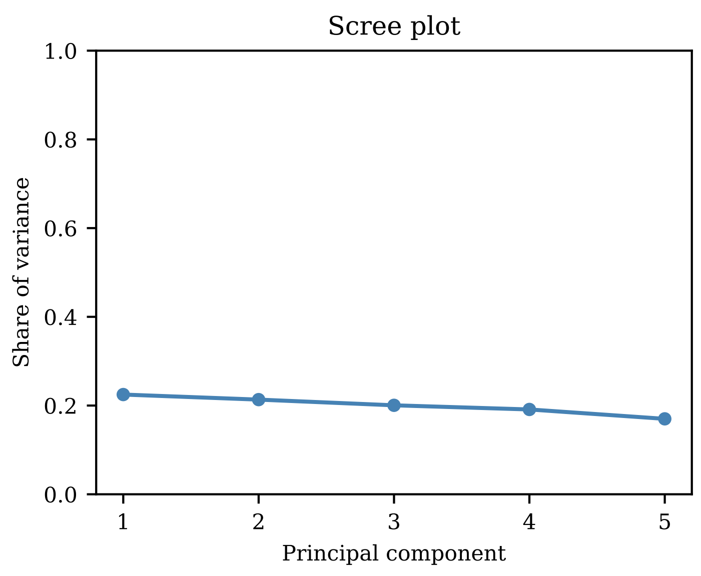
```
[28]: from pandas.plotting import scatter_matrix

# Store PCs in DataFrame
df = pd.DataFrame(PC, columns=[f'PC{i}' for i in range(1, K+1)])

_ = scatter_matrix(df, figsize=(6,6), diagonal='kde',
    edgecolor='steelblue', color='none', alpha=0.5)
```



Lastly, we use the attribute `explained_variance_ratio_` to obtain the fraction of the variance captured by each individual principal component.

[29]:
```
import matplotlib.pyplot as plt

# Create scree plot of explained variance for each component
xvalues = np.arange(1, K+1)
plt.plot(xvalues, pca.explained_variance_ratio_,
    marker='o', ms=4, c='steelblue')
plt.ylim((0.0, 1.0))
plt.xticks(xvalues)
plt.xlabel('Principal component')
plt.ylabel('Share of variance')
plt.title('Scree plot')
```

[29]:
```
Text(0.5, 1.0, 'Scree plot')
```



As the plot shows, each principal component captures approximately the same share of variance. It therefore does not really make sense to apply dimensionality reduction to this data set, since all original variables were already uncorrelated.

Solution for exercise 2

We first load the CSV file and keep only the relevant columns `GDP`, `CPI` and `UNRATE`.

```
[30]: # Uncomment this to use files in the local data/ directory
DATA_PATH = '../data'

# Load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/python-intro-PGR/main/data'
```

```
[31]: import numpy as np
import pandas as pd

file = f'{DATA_PATH}/FRED_QTR.csv'

df = pd.read_csv(file, sep=',')

# Keep only columns of interest
df = df[['GDP', 'CPI', 'UNRATE']]

# Tabulate first 5 observations
df.head()
```

```
[31]:       GDP   CPI  UNRATE
0  2086.0  23.6     3.7
1  2120.4  24.0     3.7
2  2132.6  24.4     3.8
3  2135.0  24.2     3.8
4  2105.6  23.9     4.7
```

We apply the required transformations to obtain inflation from relative changes in CPI and the GDP growth from relative changes in GDP. We obtain changes in the unemployment rate (measured in percentage points) as a simple difference.

```
[32]: # Compute inflation
      df['Inflation'] = df['CPI'].pct_change() * 100.0

      # Compute GDP growth
      df['GDP_growth'] = df['GDP'].pct_change() * 100.0

      # Compute absolute change in unemployment rate
      df['UNRATE_diff'] = df['UNRATE'].diff()
```

Next, we discard all other columns and drop observations with missing values using the `dropna()` method.

```
[33]: # Keep only (relative) changes, drop all other columns
      variables = ['GDP_growth', 'Inflation', 'UNRATE_diff']
      df = df[variables]

      # Differences create missing values for first observation, drop these
      df = df.dropna()
```

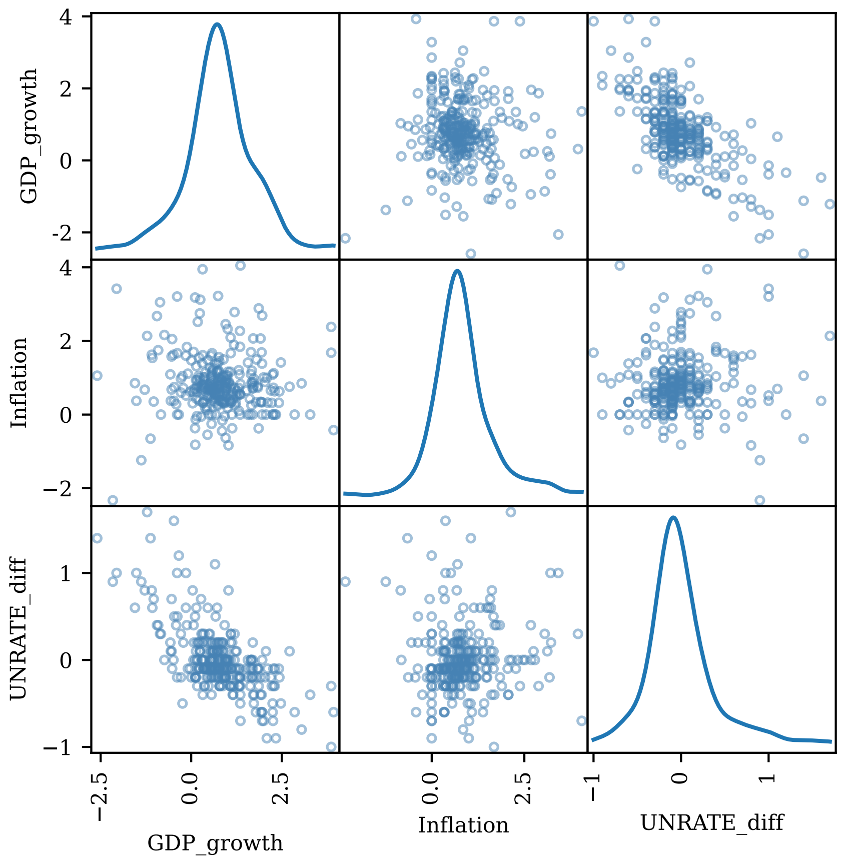
We compute the correlation matrix using the `corr()` method. The result shows that GDP growth is particularly (negatively) correlated with changes in unemployment, i.e., if the economy is growing, unemployment is going down. This is in line with economic intuition.

```
[34]: df.corr()
```

```
[34]:              GDP_growth  Inflation  UNRATE_diff
      GDP_growth        1.000     -0.065       -0.678
      Inflation        -0.065      1.000        0.025
      UNRATE_diff      -0.678      0.025        1.000
```

The pairwise scatter plots for all growth rates and relative changes can be obtained with `scatter_matrix()`, as usual.

```
[35]: from pandas.plotting import scatter_matrix

      _ = scatter_matrix(df[variables], figsize=(5,5), diagonal='kde',
          edgecolor='steelblue', color='none')
```



We now perform the principal component analysis. To do this, we convert the `DataFrame` to a regular NumPy array.

[36]:
```
from sklearn.decomposition import PCA

pca = PCA()

# perform PCA. Note that sklearn is programmed to work with NumPy arrays,
# so convert DataFrame to array
X = df.to_numpy()
pca.fit(X)
```

[36]:
```
PCA()
```

The loadings for each principal component are obtained as follows. We see that the first PC load particularly strongly on (negative) GDP growth.

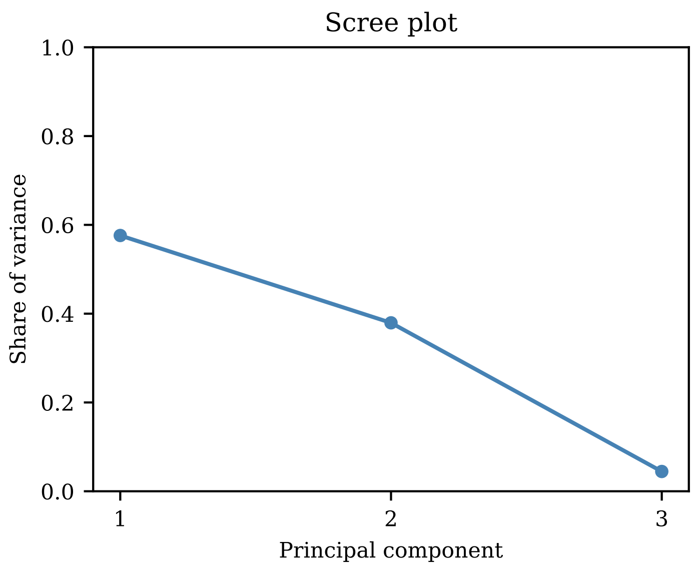
[37]:
```
loadings = pca.components_.T * np.sqrt(pca.explained_variance_)

# Print pretty table with loadings
pd.DataFrame(loadings,
    columns=[f'PC{i}' for i in range(3)],
    index=variables,
)
```

[37]:
```
                PC0    PC1    PC2
GDP_growth   -0.926 -0.112  0.080
Inflation     0.149 -0.785  0.003
UNRATE_diff   0.282  0.044  0.260
```

Lastly, we create the scree plot using the `explained_variance_ratio_` attribute. As you can see, the last principal component captures only a tiny fraction of the variance and could thus potentially be discarded.

[38]:
```
import matplotlib.pyplot as plt

K = len(variables)
xvalues = np.arange(1, K+1)

# Create scree plot
plt.plot(xvalues, pca.explained_variance_ratio_,
    marker='o', ms=4, c='steelblue')
plt.ylim((0.0, 1.0))
plt.xticks(xvalues)
plt.xlabel('Principal component')
plt.ylabel('Share of variance')
plt.title('Scree plot')
```

[38]:
```
Text(0.5, 1.0, 'Scree plot')
```



We can compute the cumulative variance explained by an increasing number of components as follows:

[39]:
```
np.cumsum(pca.explained_variance_ratio_)
```

[39]:
```
array([0.57606753, 0.9554459 , 1.        ])
```

To capture at least 90% of the variance, we only need the first two principal components.

10 Introduction to supervised learning

10.1 Simple (univariate) linear regression

Imagine the simplest linear model where the dependent variable y is assumed to be an affine function of the explanatory variable x and an error term ϵ, given by

$$y_i = \alpha + \beta x_i + \epsilon_i$$

for each observation i. In econometrics, the parameters α and β are called the intercept and slope parameters, respectively. In machine learning, the terminology often differs and you might see a simple linear model written like

$$y_i = b + w x_i + \epsilon_i$$

where b is called the *bias* and w is called a *weight*.

Our goal is to estimate the parameters α and β which is most commonly done by ordinary least squares (OLS). OLS is defined as the estimator that finds the estimates $(\widehat{\alpha}, \widehat{\beta})$ such that the sum of squared errors is minimized,

$$L(\alpha, \beta) = \frac{1}{N} \sum_i^N \Bigl(y_i - \alpha - \beta x_i \Bigr)^2$$

where L is the loss function that depends on the choice of parameters. Note that we use the "hat" notation $\widehat{\alpha}$ to distinguish the OLS estimate from the (usually unknown) true parameter α. The exact values of $\widehat{\alpha}$ and $\widehat{\beta}$ will vary depending the sample size and estimator used as we will see later in this unit.

For this simple model, the estimates are given by the expressions

$$\widehat{\beta} = \frac{\widehat{Cov}(y, x)}{\widehat{Var}(x)}$$

$$\widehat{\alpha} = \overline{y} - \widehat{\beta}\overline{x}$$

where $\widehat{Cov}(\bullet, \bullet)$ and $\widehat{Var}(\bullet)$ are the *sample* covariance and variance, respectively, and $\overline{y}$ and $\overline{x}$ are the sample means of y and x.

There is a straightforward generalization to the multivariate setting where we have a vector $\mathbf{x}_i$ of explanatory variables (which usually include the intercept) and a parameter vector $\boldsymbol{\beta}$ so that the model is given by

$$y_i = \mathbf{x}_i' \boldsymbol{\beta} + \epsilon_i$$

If we stack all $\mathbf{x}_i$ in the matrix $\mathbf{X}$ and all y_i in the vector $\mathbf{y}$, the OLS estimate of $\boldsymbol{\beta}$ is given by the well-known formula

$$\widehat{\boldsymbol{\beta}} = \left(\mathbf{X}'\mathbf{X}\right)^{-1} \mathbf{X}'\mathbf{y}$$

However, we will not be estimating linear regressions based on this formula and you should never attempt this, as naively implementing such matrix operations can lead to numerical problems. Instead, always use pre-packaged least-squares solvers such as those implemented in NumPy's `lstsq()` or SciPy's `lstsq()` functions. For econometrics and machine learning, it usually makes sense to use linear regression models such as those implemented in `statsmodels` or `scikit-learn`, which is what we turn to next.

10.1.1 Univariate linear regressions with scikit-learn

We start by fitting the simple linear model

$$y_i = \alpha + \beta x_i + \epsilon_i$$

using `scikit-learn`. For now, we proceed using synthetically generated data where we know the true relationship, assuming that

$$y_i = 1 + \frac{1}{2} x_i + \epsilon_i$$
$$\epsilon \overset{\text{iid}}{\sim} N(0, 0.7^2)$$

so that the true parameters are $\alpha = 1$ and $\beta = \frac{1}{2}$. The error term ϵ is assumed to be normally distributed with mean 0 and variance $0.7^2 = 0.49$. Note that it is customary to specify the normal distribution in terms of its variance (here 0.49), but most NumPy and SciPy functions expect the standard deviation (here 0.7) to be passed as the scale parameter of a normal distribution.

We generate a sample of $N = 30$ observations as follows:

[1]:
```
import numpy as np
from numpy.random import default_rng

rng = default_rng(123)

# Number of observations
N = 30

# True parameters
alpha = 1.0
beta = 0.5

# Use x that are uniformly spaced on [0, 10]
x = np.linspace(0, 10, N)
# Normally distributed errors
epsilon = rng.normal(scale=0.7, size=N)
# Create outcome variable
y = alpha + beta * x + epsilon
```

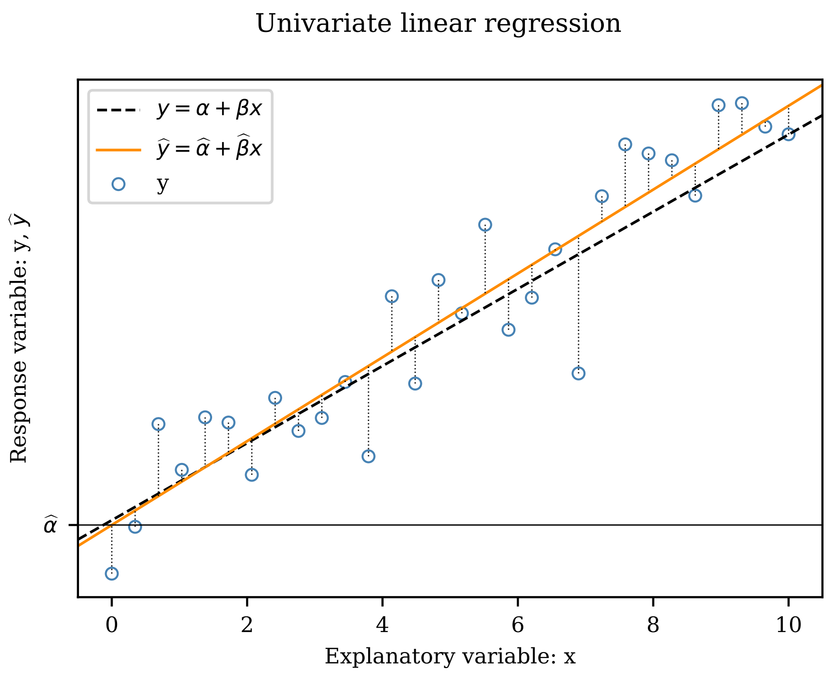
To fit a linear model, we use the `LinearRegression` class provided by `scikit-learn`. Before doing so, we need to import it from `sklearn.linear_model`. Note that most fitting routines in `scikit-learn` expect a *matrix* as opposed to a vector even if the model has only one explanatory variable, so we need to insert an artificial axis to create the matrix `X = x[:, None]`.

[2]:
```
from sklearn.linear_model import LinearRegression

# Create LinearRegression object
lr = LinearRegression(fit_intercept=True)

# fit() expects two-dimensional object, convert to N x 1 matrix
X = x[:, None]

# Fit model to data
lr.fit(X, y)

# Predict fitted values
yhat = lr.predict(X)
```

The following code visualises the sample as a scatter plot of (x_i, y_i) and adds the fitted line (in solid orange). Intuitively, for $x = 0$ the fitted line has the value $\widehat{\alpha}$ (the intercept) and its slope is equal to $\widehat{\beta}$. Note that `scikit-learn` models usually store the fitted model parameters in the `coef_` attribute (which is an array even if there is only a single explanatory variable). If the model includes an intercept, its fitted value is stored in the attribute `intercept_`.

[3]:
```
import matplotlib.pyplot as plt

# Extract parameter estimates from LinearRegression object
alpha_hat = lr.intercept_
beta_hat = lr.coef_[0]

fig, ax = plt.subplots(1, 1, figsize=(5, 3.5))

# Plot true relationship
ax.axline((0.0, alpha), slope=beta, lw=1.0, ls='--', c='black',
    label=r'$y = \alpha + \beta x$')

# Plot regression line
ax.axline((0.0, alpha_hat), slope=beta_hat, lw=1.0, c='darkorange',
    label=r'$\widehat{y} = \widehat{\alpha} + \widehat{\beta} x$')

# Plot raw data
ax.scatter(x, y, s=20, color='none', edgecolor='steelblue', alpha=1.0,
    lw=0.75, label='y')

# Plot lines connecting true and predicted values for
# each observation
for i in range(len(x)):
    # Predict yhat for given x_i
    yhat_i = alpha_hat + beta_hat * x[i]
    ax.plot([x[i], x[i]], [y[i], yhat_i], lw=0.5, ls=':',
        c='black', alpha=0.9)

# Add annotations
ax.set_xlabel('Explanatory variable: x')
ax.set_ylabel(r'Response variable: y, $\widehat{y}$')
ax.axhline(alpha_hat, lw=0.5, c='black')
ax.legend(loc='best')
ax.set_yticks((alpha_hat, ), (r'$\widehat{\alpha}$', ))

fig.suptitle('Univariate linear regression')
```

[3]:
```
Text(0.5, 0.98, 'Univariate linear regression')
```



The dashed black line shows the true relationship which is known in this synthetic sample, but in general is unknown for real data. For small sample sizes, the true and estimated parameters need not be close as the graph illustrates. The graph also shows the prediction errors as the dotted lines between the sample points y_i and the black fitted model. OLS minimizes the sum of the squares of these distances.

10.1.2 Training and test samples

In econometrics, we usually emphasise *inference*, i.e., we are interested in testing a hypothesis about the estimated parameter, for example whether it is statistically different from zero. Conversely, in machine learning the emphasis is often on prediction, i.e., our goal is to estimate a relationship from a *training* sample and make predictions for new data. Usually, we use a *test* sample that is different from the training data to assess how well our model is able to predict outcomes for new data which has not been used for estimation.

To demonstrate the use of training and test data sets, we use as simplified variant of the Ames house price data set which can be obtained from openml.org, a repository for free-to-use data suitable for machine learning tasks.

The original data set has 80 features (explanatory variables) which are characteristics of houses in Ames, a city of about 60 thousand inhabitants in the middle of Iowa, USA. The goal is to use these features to predict the house price (or "target" in ML terminology). The above website provides details on all 80 features, but we restrict ourselves to a small subset.

We could download the data set directly from openml.org using `scikit-learn` as follows:

```
from sklearn.datasets import fetch_openml
ds = fetch_openml(name='house_prices')

X = ds.data          # Get features (explanatory variables)
y = ds.target        # get dependent variable
```

which returns an object with various information about the data set. The features are stored in the `data` attribute, while the dependent variable is stored in the `target` attribute.

Instead, we will use a local copy of the simplified data set which contains only a subset of 12 features that are slightly adapted for our purposes.

[4]:
```
# Uncomment this to use files in the local data/ directory
DATA_PATH = '../data'

# Load data directly from GitHub (for Google Colab)
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/python-intro-PGR/main/data'
```

[5]:
```
import pandas as pd

file = f'{DATA_PATH}/ames_houses.csv'
df = pd.read_csv(file)

# List columns present in DataFrame
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 13 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   SalePrice         1460 non-null   float64
 1   LotArea           1460 non-null   float64
 2   Neighborhood      1460 non-null   object
 3   BuildingType      1386 non-null   object
 4   OverallQuality    1460 non-null   int64
 5   OverallCondition  1460 non-null   int64
 6   YearBuilt         1460 non-null   int64
 7   CentralAir        1460 non-null   object
 8   LivingArea        1460 non-null   float64
 9   Bathrooms         1460 non-null   int64
 10  Bedrooms          1460 non-null   int64
 11  Fireplaces        1460 non-null   int64
 12  HasGarage         1460 non-null   int64
dtypes: float64(3), int64(7), object(3)
memory usage: 148.4+ KB
```

In this section, we focus on the columns `SalePrice` which contains the house price in US dollars and `LivingArea` which contains the living area in m^2 for each house in the sample. To get some intuition for the data, we look at some descriptive statistics:

[6]:
```
df[['SalePrice', 'LivingArea']].describe()
```

[6]:
```
           SalePrice   LivingArea
count    1460.000000  1460.000000
mean   180921.195890   140.777444
std     79442.502883    48.813961
min     34900.000000    31.026587
25%    129975.000000   104.923743
50%    163000.000000   135.996777
75%    214000.000000   165.049367
max    755000.000000   524.107798
```

For our analysis, we convert the `DataFrame` columns to NumPy arrays because `scikit-learn` was not programmed to work with pandas `DataFrames`:

[7]:
```
features = ['LivingArea']
target = 'SalePrice'

# Convert DataFrame to NumPy arrays
y = df[target].to_numpy()
X = df[features].to_numpy()
```

Manually creating training and test samples

We next want to split the data set intro training and test sub-samples. We do this by randomly selecting a desired fraction of the data to be part of the training sample and assign the rest to the test sample. For this example, we assign 10% of observations to the test sample and the remainder to the training data set. Once we have randomly allocated 90% of observations to the training data set, we use the function `setdiff1()` to find the complementary test sample. This function returns the set difference of two one-dimensional arrays, i.e., all array elements from the first argument that are *not* present in the second array.

[8]:
```
import numpy as np
from numpy.random import default_rng

rng = default_rng(123)

# sample size
N = len(y)

# fraction of data used for test sample
test_size = 0.1

# training and test sample sizes
N_test = int(N * test_size)
```

```
N_train = N - N_test

# Randomly assign observations to training samples
itrain = rng.choice(np.arange(N), size=N_train, replace=False)
# Test sample is complement of training sample
itest = np.setdiff1d(np.arange(N), itrain)

# Select training sample
X_train = X[itrain]
y_train = y[itrain]

# Select test sample
X_test = X[itest]
y_test = y[itest]
```

Once we have split the sample, we estimate the model on the training sample.

[9]:
```
from sklearn.linear_model import LinearRegression

# Fit model on training sample
lr = LinearRegression()
lr.fit(X_train, y_train)

print('Regression coefficients')
print(f'  Intercept: {lr.intercept_:.1f}')
print(f'  Slope: {lr.coef_[0]:.1f}')
```

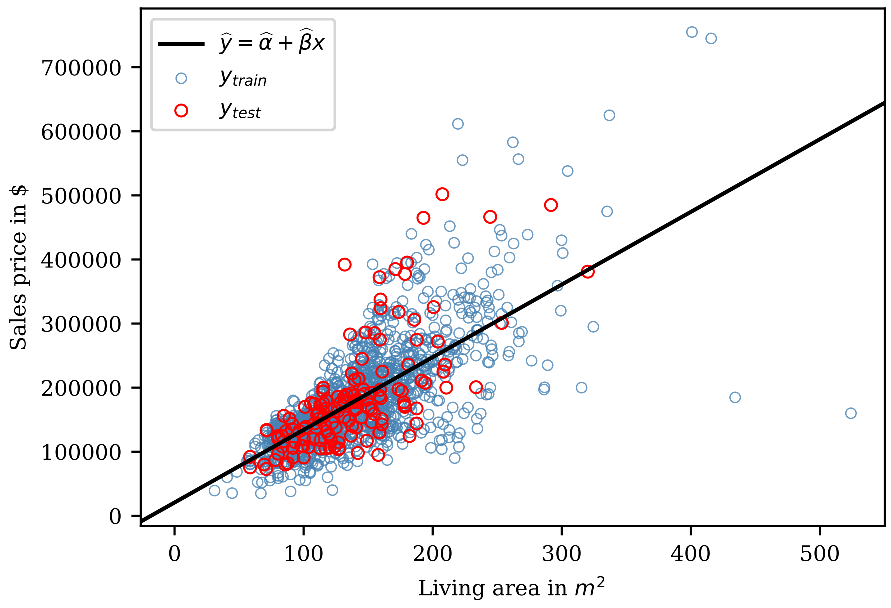
```
Regression coefficients
  Intercept: 20675.0
  Slope: 1133.9
```

The fitted coefficients show that for each additional square meter of living area, the sale price on average increases by $1,134.

The following code creates a scatter plot showing the training and test samples and adds the fitted line.

[10]:
```
intercept = lr.intercept_
slope = lr.coef_[0]

fig, ax = plt.subplots(1, 1, figsize=(5, 3.5))

# Plot regression line
ax.axline((0.0, intercept), slope=slope, lw=1.5, c='black',
    label=r'$\widehat{y} = \widehat{\alpha} + \widehat{\beta} x$')

# Plot training data
ax.scatter(X_train, y_train, s=15, color='none', edgecolor='steelblue',
    alpha=0.8, lw=0.5, label=r'$y_{train}$')

# Plot test data
ax.scatter(X_test, y_test, s=20, color='none', edgecolor='red',
    alpha=1.0, lw=0.75, label=r'$y_{test}$')

# Add annotations
ax.set_xlabel('Living area in $m^2$')
ax.set_ylabel(r'Sales price in $')
ax.legend(loc='best')
```

[10]:
```
<matplotlib.legend.Legend at 0x7fca8d47ef50>
```



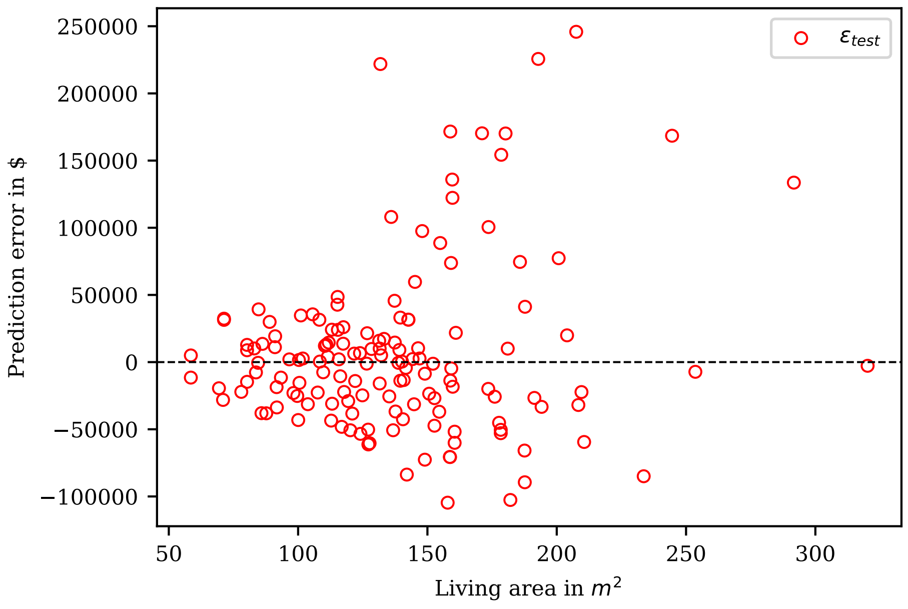
Since this is a univariate model, we can also plot the prediction error against the explanatory variable x. For this, we first need to compute the predicted values in the test sample and then the prediction error for each observations,

$$\epsilon_i = y_i - \widehat{y}_i = y_i - \widehat{\alpha} - \widehat{\beta}x_i$$

for all i that are part of the test sample.

```
[11]: # Predict model for test sample
      y_test_hat = lr.predict(X_test)

      # Prediction error for test sample
      error = y_test - y_test_hat

      fig, ax = plt.subplots(1, 1, figsize=(5, 3.5))

      # Scatter plot of prediction errors
      ax.scatter(X_test, error, s=20, color='none', edgecolor='red',
          alpha=1.0, lw=0.75, label=r'$\epsilon_{test}$')

      # Add annotations
      ax.set_xlabel('Living area in $m^2$')
      ax.set_ylabel(r'Prediction error in $')
      ax.legend(loc='best')
      ax.axhline(0.0, lw=0.75, ls='--', c='black')
```

```
[11]: <matplotlib.lines.Line2D at 0x7fca8d227130>
```



As the graph shows, the errors are reasonably centred around 0 as we would expect from a model that contains an intercept. However, the error variance seems to be increasing in x which indicates that our model might be missing some explanatory variables.

