Scientific Computing for Finance using Python

Author:
(c) Steve Phelps 2019
sphelps@sphelps.net

December 20, 2019

Contents

1 Overview of Python

(c) 2019 Steve Phelps

1.1 Python is interpreted

- Python is an *interpreted* language, in contrast to Java and C which are compiled languages.
- This means we can type statements into the interpreter and they are executed immediately.

```
5 + 5
```

```
10
```

- Groups of statements are all executed one after the other:

```
x = 5
y = 'Hello There'
z = 10.5
```

- We can visualize the above code using PythonTutor.

```
x + 5
```

```
10
```

1.2 Assignments versus equations

- In Python when we write `x = 5` this means something different from an equation $x = 5$.
- Unlike variables in mathematical models, variables in Python can refer to different things as more statements are interpreted.

```
x = 1
print('The value of x is', x)

x = 2.5
print('Now the value of x is', x)

x = 'hello there'
print('Now it is ', x)
```

```
The value of x is 1
Now the value of x is 2.5
Now it is  hello there
```

1.3 Calling Functions

We can call functions in a conventional way using round brackets

```python
round(3.14)
```

```
3
```

1.4 Types

- Values in Python have an associated *type*.
- If we combine types incorrectly we get an error.

```python
print(y)
```

```
Hello There
```

```python
y + 5
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent
↪ call last)

<ipython-input-8-b85a2dbb3f6a> in <module>
----> 1 y + 5

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

1.5 The type function

- We can query the type of a value using the `type` function.

```python
type(1)
```

```
int
```

```python
type('hello')
```

```
str
```

```python
type(2.5)
```

```
float
```

```python
type(True)
```

```
bool
```

1.6 Null values

- Sometimes we represent “no data” or “not applicable”.
- In Python we use the special value `None`.
- This corresponds to `Null` in Java or SQL.

```
result = None
```

- When we fetch the value `None` in the interactive interpreter, no result is printed out.

```
result
```

1.7 Testing for Null values

- We can check whether there is a result or not using the `is` operator:

```
result is None
```

```
True
```

```
x = 5
x is None
```

```
False
```

1.8 Converting values between types

- We can convert values between different types.

1.8.1 Converting to floating-point

- To convert an integer to a floating-point number use the `float()` function.

```
x = 1
x
```

```
1
```

```
type(x)
```

```
int
```

```
y = float(x)
y
```

```
1.0
```

1.8.2 Converting to integers

- To convert a floating-point to an integer use the `int()` function.

```python
type(y)
```

```
float
```

```python
int(y)
```

```
1
```

1.9 Variables are not typed

- *Variables* themselves, on the other hand