Automatically creating training and test samples

A lot of code was required to create the training and test samples manually. Instead, we can use `scikit-learn`'s `train_test_split()` to considerably simply this task. We need to either specify the `test_size` or `train_size` arguments to determine how the sample should be (randomly) split. The `random_state` argument is used to seed the RNG and get reproducible results.

The following code repeats the sample splitting we performed above using this simpler approach.

[12]:
```python
from sklearn.model_selection import train_test_split

# fraction of data used for test sample
test_size = 0.1

# Split into training / test samples
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=test_size,        # Fraction used for test sample
    random_state=123            # Seed for random number generator
)

# Fit model on training sample
lr = LinearRegression()
lr.fit(X_train, y_train)

print('Regression coefficients')
print(f'  Intercept: {lr.intercept_:.1f}')
print(f'  Slope: {lr.coef_[0]:.1f}')
```

```
Regression coefficients
  Intercept: 21364.5
  Slope: 1125.4
```

Note that the estimated coefficients need not be the same we got when splitting the sample manually. This is because `scikit-learn` uses a different way to randomly allocate the observations to training or test samples, and thus the composition of the training sample will differ.

10.1.3 Evaluating the model fit

One of the commonly used metrics to evaluate models is the *mean squared error (MSE)*, defined as

$$MSE = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - \widehat{y}_i\right)^2$$

which computes the average squared prediction error $y - \widehat{y}$. The magnitude of the MSE is usually hard to interpret, so we often compute the *root mean squared error (RMSE)*,

$$RMSE = \sqrt{MSE} = \left(\frac{1}{N}\sum_{i=1}^{N}\left(y_i - \widehat{y}_i\right)^2\right)^{\frac{1}{2}}$$

which can be interpreted in units of the response variable y. Lastly, another measure of a model's fit is the *coefficients of determination* or R^2, which is a normalized version of the MSE and usually takes on values between $[0, 1]$. The R^2 is defined as

$$R^2 = 1 - \frac{MSE}{\widehat{Var}(y)}$$

where $\widehat{Var}(y)$ is the sample variance of the response y. Intuitively, an R^2 of 1 means that the model predicts the response for each observation perfectly (which is unlikely), whereas an R^2 of 0 implies that the model possesses no explanatory power relative to a model that includes only the sample mean. Note that in a test sample, the R^2 could even be negative.

While these measures are easy to implement ourselves, we can just as well use the functions provided in `scikit-learn.metrics` to do the work for us: `mean_squared_error()` for the MSE and `r2_score()` for the R^2.

[13]:
```python
from sklearn.metrics import mean_squared_error, r2_score

# Compute predicted values for test sample
y_test_hat = lr.predict(X_test)

# Mean squared error (MSE)
mse = mean_squared_error(y_test, y_test_hat)

# Coefficient of determination (R²)
r2 = r2_score(y_test, y_test_hat)

print(f'Mean squared error: {mse:.1f}')
print(f'Root mean squared error {np.sqrt(mse):.1f}')
print(f'Coefficient of determination (R^2): {r2:.2f}')
```

```
Mean squared error: 2821058288.6
Root mean squared error 53113.6
Coefficient of determination (Rˆ2): 0.58
```

Since we estimated a model with intercept, the $R^2 = 0.58$ implies that the model explains 58% of the variance in the test sample.

10.2 Multivariate linear regression

10.2.1 Data with several explanatory variables

Multivariate (or multiple) linear regression extends the simple model to multiple explanatory variables or regressors. To illustrate, we'll load the Ames housing data again but use several (continuous) explanatory variables.

[14]:
```
# Uncomment this to use files in the local data/ directory
DATA_PATH = '../data'

# Load data directly from GitHub (for Google Colab)
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/python-intro-PGR/main/data'
```

[15]:
```
import pandas as pd

# Load data from CSV file
file = f'{DATA_PATH}/ames_houses.csv'
df = pd.read_csv(file)

# Print info about columns contained in DataFrame
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 13 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   SalePrice         1460 non-null   float64
 1   LotArea           1460 non-null   float64
 2   Neighborhood      1460 non-null   object
 3   BuildingType      1386 non-null   object
 4   OverallQuality    1460 non-null   int64
 5   OverallCondition  1460 non-null   int64
 6   YearBuilt         1460 non-null   int64
 7   CentralAir        1460 non-null   object
 8   LivingArea        1460 non-null   float64
 9   Bathrooms         1460 non-null   int64
 10  Bedrooms          1460 non-null   int64
 11  Fireplaces        1460 non-null   int64
 12  HasGarage         1460 non-null   int64
dtypes: float64(3), int64(7), object(3)
memory usage: 148.4+ KB
```

Compared to our earlier analysis, we'll now add the lot area (in m^2) as a feature to the model, which is thus given by

$$SalePrice_i = \alpha + \beta_0 LivingArea_i + \beta_1 LotArea_i + \epsilon_i$$

We first inspect the descriptive statistics of the newly added explanatory variable to get some idea about its distribution.

[16]:
```
df[['SalePrice', 'LivingArea', 'LotArea']].describe()
```

[16]:
```
           SalePrice   LivingArea       LotArea
count    1460.000000  1460.000000   1460.000000
mean   180921.195890   140.777444    976.949947
std     79442.502883    48.813961    927.199358
min     34900.000000    31.026587    120.762165
25%    129975.000000   104.923743    701.674628
50%    163000.000000   135.996777    880.495526
75%    214000.000000   165.049367   1077.709432
max    755000.000000   524.107798  19994.963289
```

As before, we convert the `DataFrame` columns to NumPy arrays, split the data into training and test samples and estimate a linear model.

[17]:
```
features = ['LivingArea', 'LotArea']
target = 'SalePrice'

y = df[target].to_numpy()
```

```
X = df[features].to_numpy()
```

[18]:
```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.3, random_state=123)

lr = LinearRegression()
lr.fit(X_train, y_train)

print(f'Intercept: {lr.intercept_}')
print(f'Coefficients: {lr.coef_}')
```

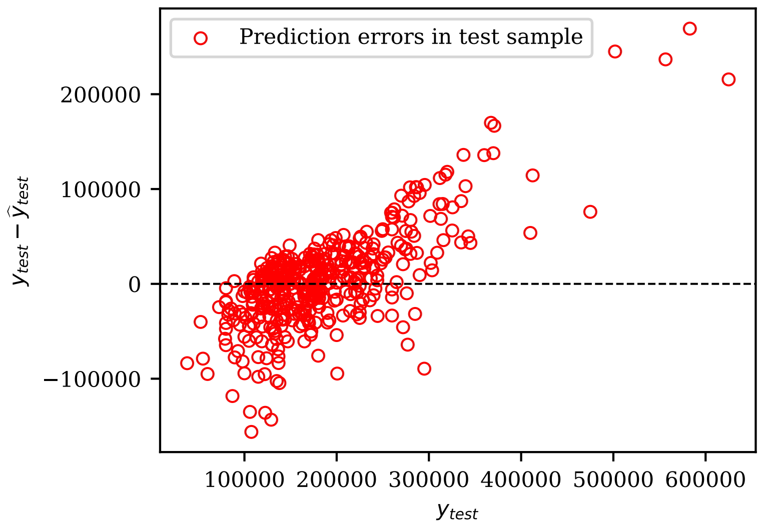
```
Intercept: 20025.027199480886
Coefficients: [1085.68252338    7.09932726]
```

The coefficient array `coef_` now stores two values, the first one for `LivingArea` and the second one for `LotArea`. These coefficients are in the same order as the features in the feature matrix **X** passed when fitting the model.

Since we now have multiple explanatory variables, we can no longer easily plot prediction errors against one feature unless we fix the remaining features at some value or use 3D scatter plots. The latter of course does not help if we have more than two explanatory variables. Instead, we can plot the prediction errors against y which we do below.

[19]:
```
import matplotlib.pyplot as plt

# Predict values in test sample
y_test_hat = lr.predict(X_test)

errors = y_test - y_test_hat

plt.scatter(y_test, errors, s=20, lw=0.75,
    color='none', edgecolor='red',
    label='Prediction errors in test sample'
)
plt.axhline(0.0, lw=0.75, ls='--', c='black')
plt.legend()
plt.xlabel(r'$y_{test}$')
plt.ylabel(r'$y_{test} - \widehat{y}_{test}$')
```

[19]:
```
Text(0, 0.5, '$y_{test} - \\widehat{y}_{test}$')
```



The scatter plot indicates that the prediction errors differ systematically for different levels of y. The model on average overpredicts the sale price for low y (hence the error is negative) and underpredicts the price for large y (hence the error is positive).

10.2.2 Creating polynomial features

Instead of including additional explanatory variables, we can also create additional terms that are functions of variables. The most common way to do this is to include a polynomial in x (or polynomials in multiple explanatory variables). Thus a simple model could be turned into a model with several terms such as

$$y_i = \alpha + \beta_0 x_i + \beta_1 x_i^2 + \beta_2 x_i^3 + \epsilon_i$$

where y is modelled as a cubic polynomial in x. Note that this model is still called *linear* despite the fact that the mapping between x and y is obviously *non-linear*. However, what matters for estimation is that the model is linear in the model parameters $(\alpha, \beta_0, \beta_1, \beta_2)$. Linear models are thus quite flexible since they can include almost arbitrary non-linear transformations of explanatory and response variables.

To illustrate, we extend the previous model which included `LivingArea` and `LotArea` to now include a polynomial of degree 2 in both variables (often the terms "degree" and "order" are used interchangeably, so this might be called a 2nd-order polynomial). Specifically, if we have two variables x and z, such a polynomial would include all terms with exponents summing up to 2 or less:

$$p(x,z) = \beta_0 + \beta_1 x + \beta_2 z + \beta_3 x^2 + \beta_4 x \cdot z + \beta_5 z^2$$

We can use these six terms as explanatory variables in our linear models and estimate the parameters $\beta_0, \dots, \beta_5$.

It would be quite error-prone to create such polynomials ourselves, so we are going to use `scikit-learn`'s `PolynomialFeatures` class to accomplish this tasks. As the name implies, this transformation creates new features that are polynomials of a given input matrix. We can request that the resulting data should include an intercept ("bias") by specifying `include_bias=True`. Note that if an intercept is included in the feature matrix, we should fit the linear model *without* an intercept (by specifying `fit_intercept=False`) as otherwise the model would contain two constant terms.

[20]:
```python
from sklearn.preprocessing import PolynomialFeatures

# Create polynomials of degree 2 or less, including an intercept (bias)
poly = PolynomialFeatures(degree=2, include_bias=True)
Xpoly_train = poly.fit_transform(X_train)

# print polynomial exponents
poly.powers_
```

[20]:
```
array([[0, 0],
       [1, 0],
       [0, 1],
       [2, 0],
       [1, 1],
       [0, 2]])
```

We can use the `powers_` attribute to get a list of exponents for each generated feature. The above output tells us that the first feature was constructed as $\mathbf{X}_{(1)}^0 + \mathbf{X}_{(2)}^0$ since the first row of exponents is `[0, 0]` and is thus the intercept, while the second feature is given by $\mathbf{X}_{(1)}^1 + \mathbf{X}_{(2)}^0 = \mathbf{X}_{(1)}$, where the notation $\mathbf{X}_{(i)}$ refers to the i-th column of the input matrix $\mathbf{X}$.

Now that we have transformed the input matrix $\mathbf{X}$, we can estimate the linear model on the expanded feature matrix as before (adding `fit_intercept=False`):

[21]:
```python
lr = LinearRegression(fit_intercept=False)
lr.fit(Xpoly_train, y_train)
```

```
print(f'Intercept: {lr.intercept_}')
print(f'Coefficients {lr.coef_}')
```

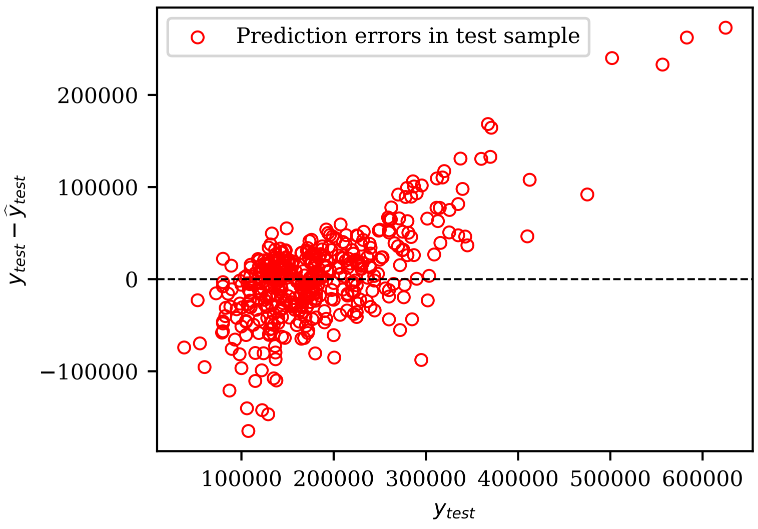
```
Intercept: 0.0
Coefficients [-2.19724359e+04  1.24027918e+03  5.75108671e+01  3.95841742e-01
 -2.46346339e-01 -3.77428502e-04]
```

The fitted model has 6 coefficients and the intercept is 0 as the `LinearRegression` model did not explicitly included one.

As earlier, we can plot the prediction errors as a function of the response variable. Before doing this, it is crucial to also transform the original explanatory variables in the test data set using the same polynomial transformation. We can achieve this by using the `transform()` method of the `poly` object we stored from earlier.

[22]:
```
import matplotlib.pyplot as plt

# Predict values in test sample, applying the same polynomial to
# explanatory variables in test sample.
Xpoly_test = poly.transform(X_test)
y_test_hat = lr.predict(Xpoly_test)

# Compute prediction errors
errors = y_test - y_test_hat

plt.scatter(y_test, errors, s=20, lw=0.75,
    color='none', edgecolor='red',
    label='Prediction errors in test sample'
)
plt.axhline(0.0, lw=0.75, ls='--', c='black')
plt.legend()
plt.xlabel(r'$y_{test}$')
plt.ylabel(r'$y_{test} - \widehat{y}_{test}$')
```

[22]:
```
Text(0, 0.5, '$y_{test} - \\widehat{y}_{test}$')
```



10.2.3 Using scikit-learn pipelines

As you just saw, additional transformation steps before fitting the model can be tedious and error-prone (we might, for example, forget to transform the test data before computing predictions for the test sample). For this reason, `scikit-learn` implements a feature called pipelines which allows us to combine multiple transformations and a final estimation step. For this to work, all steps in the pipeline except for the last must support `fit()` and `transform()` methods, and the final step in the pipeline should be an estimator such as `LinearRegression` (for details, see the section on pipelines in the `scikit-learn` user guide).

There are two ways to construct a pipeline:

1. Create an instance of the `Pipeline` class and specify the steps as name-value pairs.
2. Use the `make_pipeline()` convenience function, which sets a default name for each step.

The first approach requires a list of tuples, where each tuple contains an (arbitrary) name and an object that implements the actions taken at this step. To compose a pipeline that creates polynomial features and fits a linear model to them, we would therefore proceed as follows:

[23]:
```python
from sklearn.pipeline import Pipeline

pipe = Pipeline(steps=[
    ('poly', PolynomialFeatures(degree=2)),
    ('lr', LinearRegression(fit_intercept=False))
])

# visualise pipeline (interactive notebook only)
pipe
```

[23]:
```
Pipeline(steps=[('poly', PolynomialFeatures()),
                ('lr', LinearRegression(fit_intercept=False))])
```

In an interactive notebook, printing the `pipe` object will generate a visualisation which contains details for each step (some editors such as Visual Studio Code even make this visualisation interactive). To transform and fit the model in a single call, we simply need to invoke the `fit()` method. For example, using the training data from above, we run

[24]:
```python
# transform and fit in a single step
pipe.fit(X_train, y_train)

print(f'Coefficients: {pipe.named_steps.lr.coef_}')
```

```
Coefficients: [-2.19724359e+04  1.24027918e+03  5.75108671e+01  3.95841742e-01
 -2.46346339e-01 -3.77428502e-04]
```

The second approach is to construct the pipeline using `make_pipeline()`. For this to work, we only need to pass the objects that constitute the individual steps of a pipeline as follows:

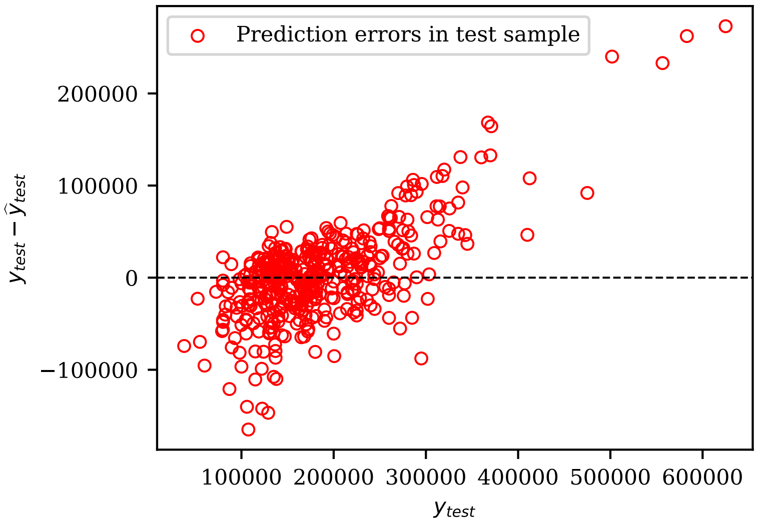
[25]:
```python
from sklearn.pipeline import make_pipeline

pipe = make_pipeline(
    PolynomialFeatures(degree=2),
    LinearRegression(fit_intercept=False)
)

# transform and fit model in single step
pipe.fit(X_train, y_train)

# print default names assigned to each step
pipe.named_steps
```

[25]:
```
{'polynomialfeatures': PolynomialFeatures(),
 'linearregression': LinearRegression(fit_intercept=False)}
```

The function assigns default names to each step which are printed above (basically, these are just lowercase versions of the class defining a step). These names are occasionally required to retrieve information for a specific step. To demonstrate that the pipeline generates results that are equivalent to our manual implementation, we conclude this section by recreating the error plot from above.

[26]:
```
# Predict test responses in a single step. No manual transformation required!
y_test_hat = pipe.predict(X_test)

# Compute prediction errors
errors = y_test - y_test_hat

plt.scatter(y_test, errors, s=20, lw=0.75,
    color='none', edgecolor='red',
    label='Prediction errors in test sample'
)
plt.axhline(0.0, lw=0.75, ls='--', c='black')
plt.legend()
plt.xlabel(r'$y_{test}$')
plt.ylabel(r'$y_{test} - \widehat{y}_{test}$')
```

[26]:
```
Text(0, 0.5, '$y_{test} - \\widehat{y}_{test}$')
```



10.3 Optimising hyperparameters with cross-validation

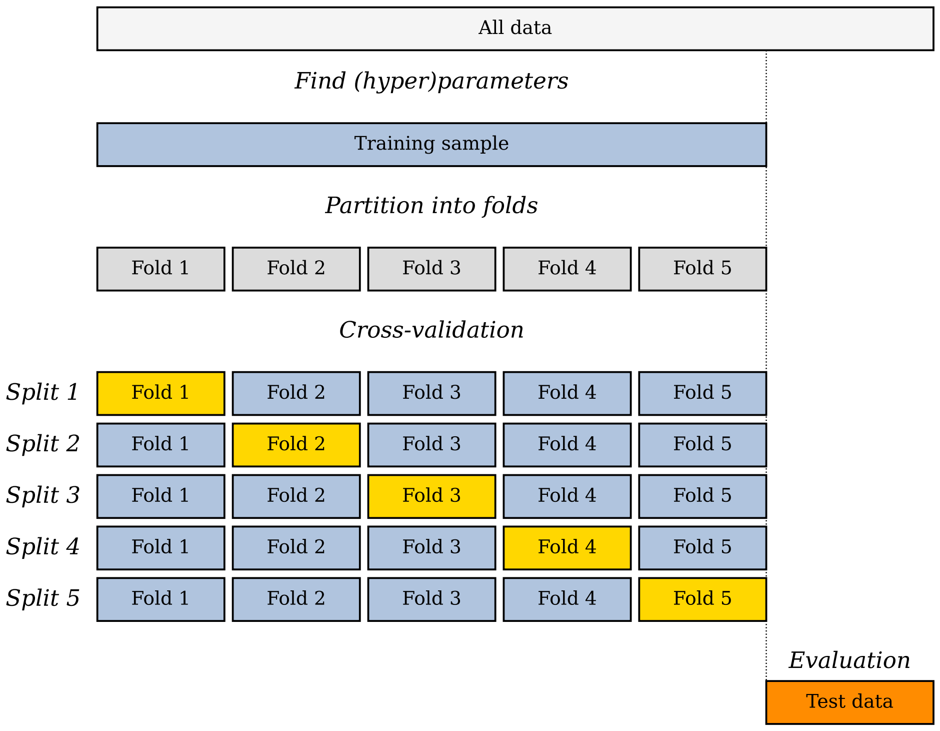
10.3.1 Outline of hyperparameter tuning

Previously, we discussed some ways to evaluate the model fit (MSE and R^2) but did not specify what to to with this information. In this section, we demonstrate how we can use these measures to tune parameters which govern the estimation process, such as the polynomial degree from the previous section. Such parameters are usually called *hyperparameters* to clearly distinguish them from the parameters that are estimated by the model (e.g., the coefficients of a linear model).

Tuning hyperparameters, estimating the model and evaluating its generalisability cannot be done based on the same data set, as these steps then become interdependent (for the same reason we don't want to evaluate the model fit on the training sample as this would lead to overfitting). To this end, in machine learning we often split the data intro *three* parts: the training, validation and test data sets. Models are estimated on the training sample, the choice of hyperparameters is determined based on the validation sample and model generalisability is determined based on the test sample which was not used in estimation or tuning at all.

However, because we might not have enough data to split the sample this way, we usually perform so-called *cross-validation* which is illustrated in the figure below:

1. We split the overall data set into training and test sub-samples.
2. The training data set is further split into K so-called *folds*.
    1. For each $k = 1, \dots, K$, a smaller training sample is formed by excluding the k-th fold and estimating the model on the remaining $K - 1$ folds. We then compute the chosen metric of model fit on the k-th fold and store the result.
    2. After cycling through all K folds, we have K values of our desired metric, which we then average to get our final measure. Hyperparameters are tuned by minimising (or maximising) this averaged metric.
3. Once hyperparameter tuning is complete, we evaluate the model on the test data set.



In this unit we will skip the final step of assessing generalisability on the test data set. Consequently, when running cross-validation, we will only use training/validation data sets and we will use the terms "validation" and "test" interchangeably.

The `scikit-learn` documentation contains a wealth of additional information on cross-validation. Another method to perform hyperparameter tuning is grid search which we won't cover in this unit.

10.3.2 Example: Tuning of the polynomial degree

To illustrate the concept, we demonstrate the procedure outlined above by tuning the polynomial degree for the example covered in the previous section by minimising the root mean squared error (RMSE).

We recreate the data set in the same way as before, using `LivingArea` and `LotArea` as explanatory variables.

[27]:
```python
import pandas as pd

df = pd.read_csv(f'{DATA_PATH}/ames_houses.csv')

features = ['LivingArea', 'LotArea']
target = 'SalePrice'
```

```
y = df[target].to_numpy()
X = df[features].to_numpy()
```

We now iterate over the candidate polynomial degrees $d = 0, \dots, 4$ and apply k-fold cross-validation with 10 folds. There is no need to manually split the sample into training and validation/test sub-sets. Instead, we use the `KFold` class to automatically create the splits for us. Once we have created an instance with the desired number of folds, e.g., `KFold(n_splits=10)`, we can call the `split()` method which iterates trough all possible combinations of training and test data sets and returns the array indices for each.

[28]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error

degrees = np.arange(5)

rmse_mean = []

for d in degrees:

    # Create pipeline to transform and fit the model. Pipeline depends
    # on polynomial degree!
    pipe = Pipeline(steps=[
        ('poly', PolynomialFeatures(degree=d, include_bias=True)),
        ('lr', LinearRegression(fit_intercept=False))
    ])

    rmse_fold = []

    # Create 10 folds
    folds = KFold(n_splits=10)

    # Iterate through all combinations of training/test data
    for itrain, itest in folds.split(X, y):

        # Extract training data for this split
        X_train = X[itrain]
        y_train = y[itrain]

        # Extract test (or validation) data for this split
        X_test = X[itest]
        y_test = y[itest]

        # Fit model on training data for given degree
        pipe.fit(X_train, y_train)

        # Predict response on test data
        y_test_hat = pipe.predict(X_test)

        # Compute RMSE as model fit measure: function returns RMSE
        # if squared=False is passed!
        rmse = mean_squared_error(y_test, y_test_hat, squared=False)

        # Store RMSE for current split
        rmse_fold.append(rmse)

    # Store average MSE for current polynomial degree
    rmse_mean.append(np.mean(rmse_fold))

# Convert to NumPy array
```

```
rmse_mean = np.array(rmse_mean)

# Print average RMSE for all polynomial degrees
rmse_mean
```

```
[28]: array([ 79020.25214052,  55268.49840787,  55228.58262438,  55885.05422781,
              437077.00283378])
```

This code returns an array of 5 averaged RMSEs, one for each $d = 0, \ldots, 4$. We can now find the optimal d by picking the one which has the lowest mean squared error in the test samples using the `argmin()` function which returns the *index* of the smallest array element.

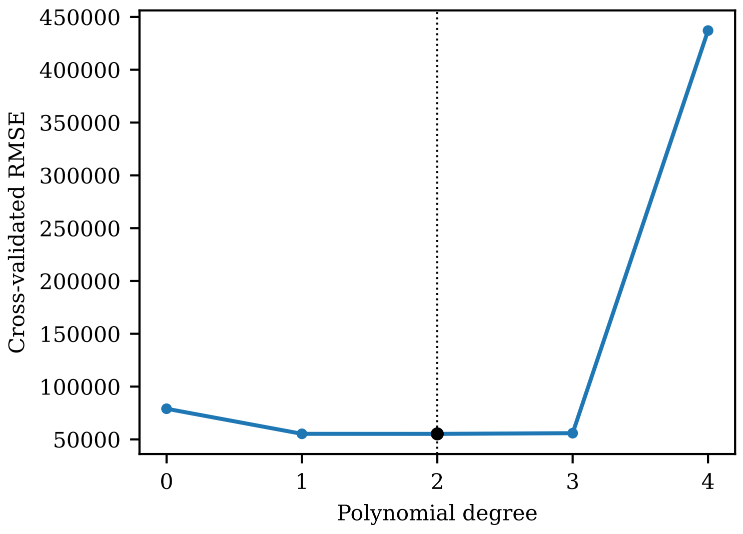
```
[29]: # Find index of polynomial degree with smallest RMSE
imin = np.argmin(rmse_mean)

# RMSE-minimising degree
dmin = degrees[imin]

print(f'Polynomial degree with minimum RMSE: {dmin}')
```

```
Polynomial degree with minimum RMSE: 2
```

Finally, it is often instructive to visualise how the RMSE evolves as a function of the hyperparameter we want to tune.

```
[30]: plt.plot(degrees, rmse_mean, marker='o', ms=3)
plt.xlabel('Polynomial degree')
plt.ylabel('Cross-validated RMSE')
plt.scatter(degrees[imin], rmse_mean[imin], s=15, c='black', zorder=100)
plt.xticks(degrees)
plt.axvline(imin, ls=':', lw=0.75, c='black')
```

```
[30]: <matplotlib.lines.Line2D at 0x7fca84770070>
```



Here we see that for $d = 0$ (the intercept-only model), the model underfits the data leading to a high prediction error. However, higher d's do not always translate into a better fit. For $d = 4$, the model vastly overfits the data, resulting in poor performance in the test sample and a very large RMSE.

Automating cross-validation

The code implemented above to run cross-validation was needlessly complex even though we leveraged the `KFold` class to do the sample splitting for us. Fortunately, `scikit-learn` provides us with even more convenience functions that further simplify this process. For example, if we want to perform tuning based on a single score (such as the root mean squared error), we can instead use `cross_val_score()`. This function requires us to specify an estimator (or a pipeline), the number of folds to use for cross-validation (`cv=10`) and the metric to evaluate. For example, if we want to compute the RMSE, we would pass the argument `scoring='neg_root_mean_squared_error'`. Note that the function returns the *negative* RMSE which we need to correct manually.

See the documentation for a complete list of valid metrics that can be passed as `scoring` arguments. Alternatively, `scikit-learn` lists available metrics by running

```
import sklearn.metrics
sklearn.metrics.get_scorer_names()
```

The following code re-implements the k-fold cross-validation from earlier using `cross_val_score()`:

```
[31]: from sklearn.model_selection import cross_val_score

degrees = np.arange(5)

rmse_mean = []
rmse_std = []

for d in degrees:

    pipe = Pipeline(steps=[
        ('poly', PolynomialFeatures(degree=d, include_bias=True)),
        ('lr', LinearRegression(fit_intercept=False))
    ])

    score = cross_val_score(
        pipe,
        X, y,
        scoring='neg_root_mean_squared_error',
        cv=10
    )

    # Function returns NEGATIVE RMSE, correct this here!
    rmse_mean.append(np.mean(-score))

# Convert to NumPy array
rmse_mean = np.array(rmse_mean)

# Print average RMSE for all polynomial degrees
rmse_mean
```

```
[31]: array([ 79020.25214052,  55268.49840787,  55228.58262438,  55885.05422781,
       437077.00283378])
```

The RMSE-minimising degree is of course the same as before:

```
[32]: imin = np.argmin(rmse_mean)
dmin = degrees[imin]

print(f'Polynomial degree with min. RMSE: {dmin}')
```

```
Polynomial degree with min. RMSE: 2
```

10.4 Linear models with regularisation: Ridge regression

The linear models we encountered so far are part of the standard econometrics toolbox. In this section, we look at extensions that make these models more useful for machine learning applications.

The first model we study is Ridge regression, which estimates a linear model but adds a penalty term to the loss function which is now given by

$$L(\mu, \boldsymbol{\beta}) = \underbrace{\sum_{i=1}^{N} \Big(y_i - \mu - \mathbf{x}_i' \boldsymbol{\beta}\Big)^2}_{\text{Sum of squared errors}} + \underbrace{\alpha \sum_{k=1}^{K} \beta_k^2}_{\text{L2 penalty}}$$

Compared to ordinary least squares (OLS) we discussed initially, the penalty term increases the loss function if the estimated coefficients $\boldsymbol{\beta}$ are large. This term is called an L2 penalty because it corresponds to the (squared) L2 vector norm. In many textbooks, you will see the penalty term written as $\alpha\|\boldsymbol{\beta}\|_2^2$ which is equivalent to the formula used above.

For any $\alpha > 0$, the loss function is increasing in the (absolute) coefficient values, thus the minimum L might be one where the elements of $\boldsymbol{\beta}$ are smaller than what they would have been with OLS. We therefore say that that Ridge regression applies *regularisation* or *shrinkage*. Note, however, that the intercept which we now denote by μ is not included in the penalty term, and thus no regularisation is applied to it.

Clearly, the regularisation strength depends on the value of α. For tiny (or zero) α, the estimated $\widehat{\boldsymbol{\beta}}$ will be close (or identical) to OLS, while for large α the estimated coefficients will be compressed towards zero. In this setting, α is a hyperparameter and we can accordingly use cross-validation to find an "optimal" value.

Why would we ever want to use Ridge regression given that OLS is the best linear unbiased estimator? It turns out that regularisation can help in scenarios where we have a large number of (potentially multicollinear) regressors in which case OLS is prone to overfitting.

10.4.1 Example: Polynomial approximation

We illustrate such problems and the benefits of Ridge regression using a highly stylised example. Imagine we want to approximate a non-linear function using a high-order polynomial, a setting which is notoriously susceptible to overfitting (also see the optional exercises for more illustrations). Assume that our model is given by

$$y_i = \cos\left(\frac{3}{2}\pi x_i\right) + \epsilon_i$$
$$\epsilon \overset{\text{iid}}{\sim} N(0, 0.25)$$

where ϵ_i as an additive, normally-distributed measurement error term with mean 0 and variance $\frac{1}{4}$.

The true values of y (without measurement error) are computed using the function `compute_true_y()` which returns y for a given x.

[33]:
```
import numpy as np

# True function (w/o errors)
def compute_true_y(x):
    return np.cos(1.5 * np.pi * x)
```

The following code creates a demo sample with $N = 100$ observations.

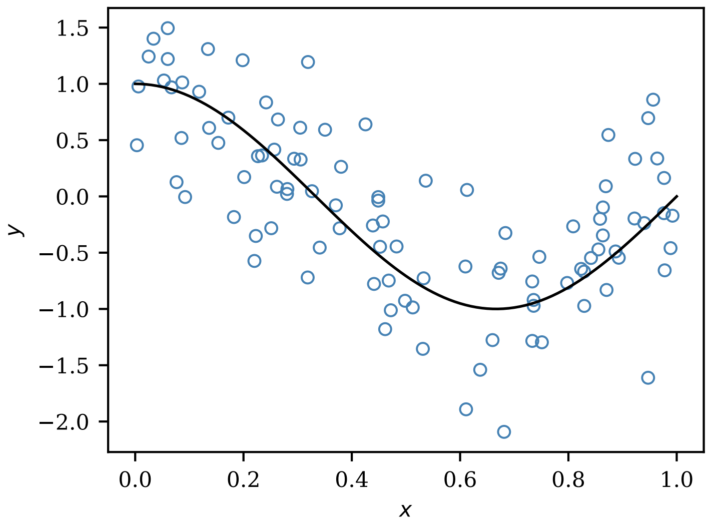
[34]:
```
from numpy.random import default_rng

# Initialise random number generator
rng = default_rng(1234)
```

```
# Sample size
N = 100

# Randomly draw explanatory variable x uniformly distributed on [0, 1]
x = rng.random(size=N)

# Draw errors from normal distribution
epsilon = rng.normal(scale=0.5, size=N)

# Compute y, add measurement error
y = compute_true_y(x) + epsilon
```

The next graph visualises the true relationship and the sample points (x_i, y_i).

[35]:
```
import matplotlib.pyplot as plt

# Sample scatter plot
plt.scatter(x, y, s=20, color='none', edgecolor='steelblue',
    lw=0.75, label='Sample')

# Plot true relationship
xvalues = np.linspace(0.0, 1.0, 101)
plt.plot(xvalues, compute_true_y(xvalues), color='black', lw=1.0,
    label='True function')

plt.xlabel('$x$')
plt.ylabel('$y$')
```

[35]:
```
Text(0, 0.5, '$y$')
```



Estimating the Ridge and linear regression models

We now estimate a polynomial approximation where we assume that $y$ is a degree-K polynomial in $x$, i.e.,

$$y_i \approx \mu + \beta_1 x_i + \beta_2 x_i^2 + \cdots + \beta_K x_i^K$$

For this example, we choose an unconventionally high $K = 15$ since we anticipate that this leads to problems with OLS.

A few more steps are required before we can run the Ridge regression:

- We need to create the polynomial in x which we do with the `PolynomialFeatures` transformation we already encountered.
- Moreover, regularisation methods can be susceptible to scaling issues, so we need to demean and normalise all input variables, i.e., we make sure that each feature has mean 0 and a variance of 1. We can automate this step using the `StandardScaler` transformation.
- Finally, the estimation step is performed by the `Ridge` class.
- We build a `Pipeline` that combines these transformation together with the estimation step. We use the function `make_pipeline()` to simplify this step.

To run the Ridge regression, we moreover need to specify the regularisation strength α which we set to $\alpha = 3$ for illustration.

[36]:
```
from sklearn.linear_model import Ridge
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.pipeline import make_pipeline

# Polynomial degree
degree = 15

# Build pipeline of transformations and Ridge estimation.
# We create the polynomial transformation w/o the intercept so we
# need to include an intercept in the fitting step.
pipe_ridge = make_pipeline(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    StandardScaler(),                           # standardise features
    Ridge(alpha=3.0, fit_intercept=True)        # Fit Ridge regression
)

# Make sure X is a matrix
X = x[:, None]

# Fit model
pipe_ridge.fit(X, y)
```

[36]:
```
Pipeline(steps=[('polynomialfeatures',
                 PolynomialFeatures(degree=15, include_bias=False)),
                ('standardscaler', StandardScaler()),
                ('ridge', Ridge(alpha=3.0))])
```

It is instructive to estimate the same model using linear regression and compare the results:

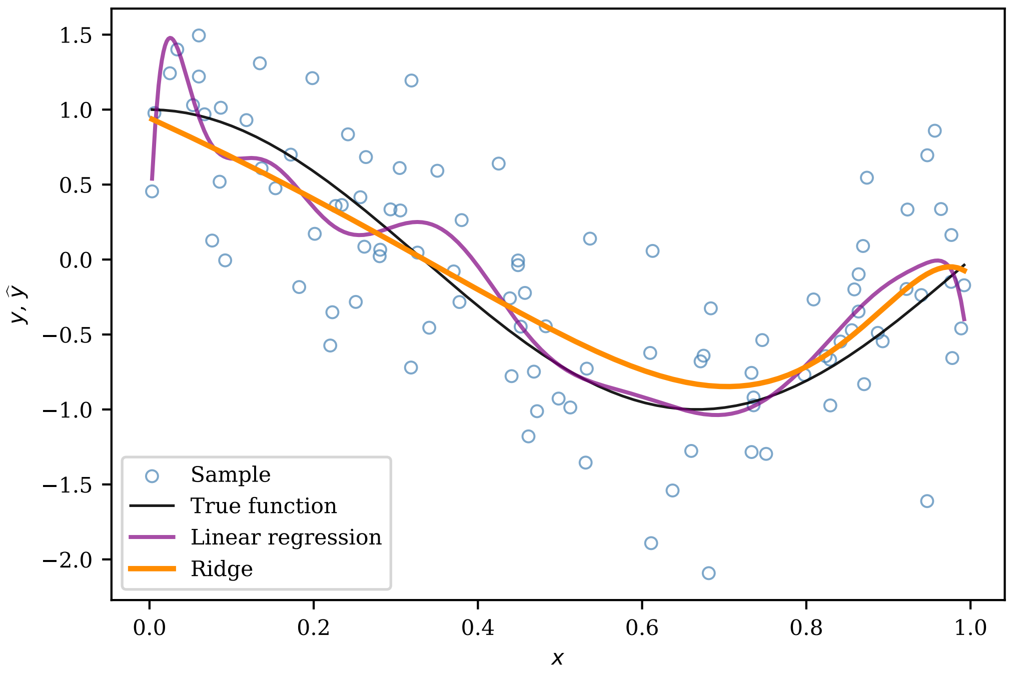
[37]:
```
from sklearn.linear_model import LinearRegression

# Create pipeline for linear model (linear regression does not require
# standardisation!)
pipe_lr = make_pipeline(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    LinearRegression(fit_intercept=True)
)

# Make sure X is a matrix
X = x[:, None]

# Fit model
pipe_lr.fit(X, y)
```

```
[37]: Pipeline(steps=[('polynomialfeatures',
                       PolynomialFeatures(degree=15, include_bias=False)),
                      ('linearregression', LinearRegression())])
```

To illustrate the difference for this artificial example, we add the model predictions from the Ridge and linear regressions to the sample scatter plot we created earlier.

```
[38]: import matplotlib.pyplot as plt

      fig, ax = plt.subplots(1, 1, figsize=(6, 4))

      # Grid on which to evaluate true model and predictions
      xvalues = np.linspace(np.amin(x), np.amax(x), 500)

      # Sample points scatter plot
      ax.scatter(x, y, s=20, color='none', edgecolor='steelblue',
          lw=0.75, alpha=0.7, label='Sample')

      # True model
      ax.plot(xvalues, compute_true_y(xvalues), color='black', lw=1.0, alpha=0.9,
          label='True function')

      # LR prediction
      ax.plot(xvalues, pipe_lr.predict(xvalues[:, None]), color='purple',
          alpha=0.7, label='Linear regression')

      # Ridge prediction
      ax.plot(xvalues, pipe_ridge.predict(xvalues[:, None]),
          color='darkorange', lw=2.0,
          label='Ridge'
      )
      ax.legend()
      ax.set_xlabel('$x$')
      ax.set_ylabel('$y$, $\widehat{y}$')
```

```
[38]: Text(0, 0.5, '$y$, $\\widehat{y}$')
```



As the graph shows, the OLS model vastly overfits the data which is what we would expect in this settings. Conversely, the prediction of the Ridge regression is reasonably close to the true function and much better behaved despite the high polynomial degree used here.

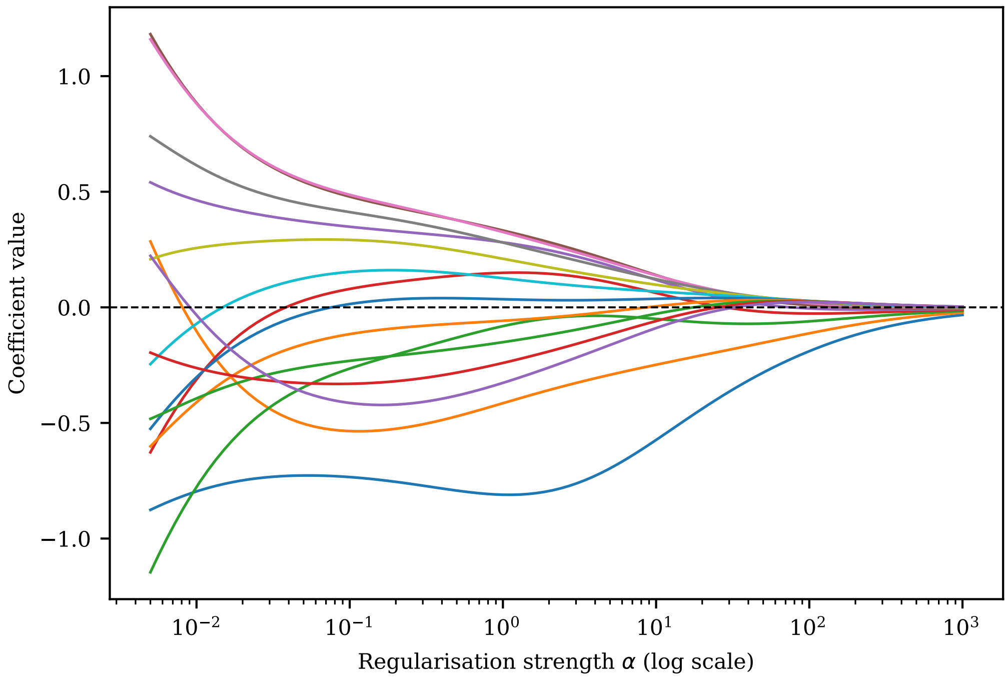
To gain some intuition for what is going on, it is instructive to plot the estimated Ridge coefficients for a whole range of $\alpha$ values which we do in the code below. Note that we choose the grid of $\alpha$ to be uniformly spaced in logs since we want to zoom in on what happens when $\alpha$ is small. This is accomplished by using `np.logspace()` instead of `np.linspace()`.

[39]:
```
# Create grid of alphas spaced uniformly in logs
alphas = np.logspace(start=np.log10(5.0e-3), stop=np.log10(1000.0), num=100)

# Re-create pipeline w/o Ridge estimator, estimation step differs for each alpha
transform = make_pipeline(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    StandardScaler()
)

# Create polynomial features
Xtrans = transform.fit_transform(x[:, None])

# Array to store coefficients for all alphas
coefs = np.empty((len(alphas), Xtrans.shape[1]))

# Estimate Ridge for each alpha, store fitted coefficients
for i, alpha in enumerate(alphas):
    ridge = Ridge(alpha=alpha, fit_intercept=True)

    # Fit model for given alpha
    ridge.fit(Xtrans, y)

    coefs[i] = ridge.coef_
```

The following plot shows each coefficient (one color corresponds to one coefficients) for different values of $\alpha$ on the x-axis. Note that the x-axis is plotted on a $\log_{10}$ scale which allows us to zoom in on smaller values of $\alpha$.

[40]:
```
plt.figure(figsize=(6,4))
plt.plot(alphas, coefs, lw=1.0)
plt.xscale('log', base=10)
plt.axhline(0.0, ls='--', lw=0.75, c='black')
plt.xlabel(r'Regularisation strength $\alpha$ (log scale)')
plt.ylabel('Coefficient value')
```

[40]:
```
Text(0, 0.5, 'Coefficient value')
```



For small α (on the left) the estimated coefficients are close to the (standardised) OLS coefficients, but their values shrink towards zero as α increases. For a very large $\alpha = 10^3$ the estimated coefficients are basically zero since the penalty dominates the sum of squared errors in the loss function.

10.4.2 Tuning the regularisation parameter via cross-validation

In the previous example, we picked an arbitrary regularisation strength α when fitting the Ridge regression. In applied work, we would want to tune α (which is a hyperparameter) using cross-validation instead. To this end, we could use the generic cross-validation functionality we studied earlier in this unit since that one works for all types of estimators. However, `scikit-learn` implements a cross-validation class specifically for Ridge regression called `RidgeCV` which we use in the code below to find an optimal α.

[41]:
```
from sklearn.linear_model import RidgeCV

# RidgeCV does not support pipelines, so we need to transform x before
# cross-validation.
transform = make_pipeline(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    StandardScaler()
)

# Create standardised polynomial features
Xtrans = transform.fit_transform(x[:, None])

# Set of candidate alphas on [1.0e-5, 5] to cross-validate.
# Spaced uniformly in logs to get denser grid for small alphas.
alphas = np.logspace(start=np.log10(1.0e-5), stop=np.log10(5), num=100)

# Create and run Ridge cross-validation
rcv = RidgeCV(alphas=alphas, store_cv_values=True).fit(Xtrans, y)
```

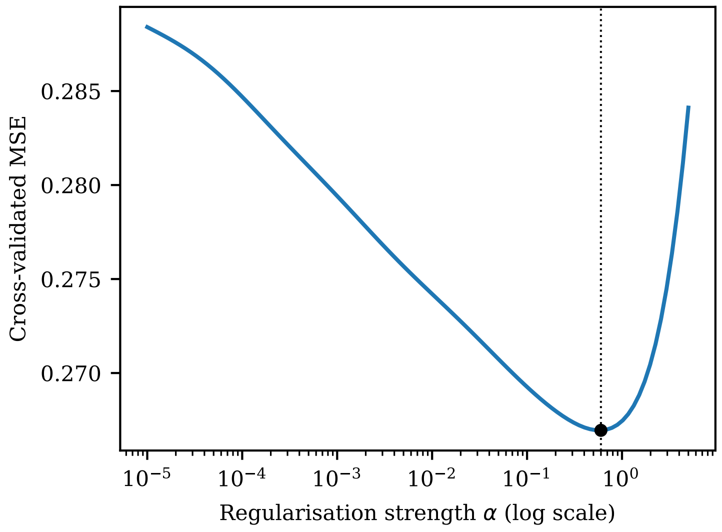
By default, RidgeCV uses the (negative) mean squared error (MSE) to find the best $\alpha$ which we can then recover from the `alpha_` attribute.

[42]:
```
# Recover best alpha that minimizes MSE
alpha_best = rcv.alpha_

# Best MSE is stored as negative score!
MSE_best = - rcv.best_score_

print(f'Best alpha: {alpha_best:.3g} (MSE: {MSE_best:.3g})')
```

```
Best alpha: 0.6 (MSE: 0.267)
```

Because we fitted `RidgeCV` with the argument `store_cv_values=True`, the fitted object stores the MSE for each CV split in the attribute `cv_values_` which we can use to plot the MSE as a function of $\alpha$.

[43]:
```
import matplotlib.pyplot as plt

# Compute average MSE for each alpha value across all folds
mse_mean = np.mean(rcv.cv_values_, axis=0)

# Index of MSE-minimising alpha
imin = np.argmin(mse_mean)

# Plot MSE against alphas, highlight minimum MSE
plt.plot(alphas, mse_mean)
plt.xlabel(r'Regularisation strength $\alpha$ (log scale)')
plt.ylabel('Cross-validated MSE')
plt.scatter(alphas[imin], mse_mean[imin], s=15, c='black', zorder=100)
plt.axvline(alphas[imin], ls=':', lw=0.75, c='black')
plt.xscale('log')
```



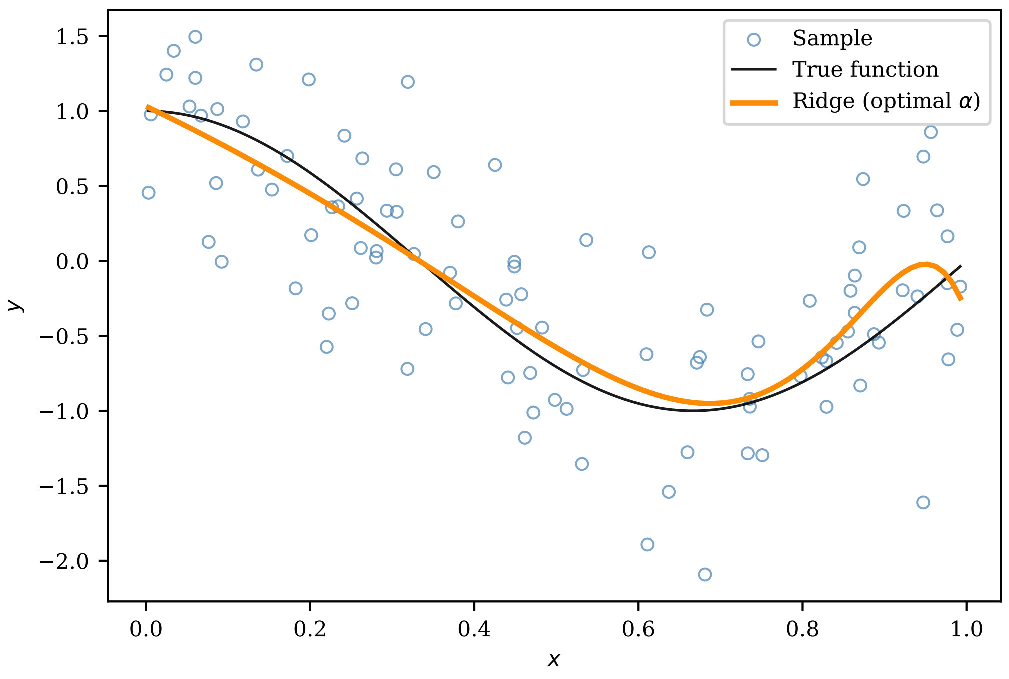
Now that we have identified the optimal $\alpha$, we can re-fit the Ridge regression and plot the prediction from this model. Note that this is not strictly necessary as the return value of `RidgeCV` can be used to do prediction based on coefficients estimated for the best $\alpha$, but because `RidgeCV` does not support pipelines, we'd have to apply any transformations manually before doing so.

[44]:
```
# Create pipeline with Ridge estimator using optimal alpha
pipe_ridge = make_pipeline(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    StandardScaler(),
    Ridge(alpha=alpha_best, fit_intercept=True)
)

# Fit Ridge regression with optimal alpha
pipe_ridge.fit(x[:, None], y)
```

```
[44]: Pipeline(steps=[('polynomialfeatures',
                        PolynomialFeatures(degree=15, include_bias=False)),
                       ('standardscaler', StandardScaler()),
                       ('ridge', Ridge(alpha=0.599686603363296))])
```

```
[45]: # Plot sample scatter, true model and Ridge prediction using optimal alpha
fig, ax = plt.subplots(1, 1, figsize=(6, 4))

# Sample scatter plot
ax.scatter(x, y, s=20, color='none', edgecolor='steelblue',
    lw=0.75, alpha=0.7, label='Sample')

xvalues = np.linspace(np.amin(x), np.amax(x), 100)
# Plot true relationship
ax.plot(xvalues, compute_true_y(xvalues), color='black', lw=1.0, alpha=0.9,
    label='True function')

# Plot model prediction
ax.plot(xvalues, pipe_ridge.predict(xvalues[:, None]), c='darkorange',
    lw=2.0, label=r'Ridge (optimal $\alpha$)')

ax.legend()
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
```

```
[45]: Text(0, 0.5, '$y$')
```



10.5 Linear models with regularisation: Lasso

Another widely used estimator with shrinkage is LASSO (least absolute shrinkage and selection operator) which adds an L1 penalty term to the objective function:

$$L(\mu, \boldsymbol{\beta}) = \underbrace{\sum_{i=1}^{N} \Big(y_i - \mu - \mathbf{x}_i' \boldsymbol{\beta} \Big)^2}_{\text{Sum of squared errors}} + \alpha \underbrace{\sum_{k=1}^{K} |\beta_k|}_{\text{L1 penalty}}$$

As with the Ridge regression, this additional term penalises large coefficient values. This term is called an L1 penalty because it corresponds to the L1 vector norm which can equivalently be written as $\alpha \|\boldsymbol{\beta}\|_1$.

While the objective looks very similar to Ridge regression, using the L1 instead of the L2 norm can produce much more parsimonious models because many coefficients end up being exactly zero and the corresponding features are thus eliminated from the model. We will see this in the example below.

10.5.1 Example: Polynomial approximation

We apply Lasso to the same random sample as in the section on Ridge which allows us to compare the two methods. The following code recreates that data, making the same functional form and distributional assumptions as in the previous section.

[46]:
```python
import numpy as np

# True function (w/o errors)
def compute_true_y(x):
    return np.cos(1.5 * np.pi * x)
```

[47]:
```python
from numpy.random import default_rng

# Initialise random number generator
rng = default_rng(1234)

# Sample size
N = 100

# Randomly draw explanatory variable x uniformly distributed on [0, 1]
x = rng.random(size=N)

# Draw errors from normal distribution
epsilon = rng.normal(scale=0.5, size=N)

# Compute y, add measurement error
y = compute_true_y(x) + epsilon
```

Estimating the Lasso and linear regression models

As with Ridge, we need to standardise the explanatory variables before fitting Lasso. The code below repeats these steps, but we now use `Lasso` to perform the model estimation. For now, we set the regularisation strength to $\alpha = 0.015$ and will use cross-validation to find the optimal value later.

Note that for Lasso it might be necessary to increase the number of iterations by increasing the `max_iter` parameter (from the default of 1,000).

[48]:
```python
from sklearn.linear_model import Lasso
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.pipeline import make_pipeline
```

```
# Polynomial degree
degree = 15

# Build pipeline of transformations and Lasso estimation.
# We create the polynomial transformation w/o the intercept so we
# need to include an intercept in the fitting step.
pipe_lasso = make_pipeline(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    StandardScaler(),
    Lasso(alpha=0.015, fit_intercept=True, max_iter=10000)
)

# Make sure X is a matrix
X = x[:, None]

pipe_lasso.fit(X, y)
```

[48]:
```
Pipeline(steps=[('polynomialfeatures',
                 PolynomialFeatures(degree=15, include_bias=False)),
                ('standardscaler', StandardScaler()),
                ('lasso', Lasso(alpha=0.015, max_iter=10000))])
```

For completeness, let’s also recreate the linear regression estimation and plot the model predictions alongside the Lasso.

[49]:
```
from sklearn.linear_model import LinearRegression

# Create pipeline for linear model (linear regression does not require
# standardisation!)
pipe_lr = make_pipeline(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    LinearRegression(fit_intercept=True)
)

# Make sure X is a matrix
X = x[:, None]

pipe_lr.fit(X, y)
```

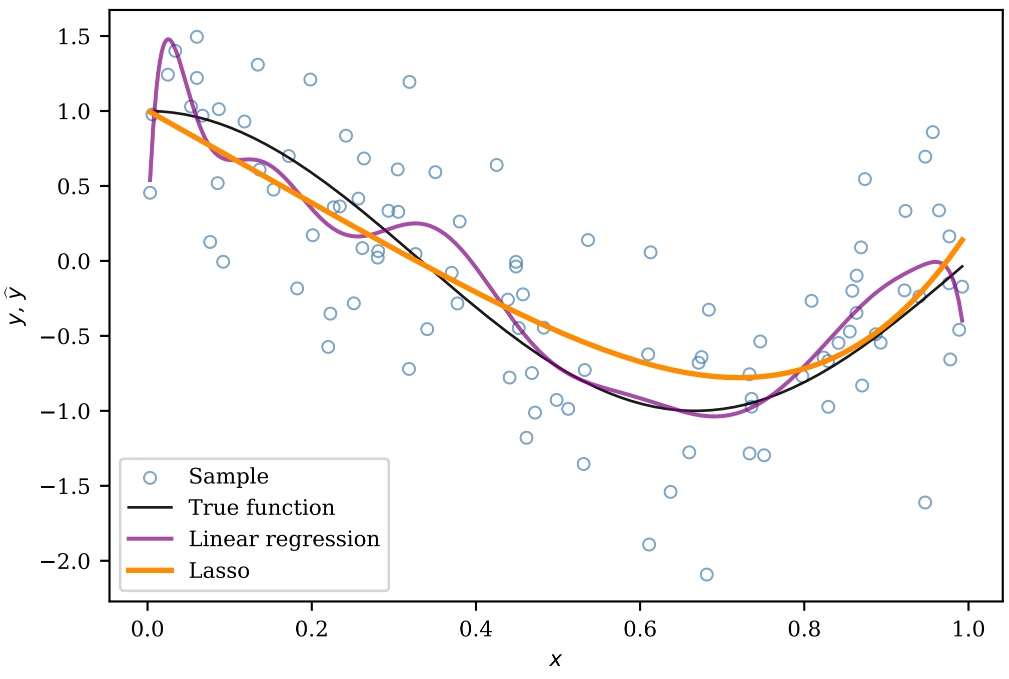
[49]:
```
Pipeline(steps=[('polynomialfeatures',
                 PolynomialFeatures(degree=15, include_bias=False)),
                ('linearregression', LinearRegression())])
```

The following code plots the sample data, the true functional relationship, and the predictions from the linear regression and Lasso models.

[50]:
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 1, figsize=(6, 4))

# Sample points scatter plot
ax.scatter(x, y, s=20, color='none', edgecolor='steelblue',
    lw=0.75, alpha=0.7, label='Sample')

# Grid on which to evaluate true model and predictions
xvalues = np.linspace(np.amin(x), np.amax(x), 500)
```

```
# True model
ax.plot(xvalues, compute_true_y(xvalues),
    color='black', lw=1.0, alpha=0.9,
    label='True function'
)

# Linear regression prediction
ax.plot(xvalues, pipe_lr.predict(xvalues[:, None]),
    color='purple', alpha=0.7,
    label='Linear regression'
)

# Lasso prediction
ax.plot(xvalues, pipe_lasso.predict(xvalues[:, None]),
    color='darkorange', lw=2.0,
    label='Lasso'
)

ax.legend()
ax.set_xlabel('$x$')
ax.set_ylabel('$y$, $\widehat{y}$')
```

```
[50]: Text(0, 0.5, '$y$, $\\widehat{y}$')
```



You might be wondering why we chose $\alpha = 0.015$ whereas we initially used $\alpha = 3$ for the Ridge regression. The reason is that the `scikit-learn` implementation of Lasso uses a slightly different loss function than the one given above, namely

$$L(\mu, \boldsymbol{\beta}) = \frac{1}{2N} \sum_{i=1}^{N} \Bigl(y_i - \mu - \mathbf{x}_i' \boldsymbol{\beta} \Bigr)^2 + \alpha \|\boldsymbol{\beta}\|_1$$

which scales the sum of squared errors term by a factor of of $(2N)^{-1}$. This makes no difference for the optimisation, but changes the interpretation of the regularisation strength α compared to Ridge regression. For our sample size of $N = 100$, an α_{Ridge} used for the `Ridge` estimator approximately corresponds to $\alpha_{Lasso} = \frac{\alpha_{Ridge}}{200}$ when plugged into `Lasso`.

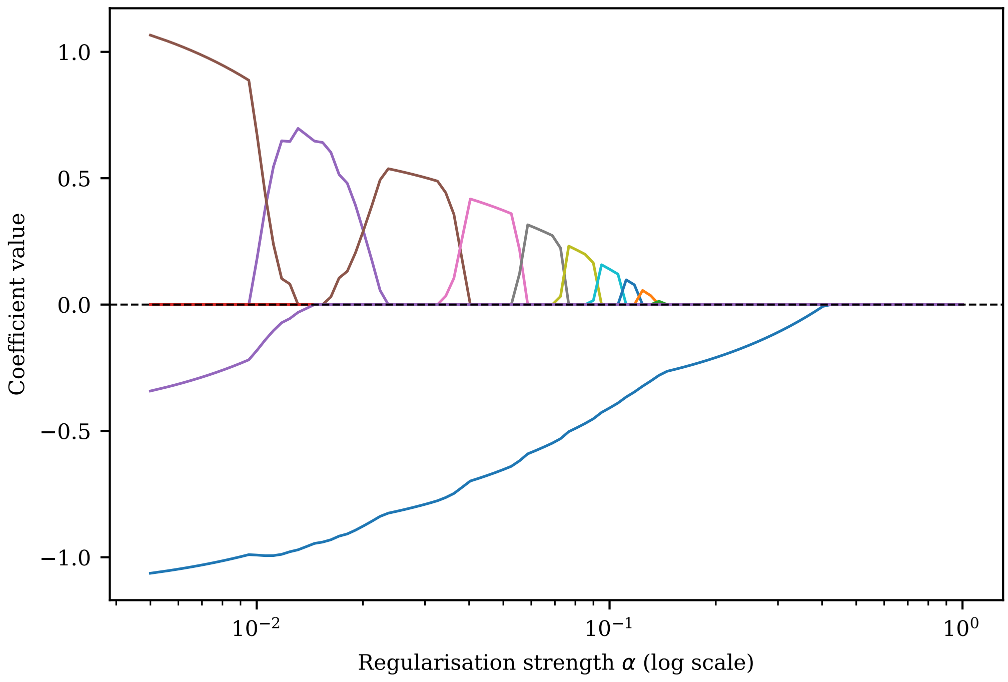
We next fit the model for different values of $\alpha$ on the interval $[5 \times 10^{-3}, 1]$ where we again space the $\alpha$ uniformly in logs.

[51]:
```
# Create grid of alphas spaced uniformly in logs
alphas = np.logspace(start=np.log10(5.0e-3), stop=np.log10(1.0), num=100)

# Re-create pipeline w/o Lasso estimator, estimation step differs for each alpha
transform = make_pipeline(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    StandardScaler()
)

# Create polynomial features
Xtrans = transform.fit_transform(x[:, None])

# Array to store coefficients for all alphas
coefs = np.empty((len(alphas), Xtrans.shape[1]))

# Estimate Lasso for each alpha, store fitted coefficients
for i, alpha in enumerate(alphas):
    lasso = Lasso(alpha=alpha,
        fit_intercept=True,
        max_iter=10000
    )

    # Fit model for given alpha
    lasso.fit(Xtrans, y)

    coefs[i] = lasso.coef_
```

Plotting the fitted coefficients against $\alpha$ on a log scale looks as follows:

[52]:
```
plt.figure(figsize=(6, 4))
plt.plot(alphas, coefs, lw=1.0)
plt.xscale('log', base=10)
plt.axhline(0.0, ls='--', lw=0.75, c='black')
plt.xlabel(r'Regularisation strength $\alpha$ (log scale)')
plt.ylabel('Coefficient value')
```

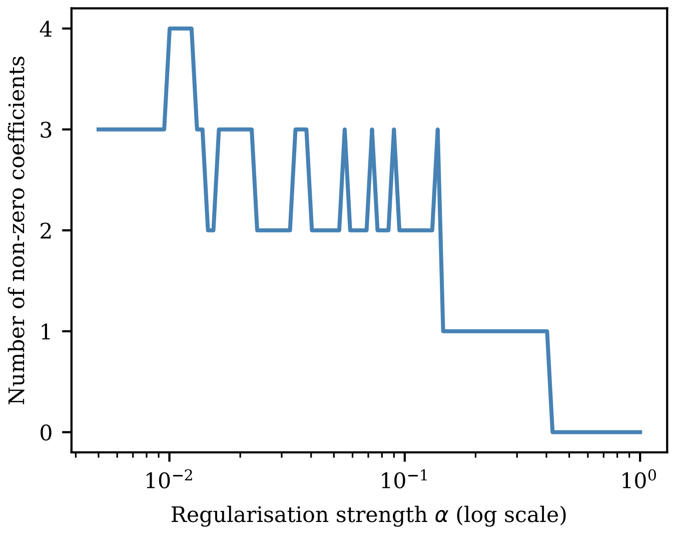
[52]:
```
Text(0, 0.5, 'Coefficient value')
```



The graph shows that the coefficient estimates are quite different than what we obtained from the Ridge estimator. In fact, most of them are exactly zero for most values of α. We highlight this result in the graph below which plots the number of non-zero coefficients against α.

```
[53]: # Number of non-zero coefficients for each alpha.
      nonzero = np.sum(np.abs(coefs) > 1.0e-6, axis=1).astype(int)

      plt.plot(alphas, nonzero, lw=1.5, c='steelblue')
      plt.xscale('log', base=10)
      plt.yticks(np.arange(0, np.amax(nonzero) + 1))
      plt.xlabel(r'Regularisation strength $\alpha$ (log scale)')
      plt.ylabel('Number of non-zero coefficients')
```

```
[53]: Text(0, 0.5, 'Number of non-zero coefficients')
```



Clearly, in this case the model estimated by Lasso is substantially less complex than the linear regression or even Ridge regression. For most values of α, only 2-3 features out of the original 15 are retained in the model!

10.5.2 Tuning the regularisation parameter via cross-validation

In the previous example, we picked an arbitrary regularisation strength $\alpha$ when fitting the Lasso. In this section, we again find an optimal $\alpha$ using cross-validation. Just like in the case of Ridge regression, `scikit-learn` implements a cross-validation class specifically for Lasso called `LassoCV` which we use in the code below to find an optimal $\alpha$.

`LassoCV` optionally accepts a grid of candidate $\alpha$ just like `RidgeCV`, but the default way to run cross-validation is to specify the fraction $\epsilon = \frac{\alpha_{min}}{\alpha_{max}}$ (default: 10^{-3}) and the grid size (default: 100). `LassoCV` then automatically determines an appropriate grid which is stored in the `alphas_` attribute once fitting is complete. Moreover, we use the `cv=...` argument to set the desired number of CV folds (default: 5).

[54]:
```
from sklearn.linear_model import LassoCV

# LassoCV does not support pipelines, so we need to transform x before
# cross-validation.
transform = make_pipeline(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    StandardScaler()
)

# Create standardised polynomial features
Xtrans = transform.fit_transform(x[:, None])

# Create and run Lasso cross-validation, use defaults for eps and n_alphas
lcv = LassoCV(max_iter=100000, cv=10).fit(Xtrans, y)
```

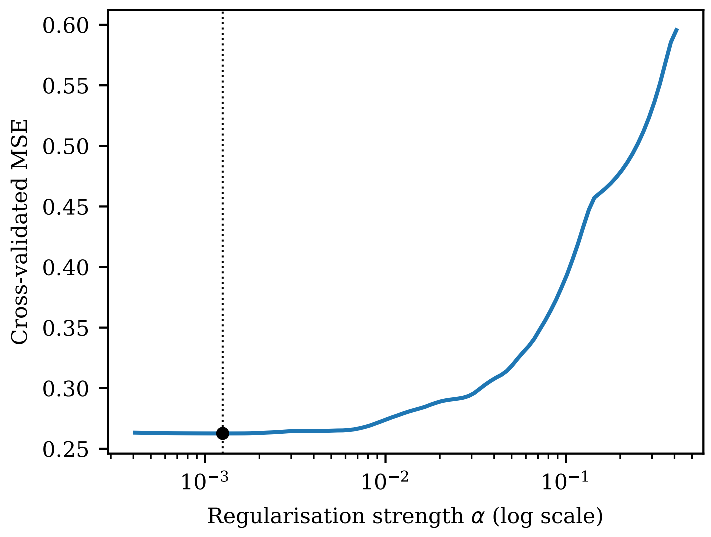
After fitting, we can recover the best alpha from the `alpha_` attribute and the MSE for each $\alpha$ on the grid and each CV fold from the `mse_path_` attribute.

[55]:
```
# Recover best alpha that minimizes MSE
alpha_best = lcv.alpha_

# MSE for each alpha, averaged over folds
mse_mean = np.mean(lcv.mse_path_, axis=1)

# Index of min. MSE
imin = np.argmin(mse_mean)

mse_best = mse_mean[imin]

print(f'Best alpha: {alpha_best:.4g} (MSE: {mse_best:.3g})')
```

```
Best alpha: 0.001251 (MSE: 0.263)
```

The next plot visualises the average MSE over the entire range of candidate $\alpha$ values.

[56]:
```
import matplotlib.pyplot as plt

# Recover grid of alphas used for CV
alphas = lcv.alphas_

# Plot MSE against alphas, highlight minimum MSE
plt.plot(alphas, mse_mean)
plt.xlabel(r'Regularisation strength $\alpha$ (log scale)')
plt.ylabel('Cross-validated MSE')
plt.scatter(alphas[imin], mse_mean[imin], s=15, c='black', zorder=100)
plt.axvline(alphas[imin], ls=':', lw=0.75, c='black')
plt.xscale('log')
```



Now that we have identified the optimal $\alpha$, we can re-fit the Lasso and plot the prediction from this model.

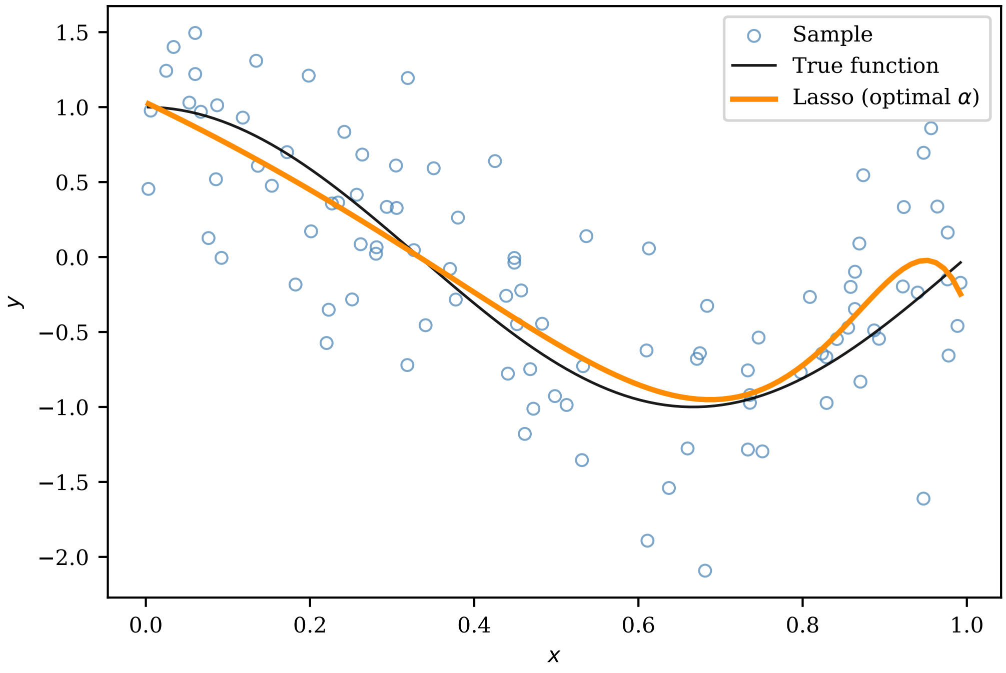
[57]:
```
# Create pipeline with Lasso using optimal alpha
pipe_lasso = make_pipeline(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    StandardScaler(),
    Lasso(alpha=alpha_best, fit_intercept=True, max_iter=100000)
)

pipe_lasso.fit(x[:, None], y)
```

[57]:
```
Pipeline(steps=[('polynomialfeatures',
                 PolynomialFeatures(degree=15, include_bias=False)),
                ('standardscaler', StandardScaler()),
                ('lasso', Lasso(alpha=0.001250626984419745, max_iter=100000))])
```

Finally, we visually compare the true model to the Lasso prediction using the optimal value of $\alpha$.

[58]:
```
# Plot sample scatter, true model and Lasso prediction using optimal alpha
fig, ax = plt.subplots(1, 1, figsize=(6, 4))

ax.scatter(x, y, s=20, color='none', edgecolor='steelblue',
    lw=0.75, alpha=0.7, label='Sample')

# Grid on which to evaluate true model and predictions
xvalues = np.linspace(np.amin(x), np.amax(x), 100)

# Plot true relationship
ax.plot(xvalues, compute_true_y(xvalues), color='black', lw=1.0, alpha=0.9,
    label='True function')

# Plot prediction from optimal Lasso model
ax.plot(xvalues, pipe_ridge.predict(xvalues[:, None]), c='darkorange',
    lw=2.0, label=r'Lasso (optimal $\alpha$)')

ax.legend()
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
```

```
[58]: Text(0, 0.5, '$y$')
```



10.6 Dealing with categorical data (optional)

So far in this unit, we only dealt with continuous data, i.e., data that can take on (almost any) real value. However, many data sets contain categorical variables which take on a finite number of admissible values or even binary variables (often called dummy or indicator variables) which can be either 0 or 1. Dealing with such data in `scikit-learn` is less straightforward than in most other software packages, so in this section we illustrate how to work with categorical variables.

To do this, we again load the Ames house data which contains several categorical and binary variables.

```
[59]: import pandas as pd

      df = pd.read_csv(f'{DATA_PATH}/ames_houses.csv')
```

For example, consider the `BuildingType` variable, which (in this simplified version of the data) takes on three values, 'Single-family', 'Townhouse' and 'Two-family' and has on top a few missing observations. We use the `value_counts()` method to tabulate the number of observations falling into each category. The argument `dropna=False` also includes the number of missing values in the tabulation.

```
[60]: df['BuildingType'].value_counts(dropna=False).sort_index()
```

```
[60]: Single-family    1220
      Townhouse         114
      Two-family         52
      NaN                74
      Name: BuildingType, dtype: int64
```

There are several ways to deal with such data (see the official documentation for details). First, since the data are stored as strings, we could use `OrdinalEncoder` to map these strings to integers.

```
[61]: from sklearn.preprocessing import OrdinalEncoder

      # Drop rows with NA
```

```
df = df.dropna(subset='BuildingType')

enc = OrdinalEncoder()

# Transform string variable into integers
bldg_int = enc.fit_transform(df[['BuildingType']].to_numpy())

# Print unique values and histogram
print(f'Unique values: {np.unique(bldg_int)}')
print(f'Histogram: {np.bincount(bldg_int.astype(int).flatten())}')
```

```
Unique values: [0. 1. 2.]
Histogram: [1220  114   52]
```

This, however, is still not particularly useful if we want to use these categories as explanatory variables in a `LinearRegression` because `scikit-learn` will simply treat them as a continuous variable that happens to take on the values 0, 1 or 2. There is nothing to enforce the categorical nature of this data.

An alternative encoding strategy is to create a binary dummy variable for each possible value of a categorical variable. This is achieved using the OneHotEncoder transformation as illustrated by the following code:

[62]:
```
from sklearn.preprocessing import OneHotEncoder

# Drop rows with NA
df = df.dropna(subset='BuildingType')

# List of unique categories
bldg_uniq = list(np.sort(df['BuildingType'].unique()))

# Create dummy variable encoder
enc = OneHotEncoder(categories=[bldg_uniq], sparse_output=False)

# Convert string variable into binary indicator variables
bldg_dummies = enc.fit_transform(df[['BuildingType']].to_numpy())
```

When creating a `OneHotEncoder`, we can optionally pass the list of possible values using the `categories=...` argument. By default, this transformation returns a sparse matrix since each row will have exactly one element that is 1 while the remaining elements are 0. Using a sparse matrix saves memory but makes interacting with the resulting array more cumbersome. For small data sets, there is no need to use sparse matrices.

The transformed data now looks as follows:

[63]:
```
# Print first 5 rows
bldg_dummies[:5]
```

[63]:
```
array([[1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.]])
```

In this particular example, the first five observations fall into the first category, hence the first column contains ones while the remaining elements are zero.

We can sum the number of ones in each column to verify that the frequency of each category remains the same:

[64]:
```
bldg_dummies.sum(axis=0)
```

[64]:
```
array([1220.,  114.,   52.])
```

The category labels taken from the original data are stored in the `categories_` attribute:

[65]:
```
enc.categories_[0]
```

[65]:
```
array(['Single-family', 'Townhouse', 'Two-family'], dtype=object)
```

Now that we have converted the categories into a dummy matrix, we can append it to the continuous explanatory variables and fit the linear model as usual:

[66]:
```
from sklearn.linear_model import LinearRegression
import pandas as pd

# Name of dependent variable
target = 'SalePrice'
# Continuous explanatory variables to include
continuous = ['LivingArea', 'LotArea']

y = df[target].to_numpy()
# Feature matrix with dummies appended
X = np.hstack((df[continuous].to_numpy(), bldg_dummies))

# Create and fit linear model
lr = LinearRegression(fit_intercept=False)
lr.fit(X, y)

# Create DataFrame containing estimated coefficients
labels = continuous + [f'BuildingType: {s}' for s in enc.categories_[0]]

coefs = pd.DataFrame(lr.coef_, index=labels, columns=['Coef'])
coefs
```

[66]:
```
                                    Coef
LivingArea                   1152.605777
LotArea                         8.385689
BuildingType: Single-family  12173.902329
BuildingType: Townhouse      37735.490189
BuildingType: Two-family    -41842.492374
```

Finally, you should be aware that pandas provides the function `get_dummies()` which accomplishes something very similar to `OneHotEnconder` but may not work as well with the `scikit-learn` API.

[67]:
```
pd.get_dummies(df['BuildingType']).head(5)
```

[67]:
```
   Single-family  Townhouse  Two-family
0              1          0           0
1              1          0           0
2              1          0           0
3              1          0           0
4              1          0           0
```

10.7 Optional exercises

Exercise 1: Interactions of explanatory variables

In applied work, we frequently encounter interactions of variables, i.e., an explanatory variable is formed as the product of two other variables. For example, if our data contains the variables x and z, we might add $x \cdot z$ as an independent variable to the model.

To illustrate, consider the following linear model which contains the interaction terms $d \cdot x$, $d \cdot z$ and $x \cdot z$:

$$y_i = \alpha + \beta_1 d_i + \beta_2 x_i + \beta_3 z_i + \beta_4 d_i x_i + \beta_5 d_i z_i + \beta_6 x_i z_i + \epsilon_i$$

where y is a function of (d, x, z) but is only observed with an additive error term ϵ_i. Assume that these variables are mutually independent and distributed as follows:

- d_i is a dummy variable that is 1 with a probability of 60% and zero otherwise;
- x_i is uniformly distributed on the interval $[0, 1]$;
- z_i is normally distributed with mean 1 and variance 4; and
- The error term ϵ_i is normally distributed with mean 0 and variance 0.3^2.

Create an artificial data set of size $N = 200$ and compute y assuming for the following model parameters:

[68]:
```
# Intercept
alpha = 0.1

# array containing beta_1, beta_2, ..., beta_6
betas = np.array([0.5, 1.5, 2.0, -0.5, -0.75, -0.1])
```

Perform the following tasks:

1. Split the sample into a training data set that comprises the first 150 observations and use the remainder as the test sample.
2. Create the matrix of features $\mathbf{X} = [d, x, z, d \cdot x, d \cdot z, x \cdot z]$. Use the `PolynomialFeatures` preprocessor to create the last three interaction terms.

   Hint: You can use `PolynomialFeatures(degree=(2, 2), interaction_only=True)` which creates interaction terms where the sum of exponents sums to 2.
3. Fit the model as a `LinearRegression`. Compute the predicted values on the test sample and plot the prediction error against the test sample values of y.
4. Create a figure with two panels and plot the model predictions for $d = 0$ and the left and $d = 1$ on the right. Each panel should display the predictions for a grid of 101 values for x uniformly spaced on $[0, 1]$ on the x-axis. Plot three different lines, one for z evaluated at the 25th, 50th and 75th percentile of the sample distribution. Contrast the predictions with the true values of y.

Exercise 2: Polynomial under- and overfitting

Consider the following non-linear model,

$$y_i = \cos\left(\frac{3}{2}\pi x_i\right) + \epsilon_i$$

where y_i is a trigonometric function of x_i but is measured with an additive error ϵ_i. In this exercise, we are going to approximate y_i using polynomials in x_i of varying degrees:

1. Create a sample of size $N = 50$ where the x_i are randomly drawn from a uniform distribution on the interval $[0, 1]$ and $\epsilon_i \overset{\text{iid}}{\sim} N(0, 0.2^2)$. Then generate y_i according to the equation given above.
2. Create a scatter plot of the sample (x_i, y_i) and add a line depicting the true non-linear relationship (without measurement error).
3. Use the `PolynomialFeatures` transformation and `LinearRegression` to approximate y as a polynomial in x. Fit this model using the polynomial degrees $d \in \{0, 1, 2, 3, 6, 10\}$.
4. Create a figure with 6 panels, one for each polynomial degree. Each panel should show the sample scatter plot, the true function $y = f(x)$ and the polynomial approximation of a given degree.
5. How does the quality of the approximation change as you increase d? Do higher-order polynomials always perform better?

Hint: When creating polynomials with `PolynomialFeatures(..., include_bias=True)`, you need to fit the model without an additional intercept as the intercept is already included in the polynomial.

Hint: The cosine function and the constant π are implemented as `np.cos()` and `np.pi` in NumPy.

Exercise 3: Polynomial fitting with scikit-learn pipelines

In this exercise, we explore how to simplify fitting models to polynomials using a feature of `scikit-learn` called pipelines. These allow us to fuse several transformations and an estimation step into a single model.

Continuing with the setup from the previous exercise, we now focus on the model with polynomial degree $d = 5$. Use either `make_pipeline()` or the `Pipeline` class to create a model that combines the `PolynomialFeatures` transformation and the fitting of a linear model in a single step.

Fit the model using this pipeline. Plot the sample, the true relationship and the predicted value for $d = 5$ on a grid of 101 x-values that are uniformly spaced on $[0, 1]$.

Exercise 4: Optimal polynomial degree with cross-validation

In the previous exercises, we fitted polynomials of varying degrees but did not make any systematic attempt to assess the model fit and choose the preferred polynomial degree (which in this setting is a so-called *hyperparameter*). In this exercise, we explore how cross-validation can be used to tune a hyperparameter in a systematic way.

Consider the following model where y is a non-linear trigonometric function of x and includes an additive error term ϵ,

$$y_i = \sin\bigl(2\pi(x_i + 0.1)\bigr) + \epsilon_i$$

and x and ϵ are assumed to be independent.

Proceed as follows:

1. Create a sample of $N = 100$ observations. Draw the x_i from the standard uniform distribution on $[0, 1]$ and let $\epsilon \overset{\text{iid}}{\sim} N(0, 0.25)$. Compute y_i according to the equation above.

   Hint: The sine function and the constant π are implemented as `np.sin()` and `np.pi` in NumPy.

2. Create a sample scatter plot of (x_i, y_i) and add a line showing the true relationship between x and y (without measurement error).

3. Program a function `compute_splits_mse(d, x, y, n_splits)` which takes as arguments the polynomial degree d, the sample observations (x, y) and the number of splits `n_splits` and returns the mean squared error (MSE) for the test sample for each of the splits. Thus the function should return an array of `n_split` MSEs.

   For this exercise, ignore the three-way split into training/validation/test samples illustrated in the lecture and only focus on the training/test samples within each split.

   Hint: You do not need to assign training/test samples manually. Use the `KFold` class and call its `split()` method to do the work for you!

   Hint: To compute the MSE for each test sample, you can use `mean_squared_error()`.

4. Using the function you just wrote, compute the MSEs for polynomial degrees $d = 0, \dots, 15$ using `n_split=10` splits. For each d, compute the average MSE (averaging over 10 splits) and its standard deviation, and use these to create a plot of average MSE on the y-axis against d on the x-axis.

5. Which degree d results in the lowest average MSE? Explain the intuition behind your findings.

6. Add the fitted line using the optimal d into the scatter plot you created earlier to visualise your results.

Exercise 5: Automating cross-validation with scikit-learn

In the previous exercise, we performed fitting and cross-validation mostly manually. This exercise explores how most of these steps can be automated using the `scikit-learn` API.

Using the same setup (functional form for y, sample size and distributional assumptions for x and ϵ) as in the previous exercise, perform the following tasks:

1. For each polynomial degree $d = 0, \dots, 15$, define a `Pipeline` that transforms x into a polynomial and fits a linear model.

   Hint: To do this, you can either create an instance of `Pipeline` or use the convenience function `make_pipeline()`.

2. For each d, perform the sample split and computation of MSEs using the function `cross_val_score()`, which takes as arguments the pipeline you just created, the sample, the metric used to compute the store (`scoring=...`) and the number of folds (`cv=10`).

   Hint: To compute the MSE, you need to specify `scoring='neg_mean_squared_error'`. Note that this returns the *negative* MSE, so you need to correct for this manually.

3. Perform the remaining tasks as in the previous exercise, i.e., compute the mean MSE for each d, find the d that gives the lowest average MSE and plot the average MSE against d.

Exercise 6: Optimising hyperparameters with validation curves

This exercise asks you implemented an even faster way to find the optimal polynomial degree d discussed in the previous exercises.

Using the same setup (functional form for y, sample size and distributional assumptions for x and ϵ) as in the previous exercise, perform the following tasks:

1. As before, create a `Pipeline` that performs the polynomial transformation and the fitting of the linear model. You can use any value for `PolynomialFeatures(degree=...)`.

2. Use the convenience function `validation_curve()` to compute the MSE for *each* $d = 0, \dots, 15$ and all cross-validation splits in a single call.

   To do this, the argument `param_name` specifies which parameter needs to be varied, and `param_range` determines the set of parameter values that are evaluated.

   Since our estimator is a pipeline, the parameter name needs to be specified as `'STEP__ARGUMENT'` where `STEP` is the name of the step in the pipeline we defined (see `Pipeline` for details), and `ARGUMENT` determines the argument name to be varied when re-creating the pipeline, which in our case is `degree`.

   Hint: For pipelines created using `make_pipeline()`, the name of each step in the pipeline is the class name converted to lower case, i.e., for `PolynomialFeatures` the step name is `polynomialfeatures`.

3. Perform the remaining tasks as in the previous exercise, i.e., compute the mean MSE for each d, find the d that gives the lowest average MSE and plot the average MSE against d.

The `scikit-learn` documentation provides additional information on validation curves which you might want to consult.

10.8 Solutions

Exercise 1: Interactions of explanatory variables

Creating the data set

[69]:
```
import numpy as np
import pandas as pd
from numpy.random import default_rng

rng = default_rng(1234)

N = 200

# Intercept
alpha = 0.1

# array containing beta_1, beta_2, ..., beta_6
betas = np.array([0.5, 1.5, 2.0, -0.5, -0.75, -0.1])

d = rng.binomial(n=1, p=0.6, size=N)
x = rng.uniform(0.0, 1.0, size=N)
z = rng.normal(loc=1.0, scale=2.0, size=N)
error = rng.normal(loc=0.0, scale=0.3, size=N)

def compute_y(d, x, z):
    return alpha + betas[0]*d + betas[1]*x + betas[2]*z + betas[3]*d*x \
        + betas[4]*d*z + betas[5]*x*z

y = compute_y(d, x, z) + error
```

The easiest way to visualise the data set is to convert it to a `DataFrame` and print the first few rows.

[70]:
```
df = pd.DataFrame({'y': y, 'd': d, 'x': x, 'z': z})
df.head(5)
```

[70]:
```
          y  d         x         z
0  2.611021  0  0.747273  1.030816
1  2.820446  1  0.002071  1.695387
2  5.234120  0  0.809356  2.074067
3  2.525688  1  0.806809  0.901620
4 -0.065962  1  0.054533 -0.594632
```

Creating interactions

The `PolynomialFeatures` transformer creates multivariate polynomials from a feature matrix $\mathbf{X}$ of a given degree. For example, if we have two variables x and z and set `degree=2`, it would return a matrix with the columns $[x, x^2, z, z^2, x \cdot z]$. The quadratic terms are not part of our model, so we pass `interactions_only=True` to keep only the interaction terms.

[71]:
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split

X = df[['d', 'x', 'z']].to_numpy()

# Number of obs. in training dataset
N_train = 150

# Split into training / test data
```

```
X_train, X_test = X[:N_train], X[N_train:]
y_train, y_test = y[:N_train], y[N_train:]

poly = PolynomialFeatures(degree=(2, 2), include_bias=False, interaction_only=True)
X_train_interactions = poly.fit_transform(X_train)
```

It is convenient to use a `DataFrame` to print a few observations of the interaction terms that were created.

```
[72]: pd.DataFrame(X_train_interactions, columns=['d*x', 'd*z', 'x*z']).head(5)
```

```
[72]:         d*x       d*z       x*z
      0  0.000000  0.000000  0.770301
      1  0.002071  1.695387  0.003511
      2  0.000000  0.000000  1.678660
      3  0.806809  0.901620  0.727435
      4  0.054533 -0.594632 -0.032427
```

The complete feature matrix **X** of course contains both the original non-interacted variables as well as the interactions, so we stack the two matrices horizontally.

```
[73]: X_train_all = np.hstack((X_train, X_train_interactions))
      X_test_all = np.hstack((X_test, poly.transform(X_test)))
```

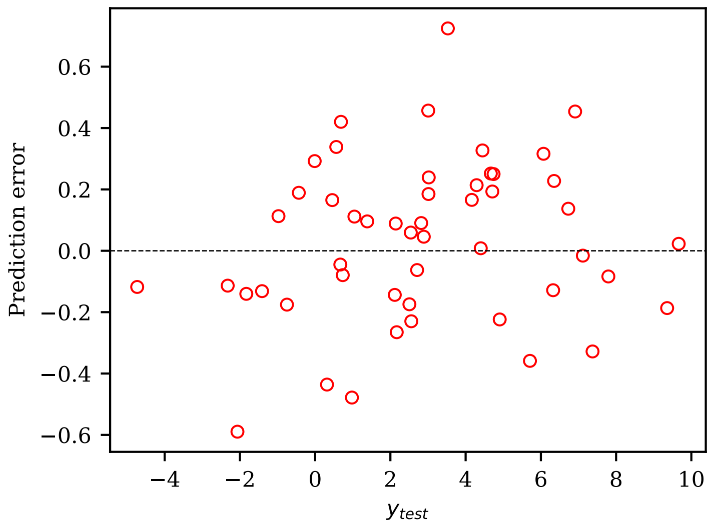
Fitting a linear regression model

We fit the model in the usual way and use `predict()` to obtain the predicted values $\widehat{y}_i$ for the test data set.

```
[74]: from sklearn.linear_model import LinearRegression

      lr = LinearRegression(fit_intercept=False)
      lr.fit(X_train_all, y_train)

      # Predict response variable on test sample
      y_test_hat = lr.predict(X_test_all)
      error = y_test - y_test_hat
```

```
[75]: import matplotlib.pyplot as plt

      # Plot prediction errors against response variable on test sample
      plt.scatter(y_test, error, lw=0.75, color='none', edgecolor='red', s=20)
      plt.axhline(0.0, lw=0.5, ls='--', c='black')
      plt.xlabel('$y_{test}$')
      plt.ylabel('Prediction error')
```

```
[75]: Text(0, 0.5, 'Prediction error')
```



Alternative way to construct interactions

Using the preprocessing and pipeline features of `scikit-learn`, we can also construct the feature matrix in a more elegant way using a `FeatureUnion` together with a `FunctionTransformer`.

The `FunctionTransformer` is initialized using the identity function `lambda x: x` and hence just returns the argument. The second component of the `FeatureUnion` is the `PolynomialFeatures` transformer which creates the interaction term in the same way as before.

[76]:
```
from sklearn.pipeline import FeatureUnion
from sklearn.preprocessing import FunctionTransformer

# Create complete feature matrix comprising both the original variables [d,x,z]
# as well as their interactions
transform_x = FeatureUnion([
    ('orig', FunctionTransformer(lambda x: x)),
    ('interact',
        PolynomialFeatures(
            degree=(2, 2),
            include_bias=False,
            interaction_only=True
        )
    )
])

# Verify that these are identical to the manually constructed feature matrix
x2 = transform_x.fit_transform(X_train)
np.all(x2 == X_train_all)
```

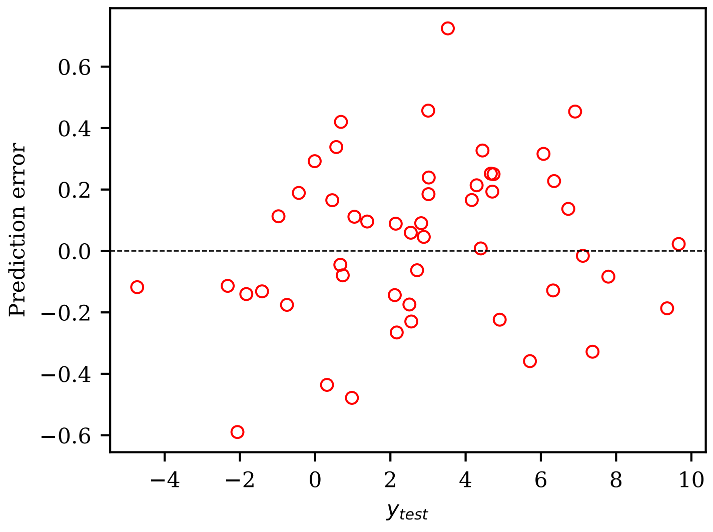
[76]:
```
True
```

We can now combine this transformer with the `LinearRegression` into a pipeline to get an object which directly fits the linear model for y.

[77]:
```
from sklearn.pipeline import make_pipeline

# Create pipeline from X transformer and linear regression model
model = make_pipeline(
    transform_x,
    LinearRegression(fit_intercept=False)
)
```

```
model.fit(X_train, y_train)

# Predict response variable on test sample
y_test_hat = model.predict(X_test)
error = y_test - y_test_hat

# Plot prediction errors against response variable on test sample
plt.scatter(y_test, error, lw=0.75, color='none', edgecolor='red', s=20)
plt.axhline(0.0, lw=0.5, ls='--', c='black')
plt.xlabel('$y_{test}$')
plt.ylabel('Prediction error')
```

[77]:
```
Text(0, 0.5, 'Prediction error')
```



Plotting predicted vs. true values

To compute the predicted values on a grid, we first need to create the 3-dimensional feature space. One easy way to do this is to create the 1-dimensional grids for each dimension and then form the Cartesian product using `pandas.merge(..., how='cross')` which creates the cross-product of two `DataFrame` objects.

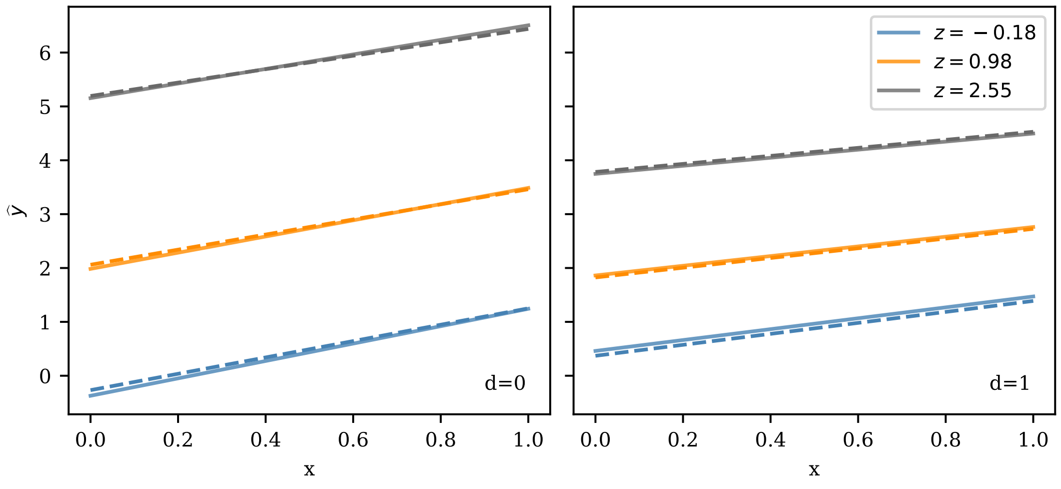
[78]:
```
dgrid = np.array([0, 1])
xgrid = np.linspace(0.0, 1.0, 101)
# Obtain grid for z as 25, 50, 75 percentiles of data
zgrid = np.percentile(z, q=[25.0, 50.0, 75.0])

df_pred = pd.DataFrame({'d': dgrid})
df_pred = df_pred.merge(pd.DataFrame({'x': xgrid}), how='cross')
df_pred = df_pred.merge(pd.DataFrame({'z': zgrid}), how='cross')

df_pred.head(6)
```

[78]:
```
   d     x         z
0  0  0.00 -0.183693
1  0  0.00  0.980220
2  0  0.00  2.546668
3  0  0.01 -0.183693
4  0  0.01  0.980220
5  0  0.01  2.546668
```

We plot the predictions for $d = 0$ on the left and the ones for $d = 1$ on the right. Each plot contains three lines, one for each value of z. The model predictions are plotted using solid lines while the true values are dashed.

[79]:
```python
fig, axes = plt.subplots(1, 2, figsize=(6, 2.75), sharex=True, sharey=True,
    constrained_layout=True)

colors = ['steelblue', 'darkorange', 'dimgrey']

for i, d in enumerate(dgrid):
    ax = axes[i]

    df_pred_d = df_pred[df_pred['d'] == d]
    X_pred = df_pred_d.to_numpy()
    y_pred = model.predict(X_pred)
    y_true = compute_y(d, df_pred_d['x'].to_numpy(), df_pred_d['z'].to_numpy())

    y_pred = y_pred.reshape((len(xgrid), len(zgrid)))
    y_true = y_true.reshape(y_pred.shape)

    for j, zj in enumerate(zgrid):
        ax.plot(xgrid, y_pred[:, j], c=colors[j], label=f'$z = {zj:.2f}$', alpha=0.8)
        ax.plot(xgrid, y_true[:, j], c=colors[j], ls='--')

    ax.set_xlabel('x')
    ax.text(0.95, 0.05, f'd={d}', transform=ax.transAxes, va='bottom', ha='right')

# Add legend to right panel
axes[1].legend()

# Add y-labels only to left panel
axes[0].set_ylabel(r'$\widehat{y}$')
```

[79]:
```
Text(0, 0.5, '$\\widehat{y}$')
```



Exercise 2: Polynomial under- and overfitting

Creating and plotting the sample

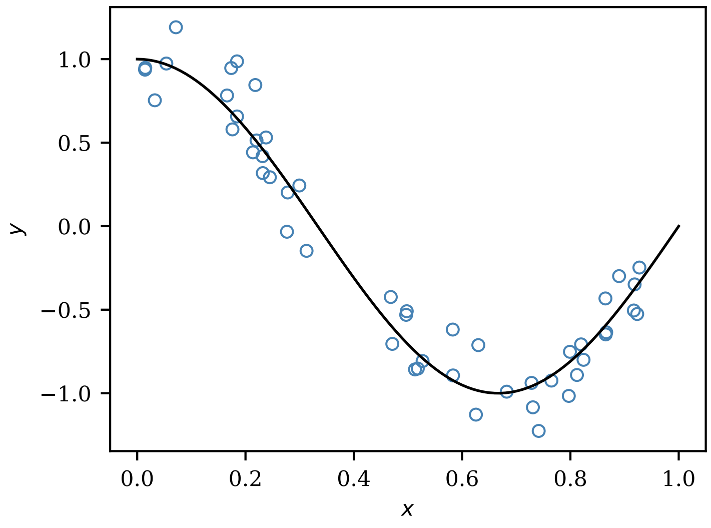
We randomly draw values for x_i and ϵ_i for each observation. We define a function `fcn(x)` which returns the true value for y without measurement error.

[80]:
```
import numpy as np

# True function (w/o errors)
def fcn(x):
    return np.cos(1.5 * np.pi * x)
```

[81]:
```
from numpy.random import default_rng

# Initialise random number generator
rng = default_rng(123)

# Sample size
N = 50

# Randomly draw explanatory variable x
x = rng.random(size=N)
epsilon = rng.normal(scale=0.2, size=N)
y = fcn(x) + epsilon
```

[82]:
```
import matplotlib.pyplot as plt


xvalues = np.linspace(0.0, 1.0, 101)
plt.plot(xvalues, fcn(xvalues), color='black', lw=1.0, label='True function')

plt.scatter(x, y, s=20, color='none', edgecolor='steelblue',
    lw=0.75, label='Sample')

plt.xlabel('$x$')
plt.ylabel('$y$')
```

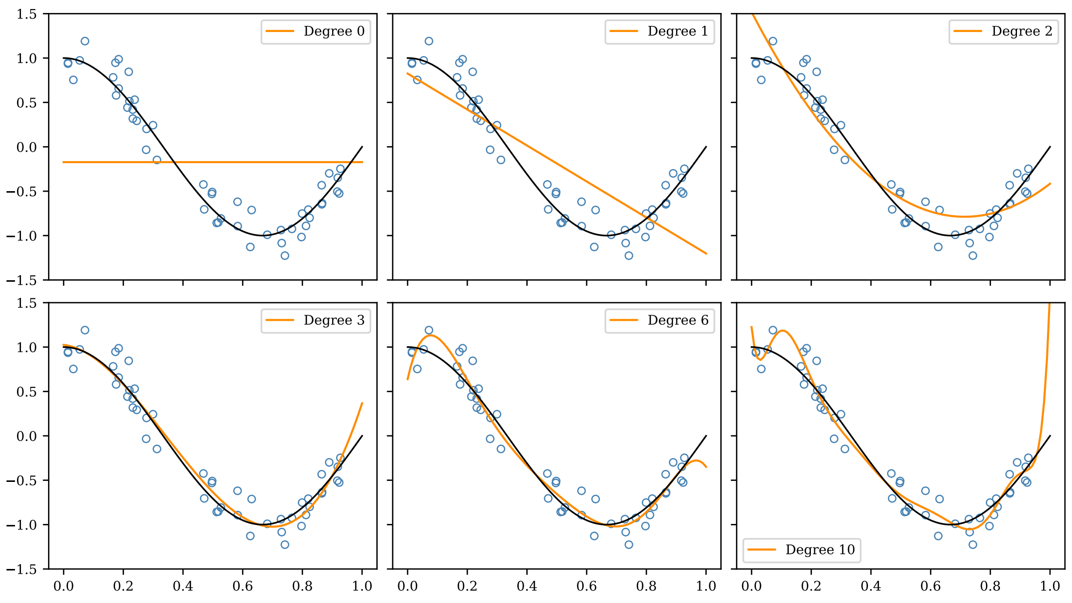
[82]:
```
Text(0, 0.5, '$y$')
```



Polynomial approximations

To plot the predicted values for each value of the polynomial degree *d*, we create a 2-by-3 figure and iterate over the axes objects. For each panel and corresponding *d*, we create a transformation as `PolynomialFeatures(degree=d, include_bias=True)` and use it to create a polynomial in *x*.

[83]:
```python
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

degrees = np.array([0, 1, 2, 3, 6, 10])

ncol = 3
nrow = int(np.ceil(len(degrees) / ncol))

fig, axes = plt.subplots(nrow, ncol, figsize=(9, 5),
    sharex=True, sharey=True,
    constrained_layout=True
)

axes = axes.flatten()

for i, ax in enumerate(axes):
    d = degrees[i]
    poly = PolynomialFeatures(degree=d, include_bias=True)
    poly.fit(x[:, None])
    Xpoly = poly.transform(x[:, None])

    lr = LinearRegression(fit_intercept=False)
    lr.fit(Xpoly, y)

    y_hat = lr.predict(poly.transform(xvalues[:, None]))
    ax.plot(xvalues, y_hat, lw=1.25, c='darkorange', label=f'Degree {d}')
    ax.plot(xvalues, fcn(xvalues), color='black', lw=1.0)
    ax.scatter(x, y, s=20, color='none', edgecolor='steelblue', lw=0.75)
    ax.set_ylim((-1.5, 1.5))
    ax.legend()
```



As the figure shows, a degree-0 polynomial is just a constant, while a polynomial of degree 1 is linear in x. Neither fits the true relationship very well ("underfitting"), but the fit initially improves as we increase d. For high d, on the other hand, the polynomial becomes too flexible and responds strongly to local "noise" introduced by measurement error ("overfitting").

Exercise 3: Polynomial fitting with scikit-learn pipelines

We create the sample in the same way as in exercise 2.

[84]:
```
import numpy as np

# True function (w/o errors)
def fcn(x):
    return np.cos(1.5 * np.pi * x)
```

[85]:
```
from numpy.random import default_rng

# Initialise random number generator
rng = default_rng(123)

# Sample size
N = 100

# Randomly draw explanatory variable x
x = np.sort(rng.random(size=N))
epsilon = rng.normal(scale=0.2, size=N)
y = fcn(x) + epsilon
```

To transform x and fit the linear model in a single step, we create a pipepline using `make_pipeline()` as follows:

[86]:
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LinearRegression

degree = 5

# Create PipeLine object that consist of a polynomial transformation and
# a linear regression model
pipe_lr = make_pipeline(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    LinearRegression(fit_intercept=True)
)
```

In Jupyter notebooks, printing the pipeline object produces a nice graphical representation that can be unfolded to see any parameters for each step of the pipeline (this will not be visible in the PDF version of this notebook).

[87]:
```
# Show graphical representation of pipeline (only in Jupyter notebook)
pipe_lr
```

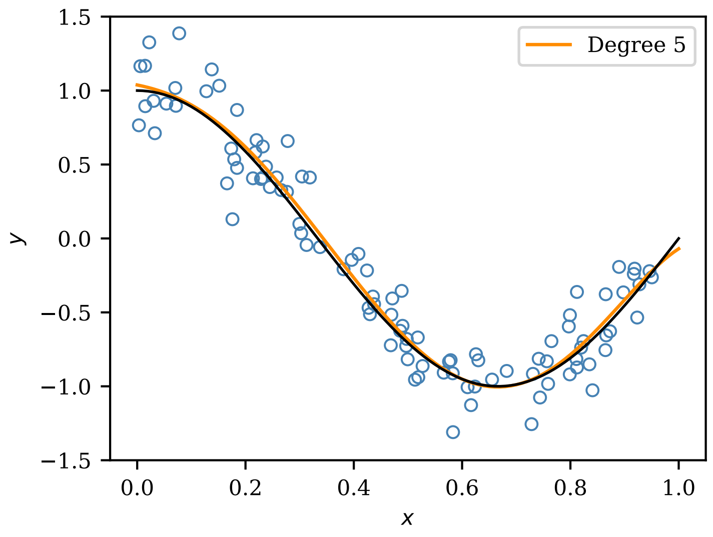
[87]:
```
Pipeline(steps=[('polynomialfeatures',
                 PolynomialFeatures(degree=5, include_bias=False)),
                ('linearregression', LinearRegression())])
```

The input data can be transformed and the model fit in a single step by calling the `fit()` method of the pipeline object.

[88]:
```
# Use pipeline to fit model
pipe_lr.fit(x[:, None], y)

# Compute predicted values on uniform grid
xvalues = np.linspace(0.0, 1.0, 101)
y_hat = pipe_lr.predict(xvalues[:, None])

# Plot predicted values
plt.plot(xvalues, y_hat, lw=1.25, c='darkorange', label=f'Degree {degree}')
# Plot true function
```

```
plt.plot(xvalues, fcn(xvalues), color='black', lw=1.0)
# Plot sample data
plt.scatter(x, y, s=20, color='none', edgecolor='steelblue', lw=0.75)
plt.ylim((-1.5, 1.5))
plt.ylabel('$y$')
plt.xlabel('$x$')
plt.legend()
```

[88]:
```
<matplotlib.legend.Legend at 0x7fca6641b3d0>
```



Exercise 4: Optimal polynomial degree with cross-validation

Creating and plotting the sample

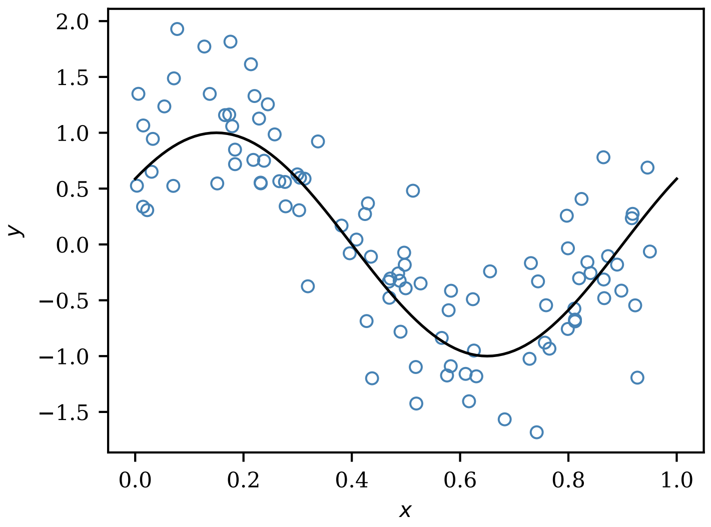
[89]:
```
import numpy as np

# True function (w/o errors)
def fcn(x):
    return np.sin(2.0 * np.pi * (x + 0.1))
```

[90]:
```
from numpy.random import default_rng

# Initialise random number generator
rng = default_rng(123)

# Sample size
N = 100

# Randomly draw explanatory variable x
x = rng.random(size=N)
epsilon = rng.normal(scale=0.5, size=N)
y = fcn(x) + epsilon
```

[91]:
```
import matplotlib.pyplot as plt

xvalues = np.linspace(0.0, 1.0, 101)
plt.plot(xvalues, fcn(xvalues), color='black', lw=1.0, label='True function')

plt.scatter(x, y, s=20, color='none', edgecolor='steelblue',
    lw=0.75, label='Sample')
```

```
)

plt.xlabel('$x$')
plt.ylabel('$y$')
```

[91]:
```
Text(0, 0.5, '$y$')
```



Function to perform fitting and MSE computation

The following implementation uses two distinct functions:

1. `fit_poly()` takes a polynomial degree `d` and the sample data, creates the polynomial and fits a linear model.
2. `compute_splits_mse()` splits the sample into `n_splits` folds, fits the model on the training subset of each split and computes the MSE on the test (or validation) sample.

[92]:
```python
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# Function to perform polynomial transformation and model fitting.
# Alternatively, we could use a Pipeline (see next exercise)
def fit_poly(d, X, y):
    poly = PolynomialFeatures(degree=d, include_bias=True)
    poly.fit(X)
    Xpoly = poly.transform(X)
    lr = LinearRegression(fit_intercept=False)
    lr.fit(Xpoly, y)

    # Return poly transformer (required for prediction) and fitted model
    return poly, lr
```

[93]:
```python
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error

# Function to compute MSE for given number of splits
def compute_splits_mse(d, x, y, n_splits=10):

    # Split sample into train/test blocks for k-fold validation
    kf = KFold(n_splits=n_splits)

    # list to store MSE for each CV split
    mse_splits = []

    # Manually iterate over folds (train/test combinations)
    for itrain, itest in kf.split(x):
        x_train = x[itrain]
        x_test = x[itest]

        y_train = y[itrain]
        y_test = y[itest]

        poly, lr = fit_poly(d, x_train[:, None], y_train)

        Xpoly_test = poly.transform(x_test[:, None])
        y_test_hat = lr.predict(Xpoly_test)

        mse = mean_squared_error(y_test, y_test_hat)

        mse_splits.append(mse)

    # Convert to array
    mse_splits = np.array(mse_splits)

    return mse_splits
```

We can test this function for a degree-0 polynomial (i.e., a constant function) on our sample data. The functions returns 10 different MSE values, one for the test sample of each split.

```
[94]: compute_splits_mse(0, x, y, n_splits=10)
```

```
[94]: array([1.05594738, 1.19357336, 0.68425812, 0.21465842, 0.64290297,
             0.89604156, 0.44140114, 0.51936866, 0.83432109, 0.59097661])
```

Computing the MSE for each hyperparameter

We now use the functions we defined to evaluate the MSEs for each polynomial degree `d`. For each `d`, `compute_splits_mse()` returns 10 values, so we compute the average MSE and its standard deviation.

```
[95]: degrees = np.arange(16)

      mse_mean = []
      mse_std = []

      for d in degrees:

          mse_splits = compute_splits_mse(d, x, y, n_splits=10)

          mse_mean.append(np.mean(mse_splits))
          mse_std.append(np.std(mse_splits))

      # Convert to NumPy arrays
      mse_mean = np.array(mse_mean)
      mse_std = np.array(mse_std)

      # Print MSEs for each degree
      mse_mean
```

```
[95]: array([0.70734493, 0.42671672, 0.33501395, 0.23900256, 0.21559544,
             0.22029082, 0.22568436, 0.22577954, 0.22433218, 0.23387504,
             0.23689982, 0.24623706, 0.25145758, 0.26281712, 0.27462137,
             0.29084427])
```

Finding the optimal parameter

We use `np.argmin()` to find the *index* of the smallest average MSE.

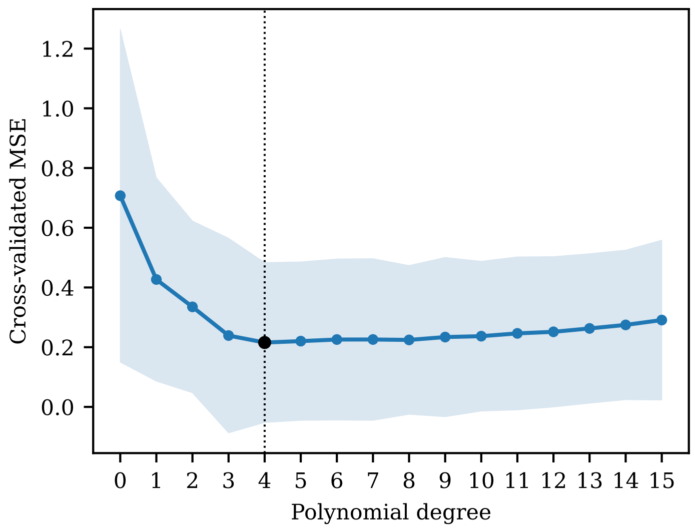
[96]:
```
# Polynomial degree that minimises MSE
imin = np.argmin(mse_mean)
dmin = degrees[imin]

print(f'Polynomial degree with min. MSE: {dmin}')
```

```
Polynomial degree with min. MSE: 4
```

The results can be visualised by plotting the average MSE by polynomial degree. As the graph shows, the minimum is obtained at $d = 4$. Intuitively, for low d the model underfits the data, whereas overfitting occurs for high values of d.

[97]:
```
plt.plot(degrees, mse_mean, marker='o', ms=3)

# Plot "confidence interval"
plt.fill_between(degrees, mse_mean-2.0*mse_std, mse_mean+2.0*mse_std,
    lw=0.25, color='steelblue', alpha=0.2, zorder=-1)

plt.xlabel('Polynomial degree')
plt.ylabel('Cross-validated MSE')
plt.scatter(degrees[imin], mse_mean[imin], s=15, c='black', zorder=100)
plt.xticks(degrees)
plt.axvline(imin, ls=':', lw=0.75, c='black')
```

[97]:
```
<matplotlib.lines.Line2D at 0x7fca664b8670>
```



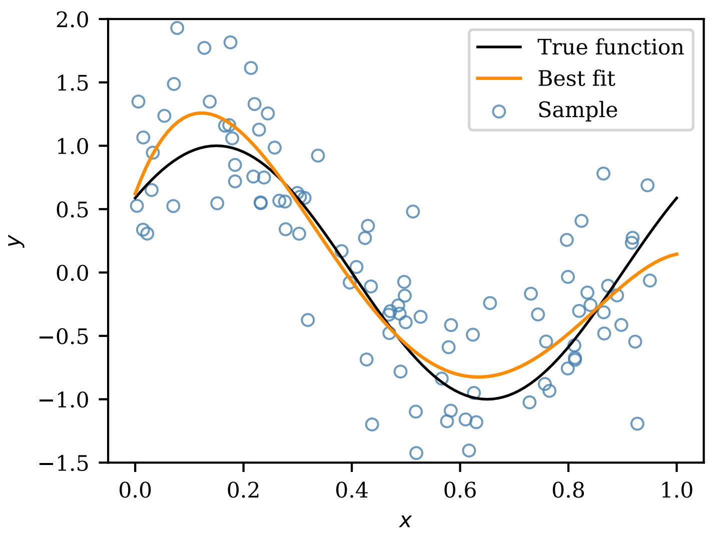
Plotting the fitted model

We use the function `fit_poly()` to re-fit the model at the optimal d and we plot the predicted values on a uniformly spaced grid of x-values.

[98]:
```
poly, lr = fit_poly(dmin, x[:, None], y)

xvalues = np.linspace(0.0, 1.0, 101)
y_hat = lr.predict(poly.transform(xvalues[:, None]))

xvalues = np.linspace(0.0, 1.0, 101)
```

```
plt.plot(xvalues, fcn(xvalues), color='black', lw=1.0,
    label='True function')

plt.plot(xvalues, y_hat, color='darkorange', lw=1.25,
    zorder=10, label='Best fit')

plt.scatter(x, y, s=20, color='none', edgecolor='steelblue', lw=0.75,
    alpha=0.8, label='Sample')

plt.xlabel('$x$')
plt.ylabel('$y$')
plt.ylim(-1.5, 2.0)
plt.legend(loc='upper right')
```

[98]:
```
<matplotlib.legend.Legend at 0x7fca663c6440>
```



Exercise 5: Automating cross-validation with scikit-learn

Creating the sample

We create the sample exactly as in the previous exercise.

[99]:
```
import numpy as np

# True function (w/o errors)
def fcn(x):
    return np.sin(2.0 * np.pi * (x + 0.1))
```

[100]:
```
from numpy.random import default_rng

# Initialise random number generator
rng = default_rng(123)

# Sample size
N = 100

# Randomly draw explanatory variable x
x = rng.random(size=N)
epsilon = rng.normal(scale=0.5, size=N)
y = fcn(x) + epsilon
```

Computing the MSE for each hyperparameter

For each d, we create a pipeline. Note that step one of the pipeline, polynomial transformation, depends on d so we need to recreate the pipeline in each iteration of the loop!

We then use `cross_val_score()` to compute the MSEs for each split. This replaces the function we implemented in the previous exercise with much shorter code.

[101]:
```
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

degrees = np.arange(16)

mse_mean = []
mse_std = []

for d in degrees:

    pipe = make_pipeline(
        PolynomialFeatures(
            degree=d,
            include_bias=True
        ),
        LinearRegression(fit_intercept=False)
    )

    scores = cross_val_score(
        pipe,
        x[:, None], y,
        scoring='neg_mean_squared_error',
        cv=10
    )

    # Stores contain NEGATIVE MSE
    mse_mean.append(np.mean(-scores))
    mse_std.append(np.std(-scores))


# Convert to NumPy arrays
mse_mean = np.array(mse_mean)
mse_std = np.array(mse_std)

# Print avg MSEs for each degree
mse_mean
```

[101]:
```
array([0.70734493, 0.42671672, 0.33501395, 0.23900256, 0.21559544,
       0.22029082, 0.22568436, 0.22577954, 0.22433218, 0.23387504,
       0.23689982, 0.24623706, 0.25145758, 0.26281712, 0.27462137,
       0.29084427])
```

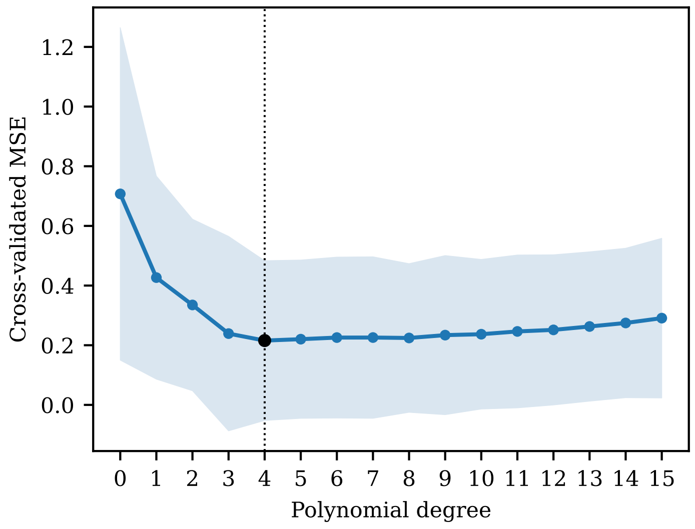
The remained of the exercise proceeds in the same way as before. Assuringly, we also find the same results.

[102]:
```
imin = np.argmin(mse_mean)
dmin = degrees[imin]

print(f'Polynomial degree with min. MSE: {dmin}')
```

```
Polynomial degree with min. MSE: 4
```

[103]:
```
plt.plot(degrees, mse_mean, marker='o', ms=3)

plt.fill_between(degrees, mse_mean-2.0*mse_std, mse_mean+2.0*mse_std,
    lw=0.25, color='steelblue', alpha=0.2, zorder=-1)

plt.xlabel('Polynomial degree')
plt.ylabel('Cross-validated MSE')
plt.scatter(degrees[imin], mse_mean[imin], s=15, c='black', zorder=100)
plt.xticks(degrees)
plt.axvline(imin, ls=':', lw=0.75, c='black')
```

[103]:
```
<matplotlib.lines.Line2D at 0x7fca662af700>
```



Exercise 6: Optimising hyperparameters with validation curves

Creating the sample

We create the sample in the same way we did in exercises 4 and 5.

[104]:
```
import numpy as np

# True function (w/o errors)
def fcn(x):
    return np.sin(2.0 * np.pi * (x + 0.1))
```

[105]:
```
from numpy.random import default_rng

# Initialise random number generator
rng = default_rng(123)

# Sample size
N = 100

# Randomly draw explanatory variable x
x = rng.random(size=N)
epsilon = rng.normal(scale=0.5, size=N)
y = fcn(x) + epsilon
```

Computing the MSE for each hyperparameter

To use `validation_curve()`, we need to correctly specify the `param_name` argument. In our case this is given by `'polynomialfeatures__degree'` since we are asking the function to vary the argument `degree` of the pipeline step called `polynomialfeatures`. Note that the `__` serves as a delimiter between the pipeline step name and the argument name.

[106]:
```python
from sklearn.model_selection import validation_curve

# Set of polynomial degrees to cross-validate
degrees = np.arange(16)

# Create pipeline. The degree value does not matter at this point, will be
# automatically updated.
pipe = make_pipeline(
    PolynomialFeatures(
        degree=0,
        include_bias=True
    ),
    LinearRegression(fit_intercept=False)
)

# Compute the MSEs for each degree and each split, returning arrays of
# size 15 x 10.
train_scores, test_scores = validation_curve(
    estimator=pipe,
    X=x[:, None], y=y,
    param_name='polynomialfeatures__degree',
    param_range=degrees,
    scoring='neg_mean_squared_error',
    cv=10
)

# Compute mean and std for each degree (scored returned by function are
# NEGATIVE MSEs)
mse_mean = np.mean(-test_scores, axis=1)
mse_std = np.std(-test_scores, axis=1)
```

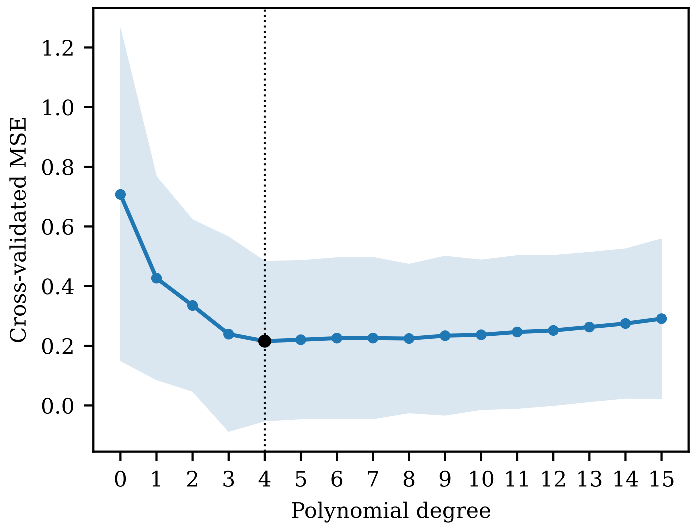
The remainder of this exercise proceeds as before.

[107]:
```python
imin = np.argmin(mse_mean)
dmin = degrees[imin]

print(f'Polynomial degree with min. MSE: {dmin}')
```

```
Polynomial degree with min. MSE: 4
```

[108]:
```python
plt.plot(degrees, mse_mean, marker='o', ms=3)

plt.fill_between(degrees, mse_mean-2.0*mse_std, mse_mean+2.0*mse_std,
    lw=0.25, color='steelblue', alpha=0.2, zorder=-1)

plt.xlabel('Polynomial degree')
plt.ylabel('Cross-validated MSE')
plt.scatter(degrees[imin], mse_mean[imin], s=15, c='black', zorder=100)
plt.xticks(degrees)
plt.axvline(imin, ls=':', lw=0.75, c='black')
```

[108]:
```
<matplotlib.lines.Line2D at 0x7fca66113d30>
```


Cross-validated MSE
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Polynomial degree

11 Solving household problems in macroeconomics and finance

In this unit, we explore how to solve simple consumption-savings problems that are common in (heterogeneous-agent) macroeconomics and household finance.

For simplicity, we study infinite-horizon problems even though life-cycle models with finitely-lived agents are becoming more common in macroeconomics and are almost universal in household finance. The implementation is almost identical in both cases, with the major difference being that

1. Infinite-horizon problems are solved by iteration until convergence (of the value or policy functions) starting from some initial guess.
2. Life-cycle problems are solved by backward induction, starting from the terminal period T and iterating backwards through periods $T-1$, $T-2$, etc. until the first period.

Throughout this unit, we will exclusively solve partial equilibrium problems for given prices (interest rates and wages). Solving for general equilibrium would require us to find an additional fixed point in terms of equilibrium prices (in steady-state models such as Aiyagari (1994) or Huggett (1993)), or a equilibrium forecasting rule (for models with aggregate uncertainty such as Krusell-Smith (1998)).

For numerical purposes it is necessary to formulate the household problem in recursive form. You may be more familiar with the sequential formulation of an infinite horizon problem which could look something like the following,

$$
\begin{aligned}
V(a_0) = &\max_{(c_t, a_{t+1})_{t=0}^{\infty}} \sum_{t=0}^{\infty} \beta^t u(c_t) \\
\text{s.t.} \quad & c_t + a_{t+1} = (1+r)a_t + y_t \quad \forall\, t \\
& c_t \geq 0,\ a_{t+1} \geq 0 \quad \forall\, t
\end{aligned}
$$

where the household chooses *sequences* of consumption $(c_t)_{t=0}^{\infty}$ and asset levels $(a_{t+1})_{t=0}^{\infty}$ for some initial assets a_0. The value function V is the maximum over such sequences. However, when we solve a problem numerically on the computer, it is usually more convenient to reformulate it recursively to obtain

$$
\begin{aligned}
V(a) = &\max_{c,a'} \Bigl\{ u(c) + \beta V(a') \Bigr\} \\
\text{s.t.} \quad & c + a' = (1+r)a + y \\
& c \geq 0,\ a' \geq 0
\end{aligned}
$$

where the household picks optimal *scalars* c and a' for given a and parameters.

Note: With recursive formulations we often use primes to denote next-period values such as a' instead of writing $t+1$ since time t does not play any explicit role.

11.1 VFI with deterministic income

11.1.1 Household problem

For starters, consider the following *deterministic* infinite-horizon consumption-savings problem,

$$\begin{aligned} V(a) &= \max_{c,a'} \left\{ \frac{c^{1-\gamma}-1}{1-\gamma} + \beta V(a') \right\} \\ \text{s.t.} \quad & c + a' = (1+r)a + y \\ & c \geq 0, \; a' \geq 0 \end{aligned}$$

where a are beginning-of-period assets, and r and y are the interest rate and labour earnings, respectively, which are both exogenous. The household has CRRA utility and chooses optimal consumption c and next-period assets a' which are required to be non-negative (i.e., we impose a borrowing constraint at 0).

Note that from a programming perspective we are interested in the case when $y > 0$ as otherwise this problem can be solved analytically (the household consumes a constant fraction of its assets due to CRRA preferences).

For $y > 0$, this problem has to be solved numerically which we do using *value function iteration (VFI)*. VFI takes an initial guess V_0 and solves the above problem repeatedly, generating a sequence of updated guesses $V_n, V_{n+1}, \dots$. We terminate the algorithm once consecutive V_n do not change anymore.

VFI itself does not dictate how the updated V is computed. In the sections below we explore two alternatives:

1. VFI combined with grid search; and
2. VFI combined with interpolation;

11.1.2 VFI with grid search

The simplest way to solve the above problem is to use the so-called *grid search* algorithm where we try all candidate savings levels from a discrete set of choices (the *grid*). Instead of picking an arbitrary $a' \in [0, (1+r)a + y]$, the household is thus constrained to pick an element $a' \in \Gamma_a$ from the pre-determined set of N_a feasible asset levels

$$\Gamma_a = \left\{ a_1, a_1, a_2, \dots, a_{N_a} \right\}$$

For convenience, we often use the grid Γ_a both as the set of points for which we solve the problem numerically as well as the household's choice set for a', thus forcing the household to pick a point on the grid on which the problem is defined. This is not strictly required (the household could in principle choose from an arbitrary finite set), but allows us to skip any interpolation of the continuation value function $V(a')$.

While grid search is generally considered obsolete in modern applications, it has a few distinct advantages:

1. Easy to implement;
2. Does not require computing derivatives (even more, it does not assume differentiability);
3. Fast (unless the grid is very dense or there are many choices);

However, we usually avoid it because of its disadvantages:

1. It's imprecise, and policy function are often not smooth (unless the grid is very dense);
2. Does not scale well to multiple choice dimensions;

You might still have to resort to grid search if your problem is not differentiable or has local maxima which breaks more sophisticated solution methods.

Outline of the algorithm

The grid search algorithm for this problem can be summarized as follows:

1. Create an asset grid $\Gamma_a = (a_1, \dots, a_{N_a})$.
2. Pick an initial guess V_0 for the value function defined on Γ_a.
3. In iteration n, perform the following steps:

   For each asset level a_i at grid point i,

   1. Find all feasible next-period asset levels $a_j \in \Gamma_a$ that satisfy the budget constraint,
      $$a_j \leq (1+r)a_i + y$$
   2. For each j, compute consumption:
      $$c_j = (1+r)a_i + y - a_j$$
   3. For each j, compute utility:
      $$U_j = u(c_j) + \beta V_n(a_j)$$
   4. Find the index k that maximises the above expression:
      $$k = \arg\max_j \Big\{ u(c_j) + \beta V_n(a_j) \Big\}$$
   5. Set $V_{n+1,i} = U_k$ and store k as the optimal choice at i.
4. Check for convergence: If $\|V_n - V_{n+1}\| < \epsilon$, for some small tolerance $\epsilon > 0$, exit the algorithm.

We implement this algorithm below. For this level of complexity, we would normally want to store the code as regular Python files (`*.py`) as opposed to writing down the problem in a Jupyter notebook. The complete implementation is therefore available in the files

- `lectures/unit11/main.py`: sets up the problem, calls the VFI solver, plots results; and
- `lectures/unit11/VFI.py`: implements VFI with grid search as well as with interpolation;

The sections below walk you through this implementation in notebook format.

Defining parameters and grids

For our implementation, we use the following parameters which are standard in macroeconomics:

Parameter	Description	Value
β	Discount factor	0.96
γ	Coef. of relative risk aversion	2
r	Interest rate	0.04
y	Labour income	1

In Python, it is convenient to store all these parameters in a single object so we don't have to pass numerous arguments to functions that perform the actual computations. To this end, we define a class which serves as a container object:

[1]:
```python
from dataclasses import dataclass

@dataclass
class Parameters:
    """
    Define object to store model parameters and their default values
    """
    beta = 0.96         # Discount factor
    gamma = 1.0         # Risk aversion
    y = 1.0             # Labour income
    r = 0.04            # Interest rate
    grid_a = None       # Asset grid (to be created)
```

The `@dataclass` decorator is a convenient short-hand to define the attributes directly within the class body (we can safely ignore the technical details).

Note: Once our code becomes more complex, it is good practice to document the purpose of a `class` or function using the triple-quote doc strings `""" ... """`. These are ignored by the Python interpreter.

We use this class definition to create a `Parameters` instance named `par` below. Additionally, we create the asset grid `grid_a` which we add to this object:

[2]:
```
import numpy as np

par = Parameters()

# Start + end point for asset grid
a_min = 0.0
a_max = 10.0
# Number of grid points
N_a = 30

# Create asset grid with more points at the beginning
grid_a  = a_min + (a_max - a_min) * np.linspace(0.0, 1.0, N_a)**1.5

# Store asset grid in Parameters object
par.grid_a = grid_a
```

Note the seemingly unconventional way to create the asset grid where we first create a uniform grid on $[0, 1]$ and then compress the grid at lower asset levels using an exponential transformation.

Alternatively, we could have created the usual uniformly-spaced grid:

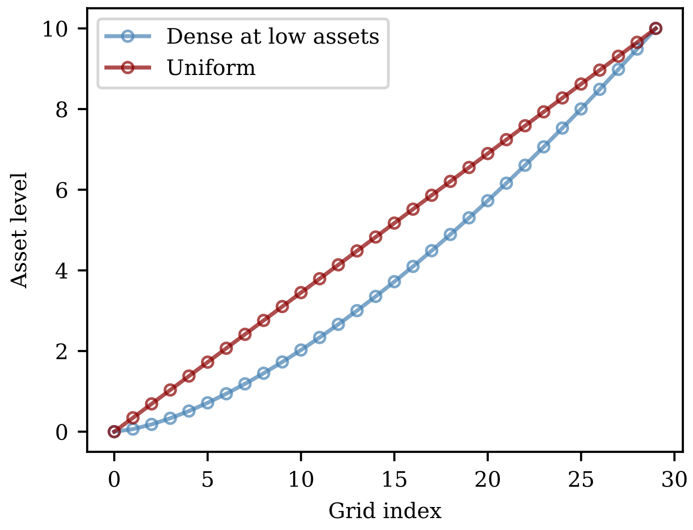
[3]:
```
grid_a_uniform = np.linspace(a_min, a_max, N_a)
```

However, we know from experience that the value and policy functions tend to be nonlinear for low assets, and therefore it is advantageous to put more grid points in that region.

You can see the difference by plotting the resulting grids:

[4]:
```
import matplotlib.pyplot as plt

# Common arguments controlling plot style
style = dict(marker='o', mfc='none', alpha=0.7, markersize=4)

plt.plot(grid_a, c='steelblue', label='Dense at low assets', **style)
plt.plot(grid_a_uniform, c='darkred', label='Uniform', **style)
plt.xlabel('Grid index')
plt.ylabel('Asset level')
plt.legend(loc='upper left')
```

[4]:
```
<matplotlib.legend.Legend at 0x7ff91ed2b220>
```



Implementing VFI with grid search

The following code implements the VFI algorithm with grid search. We define the function

```
def vfi_grid(par, tol=1e-5, maxiter=1000):
    ...
```

which takes as arguments an instance of the `Parameters` class, the termination tolerance `tol` and the maximum number of iterations `maxiter`. The function creates a few arrays to hold the intermediate and final results and then iterates on the value function until convergence or until the maximum number of iterations is exceeded.

Since the optimal savings choice has to lie on the asset grid, we represent the savings policy function as an *integer* array which contains the *indices* of the optimal asset level instead of the asset level itself.

[5]:
```
def vfi_grid(par, tol=1e-5, maxiter=1000):

    N_a = len(par.grid_a)
    vfun = np.zeros(N_a)
    vfun_upd = np.empty(N_a)
    # index of optimal savings decision (stored in integer array!)
    pfun_ia = np.empty(N_a, dtype=np.uint)

    # pre-compute cash at hand for each asset grid point
    cah = (1 + par.r) * par.grid_a + par.y

    for it in range(maxiter):

        for ia, a in enumerate(par.grid_a):

            # find all values of a' that are feasible, ie. they satisfy
            # the budget constraint
            ia_to = np.where(par.grid_a <= cah[ia])[0]

            # consumption implied by choice a'
            #   c = (1+r)a + y - a'
            cons = cah[ia] - par.grid_a[ia_to]

            # Evaluate "instantaneous" utility
            if par.gamma == 1.0:
                u = np.log(cons)
            else:
                u = (cons**(1.0 - par.gamma) - 1.0) / (1.0 - par.gamma)

            # 'candidate' value for each choice a'
            v_cand = u + par.beta * vfun[ia_to]

            # find the 'candidate' a' which maximises utility
            ia_to_max = np.argmax(v_cand)

            # store results for next iteration
            v_opt = v_cand[ia_to_max]
            vfun_upd[ia] = v_opt
            pfun_ia[ia] = ia_to_max

        diff = np.max(np.abs(vfun - vfun_upd))

        # switch references to value functions for next iteration
        vfun, vfun_upd = vfun_upd, vfun

        if diff < tol:
            msg = f'VFI: Converged after {it:3d} iterations: dV={diff:4.2e}'
            print(msg)
            break
        elif it == 1 or it % 50 == 0:
            msg = f'VFI: Iteration {it:3d}, dV={diff:4.2e}'
            print(msg)
    else:
        msg = f'Did not converge in {it:d} iterations'
        print(msg)

    return vfun, pfun_ia
```

Running the solver

We are now ready to run the VFI. Note that the second return value is an array of *indices* and hence we first need to recover the associated asset levels to get proper savings policy function. The consumption policy function can then be recovered from the budget constraint.

[6]:
```
vfun, pfun_ia = vfi_grid(par)

# Recover savings policy function from optimal asset indices
pfun_a = par.grid_a[pfun_ia]

# Recover consumption policy function from budget constraint
cah = (1.0 + par.r) * par.grid_a + par.y
pfun_c = cah - pfun_a
```

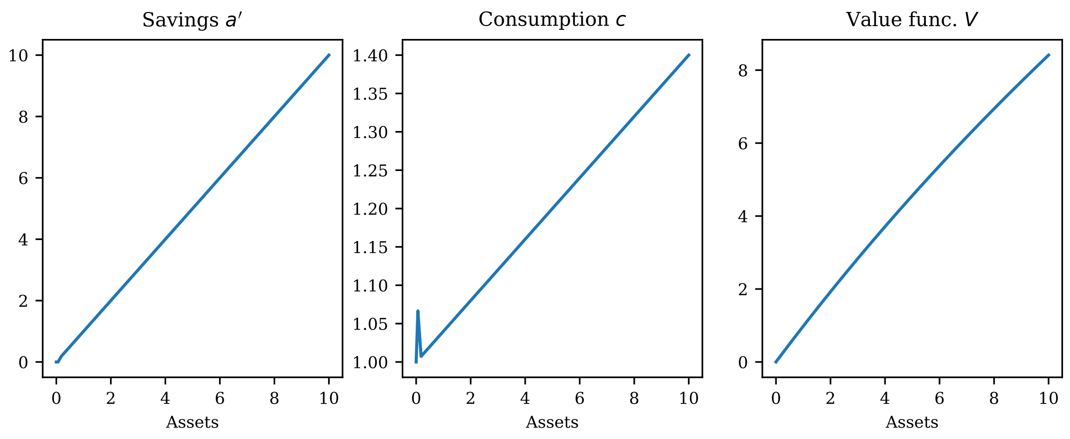
```
VFI: Iteration   0, dV=2.43e+00
VFI: Iteration   1, dV=1.17e+00
VFI: Iteration  50, dV=8.31e-03
VFI: Iteration 100, dV=1.08e-03
VFI: Iteration 150, dV=1.40e-04
VFI: Iteration 200, dV=1.82e-05
VFI: Converged after 215 iterations: dV=9.87e-06
```

Finally, it i always a good idea to visualise the results to get some economic intuition and spot errors.

[7]:
```
fig, axes = plt.subplots(1, 3, sharex=True, sharey=False, figsize=(9, 3.0))

# Plot savings in first column
axes[0].plot(par.grid_a, pfun_a)
axes[0].set_title(r'Savings $a^{\prime}$')
```

```
axes[0].set_xlabel('Assets')

# Plot consumption in second column
axes[1].plot(par.grid_a, pfun_c)
axes[1].set_title(r'Consumption $c$')
axes[1].set_xlabel('Assets')

# Plot value function in third column
axes[2].plot(par.grid_a, vfun)
axes[2].set_title('Value func. $V$')
axes[2].set_xlabel('Assets')
```

```
[7]: Text(0.5, 0, 'Assets')
```



The above plot suggests that there is a jump in the consumption policy function for low assets. There is no economic reason why this should be the case, and in fact this is an undesirable artifact of grid search. Such artifacts may appear for some choices of the asset grid, as is the case here. This is one of the reason why we usually prefer other solution methods that do not exhibit this behaviour, such as VFI with interpolation, which we turn to next.

11.1.3 VFI with interpolation

In the previous section, we saw how grid search can give rise to undesired numerical artifacts that misrepresent the solution. Moreover, unless our grid is extremely dense, the solution is unlikely to satisfy the first-order optimality conditions because the optimal a' need not be on the candidate grid.

A more advanced solution method is to combine VFI with interpolation of the continuation value paired with a maximisation step, which we explore in this section. The advantages of this method are

1. The solution is "exact" in a numerical sense.
2. It is less affected by the curse of dimensionality with many (continuous) choice variables.
3. It is easier to spot mistakes because the resulting policy functions tend to be smooth.

On the other hand, this method

1. Is likely to be slower than grid search; and
2. Is more complex to implement because we need an additional numerical maximisation routine which might require computing (numerical) derivatives.

Outline of the algorithm

To implement VFI with interpolation, we need to modify the original algorithm as follows:

1. Create an asset grid $\Gamma_a = (a_1, \dots, a_{N_a})$.
2. Pick an initial guess V_0 for value function defined on Γ_a.
3. In iteration n, perform the following steps:

   For each asset level a_i at grid point i,

   1. Compute the available resources (cash at hand):
   $$x_i = (1+r)a_i + y$$
   2. Find the maximiser
   $$a^\star = \underset{a' \in [0, x_i]}{\arg\max} \Big\{ u\left(x_i - a'\right) + \beta V_n(a') \Big\}$$
   This step is usually performed using a numerical maximisation (or minimisation) routine.
   3. The value $V^\star$ is then given by
   $$V^\star = u\left(x_{ij} - a^\star\right) + \beta V_n(a^\star)$$
   4. Store the updated value $V_{n+1,i} = V^\star$ and the associated savings policy function.
4. Check for convergence: If $\|V_n - V_{n+1}\| < \epsilon$, for some small tolerance $\epsilon > 0$, exit the algorithm.

Note that we still solve the problem on a grid $(a_1, \dots, a_{N_a})$ but we no longer require households to make savings choices exactly on this grid.

Implementing VFI with interpolation

As before, the full implementation is provided in the Python script files `lectures/unit11/main.py` and `lectures/unit11/VFI.py`.

To perform the maximisation step, we need to define an objective function that can be passed to a minimiser such as SciPy's `minimize_scalar()`. For the problem at hand, we define this objective function as

```
def f_objective(sav, cah, par, f_vfun):
    ...
```

where `sav` is the candidate savings level a', `cah` is the cash-at-hand at the current point a_i, `par` is the `Parameters` instance and `f_vfun` is a callable function that can be used to evaluate the continuation value $V_n(a')$ using interpolation. The implementation of this function looks as follows:

```
[8]: def f_objective(sav, cah, par, f_vfun):

         sav = float(sav)
         if sav < 0.0 or sav >= cah:
             return np.inf

         # Consumption implied by savings level
         cons = cah - sav

         # Continuation value interpolated onto asset grid
         vcont = f_vfun(sav)

         # evaluate "instantaneous" utility
         if par.gamma == 1.0:
             u = np.log(cons)
         else:
```

```
        u = (cons**(1.0 - par.gamma) - 1.0) / (1.0 - par.gamma)

    # Objective evaluated at current savings level
    obj = u + par.beta * vcont

    # We are running a minimiser, return negative of objective value
    return -obj
```

Because we are going to call this objective function from a minimiser, we need to return the negative utility as shown in the last line.

Now that we have the objective function, VFI with interpolation is implemented by the following function:

[9]:
```
from scipy.optimize import minimize_scalar

def vfi_interp(par, tol=1e-5, maxiter=1000):

    N_a = len(par.grid_a)
    vfun = np.zeros(N_a)
    vfun_upd = np.empty(N_a)
    # Optimal savings decision
    pfun_a = np.zeros(N_a)

    for it in range(maxiter):

        # Define function that interpolates continuation value
        f_vfun = lambda x: np.interp(x, par.grid_a, vfun)

        for ia, a in enumerate(par.grid_a):
            # Solve maximization problem at given asset level
            # Cash-at-hand at current asset level
            cah = (1.0 + par.r) * a + par.y
            # Restrict maximisation to following interval:
            bounds = (0.0, cah)
            # Arguments to be passed to objective function
            args = (cah, par, f_vfun)
            # perform maximisation
            res = minimize_scalar(f_objective, bracket=bounds, args=args)

            # Minimiser returns NEGATIVE utility, revert that
            vfun_upd[ia] = - res.fun
            # Store optimal savings a'
            pfun_a[ia] = res.x

        diff = np.max(np.abs(vfun - vfun_upd))

        # switch references to value functions for next iteration
        vfun, vfun_upd = vfun_upd, vfun

        if diff < tol:
            msg = f'VFI: Converged after {it:3d} iterations: dV={diff:4.2e}'
            print(msg)
            break
        elif it == 1 or it % 50 == 0:
            msg = f'VFI: Iteration {it:3d}, dV={diff:4.2e}'
            print(msg)
    else:
        msg = f'Did not converge in {it:d} iterations'
        print(msg)

    return vfun, pfun_a
```

Running the solver

We run the solver in the same way as we did with grid search, except that now the function returns an array of optimal savings *levels*, not the indices on the grid. We therefore no longer need to recover the actual savings policy function.

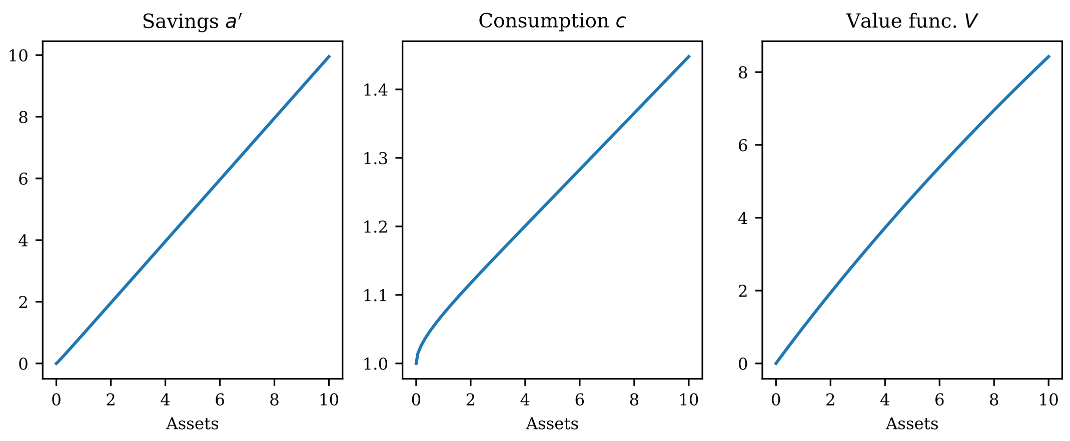
[10]:
```
vfun, pfun_a = vfi_interp(par)

# Recover consumption policy function from budget constraint
cah = (1.0 + par.r) * par.grid_a + par.y
pfun_c = cah - pfun_a
```

```
/home/richard/.conda/envs/python-intro-PGR/lib/python3.10/site-
packages/scipy/optimize/_optimize.py:2417: RuntimeWarning: invalid value
encountered in scalar multiply
  tmp2 = (x - v) * (fx - fw)

VFI: Iteration   0, dV=2.43e+00
VFI: Iteration   1, dV=1.17e+00
VFI: Iteration  50, dV=6.39e-03
VFI: Iteration 100, dV=4.35e-04
VFI: Iteration 150, dV=2.96e-05
VFI: Converged after 170 iterations: dV=9.57e-06
```

Visualising the solution, we see that the artifacts in the consumption policy function are no longer present.

[11]:
```
fig, axes = plt.subplots(1, 3, sharex=True, sharey=False, figsize=(9, 3.0))

# Plot savings in first column
axes[0].plot(par.grid_a, pfun_a)
axes[0].set_title(r'Savings $a^{\prime}$')
axes[0].set_xlabel('Assets')

# Plot consumption in second column
axes[1].plot(par.grid_a, pfun_c)
axes[1].set_title(r'Consumption $c$')
axes[1].set_xlabel('Assets')

# Plot value function in third column
axes[2].plot(par.grid_a, vfun)
axes[2].set_title('Value func. $V$')
axes[2].set_xlabel('Assets')
```

[11]:
```
Text(0.5, 0, 'Assets')
```



11.2 VFI with stochastic labour income

11.2.1 Modelling stochastic labour income

So far, we assumed that labour income y was deterministic and constant. In more realistic heterogeneous-agent models in macroeconomics and household finance, we instead model labour income as stochastic and thus risky. One frequent assumption is that labour income follows an AR(1) process in logs, i.e.,

$$\begin{aligned} \log y_{t+1} &= \rho \log y_t + \nu_{t+1} \\ \nu_{t+1} &\overset{\text{iid}}{\sim} N(0, \sigma^2) \end{aligned}$$

The corresponding household problem is then given by

$$\begin{aligned} V(y,a) = \max_{c,\,a'} &\left\{ u(c) + \beta \mathbb{E}\left[V(y',a') \mid y \right] \right\} \\ \text{s.t.} \quad & c + a' = (1+r)a + y \\ & c \geq 0,\; a' \geq 0 \end{aligned}$$

where we added the additional state variable y and need to take expectations over future realisations of y'.

There are two commonly-used approaches to incorporate such an income process into our household problem:

1. Assume that y is a continuous state variable and use Gauss-Hermite quadrature to compute expectations.
2. Discretize y to take on a few selected values and model it as a Markov chain on the discretised state space.

We will following the second approach in this unit. Two common algorithms to “convert” an AR(1) into a Markov chain are the Tauchen and the Rouwenhorst methods. The latter seems to be preferable when dealing with highly persistent processes such as labour income. The discretised labour income process with N_y grid points is then given by the states $\Gamma_y = (y_1, \dots, y_{N_y})$ and the transition matrix Π_y with typical element

$$\Pi_y(i,j) = \Pr\left[y_{t+1} = y_j \mid y_t = y_i \right] .$$

We use the function `rouwenhorst()` defined below to perform this mapping for us. Its arguments are the number of nodes of the discretized state space `n`, the mean of the AR(1) process `mu`, the autocorrelation parameter `rho` and the conditional standard deviation `sigma` (see the file `lectures/unit11/markov.py` for a complete implementation which includes error checking of input arguments). There is no need to understand the details of this function as we are only interested in using its return values: the state space stored in the vector `z` and the transition matrix `Pi`.

[12]:
```python
import numpy as np

def rouwenhorst(n, mu, rho, sigma):

    p = (1+rho)/2
    Pi = np.array([[p, 1-p], [1-p, p]])

    for i in range(Pi.shape[0], n):
        tmp = np.pad(Pi, 1, mode='constant', constant_values=0)
        Pi = p * tmp[1:, 1:] + (1-p) * tmp[1:, :-1] + \
             (1-p) * tmp[:-1, 1:] + p * tmp[:-1, :-1]
        Pi[1:-1, :] /= 2

    fi = np.sqrt(n-1) * sigma / np.sqrt(1 - rho ** 2)
    z = np.linspace(-fi, fi, n) + mu

    return z, Pi
```

Model parameters

Since the household problem with stochastic labour income has a few more parameters, we redefine the `Parameters` class from earlier to include these. The additional parameters are listed in the table below. The values for the AR(1) labour income process are standard and taken from papers such as Aiyagari (1994).

Parameter	Description	Value
β	Discount factor	0.96
γ	Coef. of relative risk aversion	2
r	Interest rate	0.04
ρ	Autocorrelation of labour income	0.95
σ	Conditional std. dev. of labour income	0.20
N_y	Number of states for Markov chain	3

The updated definition of `Parameters` now looks as follows:

[13]:
```
@dataclass
class Parameters:
    """
    Define object to store model parameters and their default values
    """
    beta = 0.96          # Discount factor
    gamma = 1.0          # Risk aversion
    r = 0.04             # Interest rate
    rho = 0.95           # Autocorrelation of log labour income
    sigma = 0.20         # Conditional standard deviation of log labour income
    grid_a = None        # Asset grid (to be created)
    grid_y = None        # Discretised labour income grid (to be created)
    tm_y = None          # Labour transition matrix (to be created)
```

We can use the above function to create a discretised representation of the risky labour income. For illustrative purposes we choose to discretise y to only three points which would be considered too few for realistic models.

[14]:
```
par = Parameters()

# Number of labour grid points
N_y = 3

# Discretise labour income process (assume zero mean in logs)
states, tm_y = rouwenhorst(N_y, mu=0.0, rho=par.rho, sigma=par.sigma)

# State space in levels
grid_y = np.exp(states)

# Store labour grid and transition matrix
par.grid_y = grid_y
par.tm_y = tm_y
```

Note that because stochastic income is log-normal and has zero mean in logs, due to the properties of the log-normal distribution the average income in *levels* in the cross-section of households is given by

$$\mathbb{E}[y] = \exp\left(\frac{1}{2}\sigma_v^2\right) > 1$$

This is often undesirable as we'd like to normalise average income in the economy to unity since this makes interpreting magnitudes easier. We can achieve this by computing the ergodic (stationary) distribution of labour using the following function (see also `lectures/unit11/markov.py` for a complete implementation). Again, there is no need to understand what this function does, we are only interested in the stationary distribution it returns in the vector `mu`.

[15]:
```
import numpy.linalg

def markov_ergodic_dist(transm):
    transm = transm.transpose()
    m = transm - np.identity(transm.shape[0])
    m[-1] = 1
    m = np.linalg.inv(m)
    mu = np.ascontiguousarray(m[:, -1])
    assert np.abs(np.sum(mu) - 1) < 1e-9
    mu /= np.sum(mu)
    return mu
```

We use this function to compute the ergodic distribution `edist` implied by the Markov chain transition matrix. We can then use this distribution to compute expectation $\mathbb{E}[y]$ and we normalise the state space by this value.

[16]:
```
# Ergodic distribution of labour income
edist = markov_ergodic_dist(tm_y)

# Non-normalised mean of labour income
mean = np.dot(edist, par.grid_y)

# Normalise states such that unconditional expectation is 1.0
par.grid_y /= mean
```

Finally, we complete the setup of the problem by re-creating the asset grid in the same way we did above.

[17]:
```
# Start + end point for asset grid
a_min = 0.0
a_max = 10.0
# Number of grid points
N_a = 30

# Create asset grid with more points at the beginning
grid_a  = a_min + (a_max - a_min) * np.linspace(0.0, 1.0, N_a)**1.5

# Store asset grid in Parameters object
par.grid_a = grid_a
```

11.2.2 VFI with grid search

Before turning to the actual implementation, it is instructive to look at the modified grid search algorithm for the case of risky labour income. The main difference is that we now have an additional state variable y and need to take care of expectations.

Outline of the algorithm

1. Create an asset grid $\Gamma_a = (a_1, \dots, a_{N_a})$.
2. Create a discretised labour income process with states $\Gamma_y = (y_1, \dots, y_{N_y})$ and transition matrix Π_y.
3. Pick an initial guess V_0 for the value function defined on $\Gamma_y \times \Gamma_a$.
4. In iteration n, perform the following steps:

   For each labour income y_i at index i,

1. Compute the expectation over labour income realisations y' given y_i,

   $$\widetilde{V}(a') = \mathbb{E}\left[V_n(y',a') \mid y_i\right]$$

   using the i-th row of the transition matrix Π_y.
2. For each asset level a_j at grid point j,
   1. Find all feasible next-period asset levels $a_k \in \Gamma_a$ that satisfy the budget constraint

      $$a_k \leq (1+r)a_j + y_i$$

   2. For each k, compute consumption

      $$c_k = (1+r)a_j + y_i - a_k$$

   3. For each k, compute utility

      $$U_k = u(c_k) + \beta\widetilde{V}(a_k)$$

   4. Find the index ℓ that maximises the above expression:

      $$\ell = \arg\max_k \left\{u(c_k) + \beta\widetilde{V}(a_k)\right\}$$

   5. Set $V_{n+1,ij} = U_\ell$ and store ℓ as the optimal choice at (i,j)

5. Check for convergence: If $\|V_n - V_{n+1}\| < \epsilon$, for some small tolerance $\epsilon > 0$, exit the algorithm.

Implementing VFI with grid search

The following code implements VFI with risky labour income using grid search (the code can also be found in `lectures/unit11/VFI_risk.py`).

[18]:
```python
def vfi_grid(par, tol=1e-5, maxiter=1000):

    N_a, N_y = len(par.grid_a), len(par.grid_y)
    shape = (N_y, N_a)
    vfun = np.zeros(shape)
    vfun_upd = np.empty(shape)
    # index of optimal savings decision
    pfun_ia = np.empty(shape, dtype=np.uint)

    # pre-compute cash at hand for each (asset, labour) grid point
    cah = (1 + par.r) * par.grid_a[None] + par.grid_y[:,None]

    for it in range(maxiter):

        # Compute expected continuation value E[V(y',a')|y] for each (y,a')
        EV = np.dot(par.tm_y, vfun)

        for iy in range(N_y):
            for ia, a in enumerate(par.grid_a):

                # find all values of a' that are feasible, ie. they satisfy
                # the budget constraint
                ia_to = np.where(par.grid_a <= cah[iy, ia])[0]

                # consumption implied by choice a'
                #   c = (1+r)a + y - a'
                cons = cah[iy, ia] - par.grid_a[ia_to]

                # Evaluate "instantaneous" utility
                if par.gamma == 1.0:
                    u = np.log(cons)
                else:
                    u = (cons**(1.0 - par.gamma) - 1.0) / (1.0 - par.gamma)

                # 'candidate' value for each choice a'
                v_cand = u + par.beta * EV[iy, ia_to]

                # find the 'candidate' a' which maximizes utility
                ia_to_max = np.argmax(v_cand)

                # store results for next iteration
                v_opt = v_cand[ia_to_max]
                vfun_upd[iy, ia] = v_opt
                pfun_ia[iy, ia] = ia_to_max

        diff = np.max(np.abs(vfun - vfun_upd))

        # switch references to value functions for next iteration
        vfun, vfun_upd = vfun_upd, vfun

        if diff < tol:
            msg = f'VFI: Converged after {it:3d} iterations: dV={diff:4.2e}'
            print(msg)
            break
        elif it == 1 or it % 50 == 0:
            msg = f'VFI: Iteration {it:3d}, dV={diff:4.2e}'
            print(msg)
    else:
        msg = f'Did not converge in {it:d} iterations'
        print(msg)

    return vfun, pfun_ia
```

We can now run and plot the solution as we did in the case of deterministic income.

[19]:
```
vfun, pfun_ia = vfi_grid(par)

# Recover savings policy function from optimal asset indices
pfun_a = par.grid_a[pfun_ia]

# Recover consumption policy function from budget constraint
cah = (1.0 + par.r) * par.grid_a + par.grid_y[:, None]
pfun_c = cah - pfun_a
```

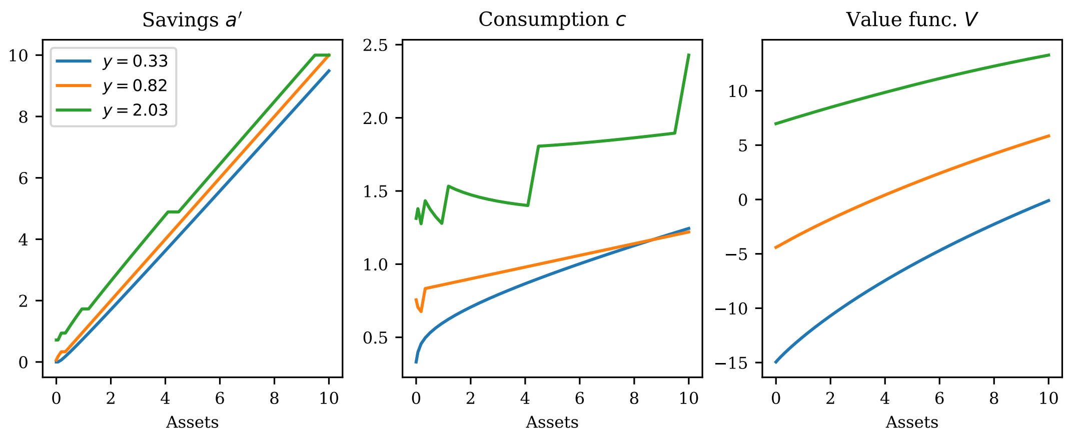
```
VFI: Iteration   0, dV=2.52e+00
VFI: Iteration   1, dV=1.38e+00
VFI: Iteration  50, dV=3.02e-02
VFI: Iteration 100, dV=2.07e-03
VFI: Iteration 150, dV=2.29e-04
VFI: Iteration 200, dV=2.87e-05
VFI: Converged after 226 iterations: dV=9.88e-06
```

Finally, we plot the solution for all levels of labour income. Note that we need to transpose the result arrays because Matplotlib's `plot()` expects input arrays to have the same number of elements along the first dimension.

[20]:
```
fig, axes = plt.subplots(1, 3, sharex=True, sharey=False, figsize=(9, 3.0))

# Plot savings in first column
axes[0].plot(par.grid_a, pfun_a.T)
axes[0].set_title(r'Savings $a^{\prime}$')
axes[0].set_xlabel('Assets')
# Insert legend
labels = [f'$y={y:.2f}$' for y in par.grid_y]
```

```
axes[0].legend(labels, loc='upper left')

# Plot consumption in second column
axes[1].plot(par.grid_a, pfun_c.T)
axes[1].set_title(r'Consumption $c$')
axes[1].set_xlabel('Assets')

# Plot value function in third column
axes[2].plot(par.grid_a, vfun.T)
axes[2].set_title('Value func. $V$')
axes[2].set_xlabel('Assets')
```

```
[20]: Text(0.5, 0, 'Assets')
```



With stochastic labour income, the numerical artifacts produced by grid search are even more pronounced which speaks against using this method for this type of problem, if possible.

11.2.3 VFI with interpolation

We can adjust the interpolation algorithm to deal with stochastic labour income in the same way as we augmented the grid search.

Outline of the algorithm

To implement VFI with interpolation, we need to modify the original algorithm as follows:

1. Create an asset grid $\Gamma_a = (a_1, \dots, a_{N_a})$.
2. Create a discretised labour income process with states $\Gamma_y = (y_1, \dots, y_{N_y})$ and transition matrix Π_y.
3. Pick an initial guess V_0 for value function defined on $\Gamma_y \times \Gamma_a$.
4. In iteration n, perform the following steps:

   For each labour income y_i at index i,

   1. Compute the expectation over labour income realisations y' given y_i,

      $$\widetilde{V}(a') = \mathbb{E}\left[V_n(y', a') \mid y_i\right]$$

      using the i-th row of the transition matrix Π_y.
   2. For each asset level a_j at grid point j,
      1. Compute the available resources (cash at hand):

         $$x_{ij} = (1 + r)a_i + y_i$$

2. Find the maximiser
$$a^\star = \underset{a' \in [0, x_{ij}]}{\arg\max} \Big\{ u\left(x_{ij} - a'\right) + \beta \widetilde{V}(a') \Big\}$$
This step is usually performed using a numerical maximisation (or minimisation) routine.
3. The value $V^\star$ is then given by
$$V^\star = u\left(x_{ij} - a^\star\right) + \beta \widetilde{V}(a^\star)$$
4. Store the updated value $V_{n+1,ij} = V^\star$ and the associated savings policy function.

5. Check for convergence: If $\|V_n - V_{n+1}\| < \epsilon$, for some small tolerance $\epsilon > 0$, exit the algorithm.

Implementing VFI with interpolation

The following code implements VFI with risky labour income using interpolation (the code can also be found in `lectures/unit11/VFI_risk.py`).

Note that the objective function `f_objective()` remains unchanged from before because we adapted the argument `f_vfun` to interpolate over the pre-computed expected value of $\widetilde{V}(a')$ instead of $V_n(a')$ as in the original implementation.

[21]:
```python
def vfi_interp(par, tol=1e-5, maxiter=1000):

    N_a, N_y = len(par.grid_a), len(par.grid_y)
    shape = (N_y, N_a)
    vfun = np.zeros(shape)
    vfun_upd = np.empty(shape)
    # Optimal savings decision
    pfun_a = np.zeros(shape)

    for it in range(maxiter):

        # Compute expected continuation value E[V(y',a')|y] for each (y,a')
        EV = np.dot(par.tm_y, vfun)

        for iy, y in enumerate(par.grid_y):

            # function to interpolate continuation value
            f_vfun = lambda x: np.interp(x, par.grid_a, EV[iy])

            for ia, a in enumerate(par.grid_a):
                # Solve maximization problem at given asset level
                # Cash-at-hand at current asset level
                cah = (1.0 + par.r) * a + y
                # Restrict maximisation to following interval:
                bounds = (0.0, cah)
                # Arguments to be passed to objective function
                args = (cah, par, f_vfun)
                # perform maximisation
                res = minimize_scalar(f_objective, bracket=bounds, args=args)

                # Minimiser returns NEGATIVE utility, revert that
                v_opt = - res.fun
                sav_opt = float(res.x)

                vfun_upd[iy, ia] = v_opt
                pfun_a[iy, ia] = sav_opt

        diff = np.max(np.abs(vfun - vfun_upd))

        # switch references to value functions for next iteration
        vfun, vfun_upd = vfun_upd, vfun

        if diff < tol:
            msg = f'VFI: Converged after {it:3d} iterations: dV={diff:4.2e}'
            print(msg)
            break
        elif it == 1 or it % 50 == 0:
            msg = f'VFI: Iteration {it:3d}, dV={diff:4.2e}'
            print(msg)
    else:
        msg = f'Did not converge in {it:d} iterations'
        print(msg)

    return vfun, pfun_a
```

The following code is used to perform the VFI. Note that once as in the deterministic setting, the savings policy function returns savings *levels*, even though now all return values are 2-dimensional arrays.

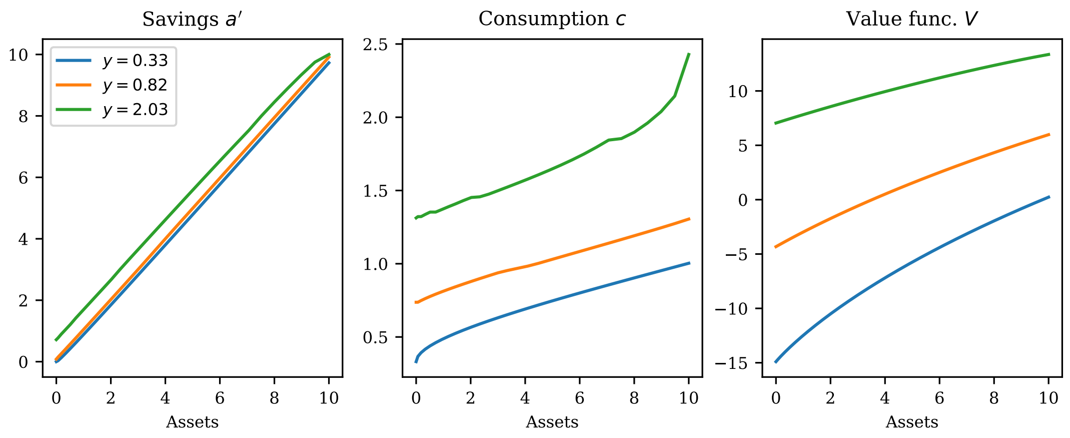
[22]:
```
vfun, pfun_a = vfi_interp(par)

# Recover consumption policy function from budget constraint
cah = (1.0 + par.r) * par.grid_a + par.grid_y[:, None]
pfun_c = cah - pfun_a
```

```
/home/richard/.conda/envs/python-intro-PGR/lib/python3.10/site-
packages/scipy/optimize/_optimize.py:2417: RuntimeWarning: invalid value
encountered in scalar multiply
  tmp2 = (x - v) * (fx - fw)

VFI: Iteration   0, dV=2.52e+00
VFI: Iteration   1, dV=1.38e+00
VFI: Iteration  50, dV=2.95e-02
VFI: Iteration 100, dV=1.95e-03
VFI: Iteration 150, dV=2.16e-04
VFI: Iteration 200, dV=2.73e-05
VFI: Converged after 225 iterations: dV=9.80e-06
```

We use the same code as above to visualise the results:

[23]:
```
fig, axes = plt.subplots(1, 3, sharex=True, sharey=False, figsize=(9, 3.0))

# Plot savings in first column
axes[0].plot(par.grid_a, pfun_a.T)
axes[0].set_title(r'Savings $a^{\prime}$')
axes[0].set_xlabel('Assets')
# Insert legend
labels = [f'$y={y:.2f}$' for y in par.grid_y]
axes[0].legend(labels, loc='upper left')

# Plot consumption in second column
axes[1].plot(par.grid_a, pfun_c.T)
axes[1].set_title(r'Consumption $c$')
axes[1].set_xlabel('Assets')

# Plot value function in third column
axes[2].plot(par.grid_a, vfun.T)
axes[2].set_title('Value func. $V$')
axes[2].set_xlabel('Assets')
```

[23]:
```
Text(0.5, 0, 'Assets')
```



The policy functions are much smoother compared to those produced by grid search. However, as you can see there is another artifact of our numerical implementation that did not stand out as much earlier: the consumption policy for the highest labour state is suspiciously upward-sloping towards the end! This is a consequence of not permitting extrapolation of the continuation value, and hence any savings that would take the household beyond the grid upper bound are wasted. It is therefore optimal to consume all such "excess" savings, which gives rise to the increase in consumption at the right end of the asset grid. We could address this problem by allowing extrapolation, but linear extrapolation of value functions is undesirable for other reasons.

11.3 EGM with stochastic labour income

While value function iteration is a robust and long-established method to solve dynamic problems, it tends to be slow when combined with interpolation, while it produces numerical artifacts when combined with grid search. The endogenous grid-point method (EGM) due to Carroll (2005) addresses both the issue of speed and at the same time produces smooth and accurate solutions.

To fix ideas, let's revisit the stochastic labour income we studied above:

$$\begin{aligned} V(y,a) &= \max_{c,a'} \Big\{ u(c) + \beta \mathbb{E}\big[V(y',a') \mid y \big] \Big\} \\ \text{s.t.} \quad & c + a' = (1+r)a + y \\ & c \geq 0,\ a' \geq 0 \end{aligned}$$

The optimality conditions for this problem are the standard Euler equation combined with the budget constraint:

$$\begin{aligned} c^{-\gamma} &= \beta(1+r)\mathbb{E}\left[(c')^{-\gamma} \middle| y \right] \\ c + a' &= (1+r)a + y \end{aligned}$$

So far, we haven't used the Euler equation at all, but it can be leveraged to invert the problem and speed up the solution. Suppose that instead of trying to determine the optimal savings level a' for an exogenously given beginning-of-period asset level a, we ask the following: if we exogenously impose that the household optimally saves a', what is the beginning-of-period asset level a that rationalizes this savings choice given the household's optimality conditions?

To answer this question, assume we have a guess for the consumption policy function $C(y,a)$; for example, we could initially impose that the household chooses to consume everything, i.e., $C(y,a) = a$. For a given a', from the Euler equation we know that

$$(c^*)^{-\gamma} = \beta(1+r)\mathbb{E}\left[C(y',a')^{-\gamma} \mid y \right]$$

where we use the notation c^* to indicate that this consumption level is a function of the exogenously imposed a'. Consequently, optimal consumption today must be given by

$$c^* = \Big(\beta(1+r)\mathbb{E}\left[C(y',a')^{-\gamma} \mid y \right] \Big)^{-\frac{1}{\gamma}}$$

With c^* and a' in hand, the only unknown in the budget constraint are the beginning-of-period assets a^* which we again denote using $*$ to stress that it is a function of the exogenous a':

$$a^* = \frac{c^* + a' - y}{1+r}$$

Note that the beginning-of-period assets a^* thus arise *endogenously* as a function of the exogenously-imposed optimal savings a', hence the name *endogenous* grid-point method.

To summarize, every point on the exogenous savings $\Gamma_{a'} = \{a'_1, a'_2, \dots, a'_{N_{a'}}\}$ gives rise to an endogenous tuple of values (c^*, a^*). After applying this procedure to all $N_{a'}$ exogenous savings levels, we have the collection of values $(c_i^*)_{i=1}^{N_{a'}}$ and $(a_i^*)_{i=1}^{N_{a'}}$. We are usually not interested in characterising the solution to the household problem as a function of the endogenous grid of $\{a_1^*, a_2^*, \dots\}$, so the final step is to use these points to interpolate the consumption policy function back onto our standard beginning-of-period asset grid $\Gamma_a = \{a_1, a_2, \dots, a_{N_a}\}$. This yields an updated consumption policy function $C(y, a)$ mapping a into c which we can use in the next iteration.

Outline of the algorithm

To solve the household problem with stochastic labour income with EGM, we proceed as follows:

1. Create an asset grid $\Gamma_a = (a_1, \dots, a_{N_a})$ and optionally and exogenous savings grid $\Gamma_{a'} = (a'_1, \dots, a'_{N_{a'}})$. For simplicity we use the same grid for both purposes.
2. Create a discretised labour income process with states $\Gamma_y = (y_1, \dots, y_{N_y})$ and transition matrix Π_y.
3. Pick an initial guess $C_0(y, a)$ for the consumption policy function defined on $\Gamma_y \times \Gamma_a$. One possibility is to assume that $C_0(y, a) = a$.
4. In iteration n, perform the following steps:

   For each labour income y_i at index i,

   1. Compute the vector of *expected marginal utilities* for all points a'_j on the exogenous savings grid with typical element
   $$m_{ij} \equiv \mathbb{E}\left[C_n\left(y', a'_j\right)^{-\gamma} \;\middle|\; y_i\right]$$
   2. Invert the Euler equation to get a vector of today's consumption levels,
   $$c_{ij}^* = \left[\beta(1+r)m_{ij}\right]^{-\frac{1}{\gamma}}$$
   3. Use the budget constraint to find the vector of required beginning-of-period asset levels
   $$a_{ij}^* = \frac{c_{ij}^* + a'_j - y_i}{1+r}$$
   4. Use the collection of values $(a_{ij})_{j=1}^{N_{a'}}$ and $(c_{ij})_{j=1}^{N_{a'}}$ to interpolate optimal consumption onto the beginning-of-period asset grid Γ_a. This yields an updated guess for $C_{n+1}(y_i, a)$.
   5. Finally, note that the Euler equation only holds for *interior* solutions but does not hold when the household is borrowing constraint. The endogenous grid point a_{i1}^* associated with the first savings grid-point $a'_1 = 0$ exactly identifies the beginning-of-period asset level at which the household is no longer borrowing constraint. For all asset levels below this point, we know that the household will choose not to save, so we set
   $$C_{n+1}(y_i, a) = (1+r)a + y_i \quad \forall a < a_{i1}^*$$
5. Check for convergence: If $\|C_n - C_{n+1}\| < \epsilon$, for some small tolerance $\epsilon > 0$, exit the algorithm.

Implementing EGM with stochastic labour income

The `lectures/unit11` folder contains two EGM implementations,

1. `lectures/unit11/EGM.py`: implements infinite-horizon EGM for the problem with deterministic labour income;
2. `lectures/unit11/EGM_risk.py`: implements infinite-horizon EGM for the problem with stochastic labour income.

Setting up the problem is identical to the case of VFI, see `lectures/unit11/main.py` for the problem with deterministic labour and `lectures/unit11/main_risk.py` for the stochastic version. For completeness, the implementation is shown below. Note that here we use SciPy's interpolation routine `interp1d()` since it supports extrapolation, unlike its NumPy counterpart `np.interp()`.

[24]:
```python
from scipy.interpolate import interp1d

def egm(par, tol=1.0e-8, maxiter=10000):

    N_a, N_y = len(par.grid_a), len(par.grid_y)
    shape = (N_y, N_a)

    # Cash-at-hand at every asset/savings level
    cah = (1.0 + par.r) * par.grid_a[None] + par.grid_y[:, None]

    # Initial guess for consumption policy function
    pfun_c = np.copy(cah)
    pfun_c_upd = np.zeros(shape)

    # Extract parameters from par object
    beta, gamma, r = par.beta, par.gamma, par.r

    for it in range(maxiter):

        # Iterate over all labour income states
        for iy, y in enumerate(par.grid_y):

            # Expected marginal utility tomorrow
            mu = np.dot(par.tm_y[iy], pfun_c**(-gamma))
            # Compute right-hand side of Euler equation (EE)
            ee_rhs = beta * (1.0 + r) * mu

            # Invert EE to get consumption as a function of savings today
            cons_sav = ee_rhs**(-1.0/gamma)

            # Use budget constraint to get beginning-of-period assets
            assets_sav = (cons_sav + par.grid_a - y) / (1.0 + r)

            # Interpolate back onto exogenous savings grid
            f_cons = interp1d(
                assets_sav, cons_sav,
                copy=False, assume_sorted=True,
                bounds_error=False, fill_value='extrapolate'
            )
            pfun_c_upd[iy] = f_cons(par.grid_a)

            # Fix consumption in region where HH does not save
            amin = assets_sav[0]
            idx = np.where(par.grid_a <= amin)[0]
            # HH consumes entire cash-at-hand
            pfun_c_upd[iy, idx] = cah[iy, idx]

        # Make sure that consumption policy satisfies constraints
        assert np.all(pfun_c_upd >= 0.0) and np.all(pfun_c_upd <= cah)
        # Compute max. absolute difference to policy function from previous
        # iteration.
        diff = np.max(np.abs(pfun_c - pfun_c_upd))

        # switch references to policy functions for next iteration
        pfun_c, pfun_c_upd = pfun_c_upd, pfun_c

        if diff < tol:
            # Convergence achieved, exit loop
            msg = f'EGM: Converged after {it:d} iterations: d(c)={diff:4.2e}'
            print(msg)
            break
        elif it == 1 or it % 50 == 0:
            msg = f'EGM: Iteration {it:4d}, dV={diff:4.2e}'
            print(msg)
    else:
        msg = f'Did not converge in {it:d} iterations'
        print(msg)

    pfun_a = cah - pfun_c

    return pfun_a, pfun_c
```

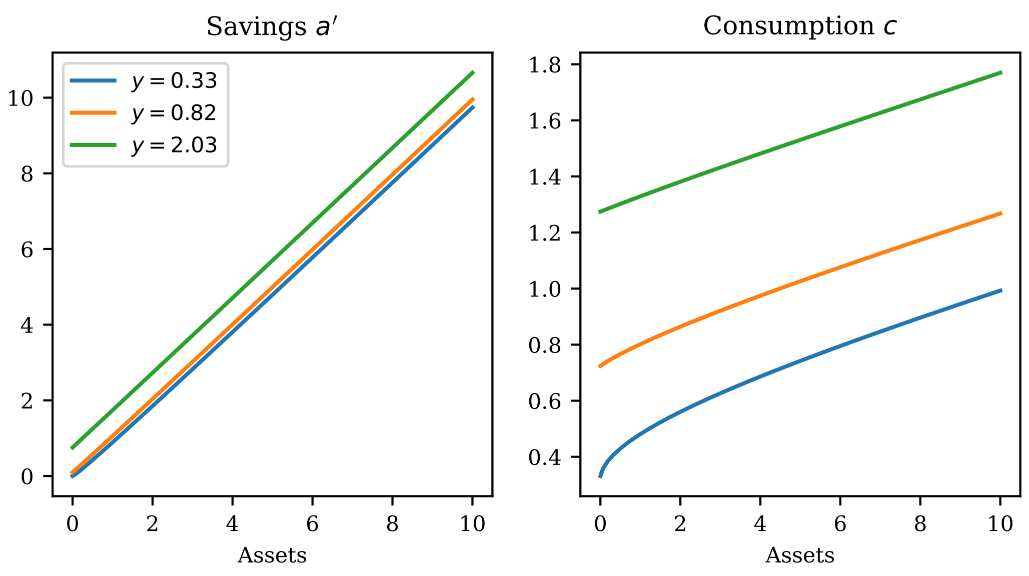
The following code runs and plots the results. It assumes that the parameters have been initialised as shown in the previous sections. Note that the EGM method does not return the value function since it is not needed for the algorithm. Once the policy functions have been computed, it is however possible to construction the value function from these by iteration (without having to perform maximisation since we already know the optimal choices).

[25]:
```
# Run EGM, store savings and consumption policy functions.
pfun_a, pfun_c = egm(par)
```

```
EGM: Iteration    0, dV=5.13e+00
EGM: Iteration    1, dV=1.73e+00
EGM: Iteration   50, dV=4.54e-03
EGM: Iteration  100, dV=3.29e-04
EGM: Iteration  150, dV=2.47e-05
EGM: Iteration  200, dV=1.88e-06
EGM: Iteration  250, dV=1.44e-07
EGM: Iteration  300, dV=1.10e-08
EGM: Converged after 302 iterations: d(c)=9.95e-09
```

[26]:
```
# Visualise the EGM solution
fig, axes = plt.subplots(1, 2, sharex=True, sharey=False, figsize=(6.5, 3.0))

# Plot savings in first column
axes[0].plot(par.grid_a, pfun_a.T)
axes[0].set_title(r'Savings $a^{\prime}$')
axes[0].set_xlabel('Assets')
# Insert legend
labels = [f'$y={y:.2f}$' for y in par.grid_y]
axes[0].legend(labels, loc='upper left')

# Plot consumption in second column
axes[1].plot(par.grid_a, pfun_c.T)
axes[1].set_title(r'Consumption $c$')
axes[1].set_xlabel('Assets')
```

[26]:
```
Text(0.5, 0, 'Assets')
```



We close this section with a comment on the relative run time of the methods we have studied above to solve the problem with stochastic labour income. The following table shows the relative run times of each solution method, with the run time for VFI with grid search normalised to one. The problem is solved on 50 asset grid points and risky labour income is discretised to three grid points.

Method	Relative run time
VFI + grid search	1.00
VFI + linear interp.	22.01
EGM	0.21

As you can see, EGM is not only more accurate but also 5 times faster than VFI with grid search, and 100 times faster than VFI with linear interpolation which on top yields worse results.

12 Error handling

In this unit we will briefly look at error handling in Python. The Python approach to error handling is "to ask for forgiveness rather than for permission." This means that when writing Python code, we frequently don't check whether some data satisfies certain requirements, but we instead attempt to clean up once something does not work as expected.

12.1 Exceptions

If something goes wrong in a function, we in principle have two options to communicate the error to the caller:

1. We can return some special value (a status code or error flag) that signals when something fails.

   This approach is quite inelegant, since error codes can overlap with the actual result a function would return in the absence of error. For this reasons, functions need to implement two different return values and reserve one for the status code.

   In Python, this could look like this:

   ```
   def func(x):
       # process x
       # Two return values: actual result and error flag
       return result, flag
   ```

2. We can use so-called exceptions for error handling. This is the approach taken by almost all modern languages such as Java, C++ and also Python (see here for the official documentation on error and exception handling).

   Exceptions provide means to communicate errors that are completely independent of regular return values. Furthermore, exceptions propagate along the entire call stack: If we call `func1()`, which in turn calls `func2()`, and an error occurs in `func2()`, there is no need to handle this error in `func1()`: the exception will automatically be propagated to the caller of `func1()`.

12.1.1 Common exceptions

We have already encountered numerous exceptions throughout this course, but so far we did not know how to handle them other than fixing the code that produced the exception.

There are numerous exceptions in Python, see here for a list of built-in ones. We provide a few examples of exceptions that you are most likely to encounter below.

Examples:

Trying to access an element in a collection outside of the permissible ranger produces an `IndexError`.

```
[1]: # access to out-of-bounds index in a collection
     items = 1, 2, 3
     items[5]
```

```
IndexError: tuple index out of range
```

Retrieving a non-existent key in a dictionary raises another type of exception, a `KeyError`.

```
[2]: # Access non-existant dictionary key
     dct = {'language': 'Python', 'version': 3.8}
     dct['course']
```

```
KeyError: 'course'
```

Mistakenly trying to access a non-existent attribute will trigger an `AttributeError`:

```
[3]: value = 1.0
     value.shape
```

```
AttributeError: 'float' object has no attribute 'shape'
```

When we try to apply an operation to data that does not support that particular operation, we get a `TypeError`:

```
[4]: items = 1, 2, 3
     items + 1
```

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

Division by zero also triggers an exception of type `ZeroDivisionError`:

```
[5]: 1/0
```

```
ZeroDivisionError: division by zero
```

Attempting to import a module or symbol from within a module that does not exist raises an `ImportError`:

```
[6]: from numpy import function_that_does_not_exist
```

```
ImportError: cannot import name 'function_that_does_not_exist' from 'numpy' (/home/richard/.
  ↪conda/envs/py3-default/lib/python3.10/site-packages/numpy/__init__.py)
```

Performing an operation on arrays of non-conforming shape produces a `ValueError`:

```
[7]: import numpy as np

     a = np.arange(3)
     b = np.arange(2)
     a + b
```

```
ValueError: operands could not be broadcast together with shapes (3,) (2,)
```

Trying to open a non-existing file will raise an `FileNotFoundError`.

[8]:
```
open('file_does_not_exists.txt', 'rt')
```

```
FileNotFoundError: [Errno 2] No such file or directory: 'file_does_not_exists.txt'
```

In other cases, for example when using NumPy's `loadtxt`, trying to open a non-existing file will instead raise a `OSError`.

[9]:
```
import numpy as np

data = np.loadtxt('path/to/nonexisting/file.txt')
```

```
OSError: path/to/nonexisting/file.txt not found.
```

12.2 Handling errors

As you just saw, there are numerous types of exceptions raised by Python libraries we use every day. We can handle these in two ways:

1. Avoid errors before they arise.
2. Catch exceptions once they arise in special exception-handling blocks.

12.2.1 Avoiding errors

We could have avoided almost all of the above exception if we had surrounded them with `if` statements and checked whether an operation could actually be performed.

This, however, is usually not the way we write Python code, unless we are implementing library functions that need to work in situations over which we have little control. We certainly don't want to clutter "regular" code with `if` statements everywhere. There are other ways to avoid errors.

Examples:

Returning to the dictionary example, we could write something like this:

[10]:
```
# Access non-existant dictionary key
dct = {'language': 'Python', 'version': 3.10}
if 'course' in dct:
    print(dct['course'])
```

However, if we have a default value that should be used whenever a key is not present, we can more elegantly use the `get()` method which accepts a default value. No `if`'s needed:

[11]:
```
# access non-existing key
dct.get('course', 'Default programming language')
```

[11]:
```
'Default programming language'
```

If a key does exist, the default will of course be ignored:

[12]:
```
# access existing key
dct.get('language', 'Default programming language')
```

[12]:
```
'Python'
```

Another easily avoidable exception is the `IndexError`. There is hardly ever a reason to attempt retrieving elements at arbitrary indices. Usually, we first check the size of a collection:

[13]:
```
items = 1, 2, 3

# Assume idx was passed as an argument to a function
# so we need to handle unforeseen cases
idx = 1000

# Enforce valid upper bound in case the index is
# out of bounds.
items[min(idx, len(items) - 1)]
```

[13]:
```
3
```

When operating on NumPy arrays, we frequently have to retrieve their dimensions first, so there is no risk of accessing an invalid position:

[14]:
```
import numpy as np

mat = np.arange(6).reshape(2, 3)

# Retrieve array dimensions
nrow, ncol = mat.shape

# Loop makes sure to never step out of bounds
for i in range(nrow):
    for j in range(ncol):
        print(mat[i, j])
```

```
0
1
2
3
4
5
```

There are also many helper routines that allow for “robust” programming. Imagine we want a function that returns the element at position `[0,0]`:

[15]:
```
def get_elem(x):
    return x[0,0]
```

Calling this on a matrix works as intended:

[16]:
```
get_elem(np.ones((2,2)))
```

[16]:
```
1.0
```

But what if we pass a nested list or tuple?

[17]:
```
get_elem([[1,2], [3,4]])
```

```
TypeError: list indices must be integers or slices, not tuple
```

With very little effort, we can make this function more robust by using `np.atleast_2d()` which ensures that its result is at least a 2-dimensional NumPy array (it returns higher-dimensional arrays unmodified):

```
[18]: import numpy as np

      def get_elem(x):
          x = np.atleast_2d(x)
          return x[0,0]
```

```
[19]: get_elem([[1,2], [3,4]])        # Now works on nested lists
```

```
[19]: 1
```

This function suddenly becomes much more flexible, maybe too flexible since it works on all sorts of arguments:

```
[20]: get_elem([1, 2])        # simple list
      get_elem(1.0)           # scalar
```

```
[20]: 1.0
```

NumPy also implements `np.atleast_1d()` and `np.atleast_3d()` which serve the same purpose, but return 1-dimensional or 3-dimensional arrays instead.

12.2.2 Raising exceptions

There are situations when we explicitly want to ensure that some condition is met, instead of letting the code fail somewhere down the line. This is particularly important when we write library functions that might be called from many different contexts or by many different users. Raising an exception with a clear error message is beneficial in such situations.

To illustrate the benefit of clear error messages, consider the following (highly artificial) example:

```
[21]: def get_row(mat, i):
          # restrict to valid row indices
          irow = min(mat.shape[0] - 1, max(0, i))

          # return row
          row = mat[irow]
          return row
```

We define the function `get_row` that returns the `i`-th row of a matrix. The function ensures that the row index is within the admissible range for the given array.

Let's call this function as follows:

```
[22]: import numpy as np
      mat = np.arange(6).reshape((3, 2))
      get_row(mat, 1.0)
```

```
IndexError: only integers, slices (`:`), ellipsis (`...`), numpy.newaxis (`None`) and
↪integer or boolean arrays are valid indices
```

This raises an `IndexError`, notifying the user that the statement `row = mat[irow]` was problematic. However, the caller does not know what `irow` is since this is not the name of the original argument. In the worst case, the user would have to inspect the implementation of `get_row()` to figure out what is wrong.

How can we rectify this situation? We cannot prevent someone from calling this function with an inadmissible value, but we can raise an exception once such a value is encountered.

We raise exceptions using the `raise` statement which is followed by an exception:

[23]:
```python
def get_row(mat, i):
    # Check whether i is an integer
    if not isinstance(i, int):
        msg = f'Integer argument required, received {i}'
        raise ValueError(msg)
    # restrict to valid row indices
    irow = min(mat.shape[0] - 1, max(0, i))

    # return row
    row = mat[irow]
    return row
```

To check whether `i` is of integer type, we use the `isinstance()` function.

The convention is to raise a `ValueError` when a function argument does not satisfy some requirement. We can optionally pass an error message, as in the example above. There is no need or possibility to add an explicit `return` statement: as soon as an exception is raised, any remaining code is skipped. We will examine the details below.

[24]:
```python
get_row(mat, 1)     # Call with integer argument; works as intended.
```

[24]:
```
array([2, 3])
```

[25]:
```python
get_row(mat, 1.0)   # Call with float argument; raises exception
```

```
ValueError: Integer argument required, received 1.0
```

As you see, an exception is raised and a clear error message is returned to the caller.

12.2.3 Catching exceptions

If we are unable or unwilling to take measures to avoid an error, we have to deal with the resulting exception, should one occur. If we fail to do so, the entire program will be terminated.

We handle exceptions using the `try` statement (we sometimes say we "catch" exceptions, which is the keyword used in some other programming languages):

- The code that potentially raises an exception is placed in the `try` clause.
- If an error occurs, control is immediately passed on to
  the `except` clause and any remaining statements in the `try` clause are skipped.
- The `except` clause takes care of handling the exception, should one occur. If no exception is raised, the `except` clause is never executed.

Examples:

Say we need to process an integer value but are unsure about the data type of the input; calling `int()` might therefore work, or it might not:

[26]:
```
x = 1.2345
int(x)          # Works, float is truncated to integer
```

[26]:
```
1
```

[27]:
```
x = 'abc'
int(x)          # Does not work
```

```
ValueError: invalid literal for int() with base 10: 'abc'
```

Calling `int()` with a string such as `'abc'` which cannot be interpreted as an integer will raise a `ValueError`. We could handle such a situation as follows:

[28]:
```
x = 'abc'

try:
    i = int(x)
    print('Conversion to integer works!')
except ValueError:
    print(f'{x} cannot be converted to an integer')
```

```
abc cannot be converted to an integer
```

We see that the execution of the `try` clause terminates as soon as the exception is raised, so the `print()` function is never called. Instead, execution is passed on to the `except` clause which matches the exception type.

We can have multiple `except` clauses covering all sorts of exceptions:

[29]:
```
def func(x):
    try:
        i = int(x)
        print('Conversion to integer works!')
        # Return some value
        return 10/i
    except ValueError:
        print(f'{x} cannot be converted to an integer')
    except ZeroDivisionError:
        print('Division by zero')
    except:
        print('Other exception type occured')
```

[30]:
```
func('abc')         # ValueError: cannot convert integer
```

```
abc cannot be converted to an integer
```

[31]:
```
func(0)             # ZeroDivisionError
```

```
Conversion to integer works!
Division by zero
```

An `except` clause without an exception type catches any exceptions which do not match any preceding `except` clause. For example, this code raises a `TypeError` which is not specifically handled:

[32]:
```
func([1, 2, 3])     # TypeError, caught by default clause
```

```
Other exception type occured
```

If there is no default `except` clause and an unhandled exception occurs, it will be propagated back to the caller as if no error handling was present at all:

[33]:
```
# Define func to only handle ValueError
def func(x):
    try:
        i = int(x)
        print('Conversion to integer works!')
        # Return some value
        return 10/i
    except ValueError:
        print(f'{x} cannot be converted to an integer')
```

[34]:
```
func(0)          # Raises ZeroDivisionError, which is passed to caller
```

```
Conversion to integer works!
```

```
ZeroDivisionError: division by zero
```

This even works across multiple levels of the call stack:

[35]:
```
# inner function converts to integer
def inner(x):
    i = int(x)
    return i

# outer function divides by integer value
def outer(x):
    i = inner(x)
    return 10 /i
```

[36]:
```
outer('abc')            # ValueError raised in inner()
```

```
ValueError: invalid literal for int() with base 10: 'abc'
```

Here we call `outer()`, which in turn calls `inner()`, passing on its argument. Conversion to an integer fails in `inner()`, but since `outer()` does not handle this exception, it is automatically passed on the the original call site.

12.3 Optional exercises

Exercise 1: Sign function

Revisit the sign function you implemented in Unit 4, Exercise 1. To refresh your memory, the suggested solution looks as follows:

[37]:
```
import numpy as np

def sign(x):
    if x < 0.0:
        return -1.0
    elif x == 0.0:
        return 0.0
    elif x > 0.0:
        return 1.0
    else:
        # Argument is not a proper numerical value, return NaN
        # (NaN = Not a Number)
        return np.nan
```

This implementation is not very robust, as it returns all sorts of exceptions when passed non-numeric arguments:

[38]:
```
sign('abc')            # pass in string
```

```
TypeError: '<' not supported between instances of 'str' and 'float'
```

[39]:
```
sign(np.array([1, 2, 3]))          # Pass in NumPy array
```

```
ValueError: The truth value of an array with more than one element is ambiguous. Use a.any()
  ↪or a.all()
```

Modify the `sign()` function such that it only accepts built-in numerical Python types (integers, floats) and raises a `ValueError` in all other cases

Exercise 2: Factorials

Consider the `factorial()` function you wrote in Unit 4, Exercise 4:

[40]:
```
def factorial(n):
    if n == 0:
        return 1
    else:
        # Use recursion to compute factorial
        return n * factorial(n-1)
```

This implementation is also not very robust to nonsensical arguments, for example:

[41]:
```
factorial(1.123)
```

```
RecursionError: maximum recursion depth exceeded in comparison
```

Modify this function such that it only accepts *numerical* arguments that are either integers, or can be interpreted as integers without loss of data, such as a float `1.0` or a scalar array `np.array(1.0)`.

The function should raise a `ValueError` for all other inputs.

Exercise 3: Bisection

Recall the `bisect()` function from Unit 4, Exercise 5:

[42]:
```
def bisect(f, a, b, tol=1.0e-6, xtol=1.0e-6, maxiter=100):

    for iteration in range(maxiter):
        # Compute candidate value as midpoint between a and b
        mid = (a + b) / 2.0
        if abs(b-a) < xtol:
            # Remaining interval is too small
```

```
                break

            fmid = f(mid)

            if abs(fmid) < tol:
                # function value is close enough to zero
                break

            print(f'Iteration {iteration}: f(mid) = {fmid:.4e}')
            if fmid*f(b) > 0.0:
                # f(mid) and f(b) have the same sign, update upper bound b
                print(f'  Updating upper bound to {mid:.8f}')
                b = mid
            else:
                # f(mid) and f(a) have the same sign, or at least one of
                # them is zero.
                print(f'  Updating lower bound to {mid:.8f}')
                a = mid

        return mid
```

This function accepts quite a few arguments, but we never implemented any input validation. Add the following input checks at the top of the function and raise a `ValueError` if any of them is violated:

1. Check that `f(a)` and `f(b)` are of opposite sign, a precondition for the bisection algorithm to work.
2. Check that `tol` and `xtol` are positive and can be interpreted as floating-point numbers.
3. Check that `maxiter` is positive and can be interpreted as an integer.

12.4 Solutions

Solution for exercise 1

We can use the built-in `float()` function to determine whether something can be represented as a floating-point number.

We use only the default `except` clause without any type specification as the code in the `try` clause raises several types of exceptions, depending on the input argument.

[43]:
```python
import numpy as np

def sign(x):
    try:
        # Convert to float, which is more generic than int
        x = float(x)
    except:
        # The above statement raises at least two types
        # of exceptions: ValueError and TypeError
        raise ValueError('Numerical argument required!')

    if x < 0.0:
        return -1.0
    elif x == 0.0:
        return 0.0
    elif x > 0.0:
        return 1.0
    else:
        # Argument is not a proper numerical value, return NaN
        # (NaN = Not a Number)
        return np.nan
```

```
[44]: sign(123)          # integer argument
```

```
[44]: 1.0
```

```
[45]: sign('abc')          # string argument
```

```
ValueError: could not convert string to float: 'abc'


During handling of the above exception, another exception occurred:

ValueError: Numerical argument required!
```

```
[46]: sign(np.array([1, 2, 3]))        # NumPy array argument
```

```
TypeError: only size-1 arrays can be converted to Python scalars


During handling of the above exception, another exception occurred:

ValueError: Numerical argument required!
```

Solution for exercise 2

One possible solution looks as follows:

```
[47]: def factorial(n):
          try:
              i = int(n)
              assert i == n
          except:
              raise ValueError(f'Not an integer argument: {n}')

          if i == 0:
              return 1
          else:
              # Use recursion to compute factorial
              return i * factorial(i-1)
```

We perform input validation in two steps:

1. We use the `int()` function to convert the input to an integer. This will eliminate some inadmissible arguments such as `'abc'` or `[1, 2, 3]` but will accept others such as `'1'` or `1.1`. We want to eliminate these as well, since `'1'` is not numeric and `1.1` cannot be represented as an integer without loss of data.
2. We achieve this with the `assert` statement where we check whether `i == n`: this will only be true if `n` is numerical and does not have a fractional part.

   The `assert` statement will raise an `AssertionError` whenever a condition is not `True`, which will also be handled by the `except` clause.

```
[48]: factorial(1)          # integer argument
```

```
[48]: 1
```

```
[49]: factorial(1.0)            # not an integer argument, but can be
                                # represented as an integer.
```

```
[49]: 1
```

```
[50]: factorial(1.1)            # Floating-point argument that
                                # cannot be represented as an integer
```

```
AssertionError


During handling of the above exception, another exception occurred:

ValueError: Not an integer argument: 1.1
```

```
[51]: factorial('1')            # String argument
```

```
AssertionError


During handling of the above exception, another exception occurred:

ValueError: Not an integer argument: 1
```

```
[52]: factorial(np.array(10.0))       # Scalar array argument
```

```
[52]: 3628800
```

Solution for exercise 3

We modify the function as follows:

```
[53]: def bisect(f, a, b, tol=1.0e-6, xtol=1.0e-6, maxiter=100):

          fa = f(a)
          fb = f(b)

          if fa*fb > 0.0:
              raise ValueError(f'Not a bracketing interval [{a}, {b}]')
          try:
              tol = float(tol)
              assert tol > 0.0
          except:
              raise ValueError('Argument tol must be a positive number!')

          try:
              xtol = float(xtol)
              assert xtol > 0.0
          except:
              raise ValueError('Argument xtol must be a positive number!')

          try:
              maxiter = int(maxiter)
              assert maxiter > 0
          except:
```

```
        raise ValueError('Argument maxiter must be a positive integer!')


    for iteration in range(maxiter):
        # Compute candidate value as midpoint between a and b
        mid = (a + b) / 2.0
        if abs(b-a) < xtol:
            # Remaining interval is too small
            break

        fmid = f(mid)

        if abs(fmid) < tol:
            # function value is close enough to zero
            break

        print(f'Iteration {iteration}: f(mid) = {fmid:.4e}')
        if fmid*f(b) > 0.0:
            # f(mid) and f(b) have the same sign, update upper bound b
            print(f'  Updating upper bound to {mid:.8f}')
            b = mid
        else:
            # f(mid) and f(a) have the same sign, or at least one of
            # them is zero.
            print(f'  Updating lower bound to {mid:.8f}')
            a = mid

    return mid
```

As in the main loop of the function, we check whether two values are non-zero and have the same sign using the condition `fa * fb > 0`, in which case we have no bracketing interval and need to raise a `ValueError`.

The remaining checks are performed using the same code as in earlier exercises.

[54]:
```
# Function call with valid argument
x0 = bisect(lambda x: x**2.0 - 4.0, -3.0, 0.0, tol=1.0e-3)
```

```
Iteration 0: f(mid) = -1.7500e+00
  Updating upper bound to -1.50000000
Iteration 1: f(mid) = 1.0625e+00
  Updating lower bound to -2.25000000
Iteration 2: f(mid) = -4.8438e-01
  Updating upper bound to -1.87500000
Iteration 3: f(mid) = 2.5391e-01
  Updating lower bound to -2.06250000
Iteration 4: f(mid) = -1.2402e-01
  Updating upper bound to -1.96875000
Iteration 5: f(mid) = 6.2744e-02
  Updating lower bound to -2.01562500
Iteration 6: f(mid) = -3.1189e-02
  Updating upper bound to -1.99218750
Iteration 7: f(mid) = 1.5640e-02
  Updating lower bound to -2.00390625
Iteration 8: f(mid) = -7.8087e-03
  Updating upper bound to -1.99804688
Iteration 9: f(mid) = 3.9072e-03
  Updating lower bound to -2.00097656
Iteration 10: f(mid) = -1.9529e-03
  Updating upper bound to -1.99951172
```

[55]:
```
# Function call with f(a) and f(b) both positive
x0 = bisect(lambda x: x**2.0 - 4.0, 10.0, 20.0)
```

```
ValueError: Not a bracketing interval [10.0, 20.0]
```

[56]:
```
# Function call with invalid tolerance criterion
x0 = bisect(lambda x: x**2.0 - 4.0, -3.0, 0.0, tol=0.0)
```

```
AssertionError


During handling of the above exception, another exception occurred:

ValueError: Argument tol must be a positive number!
```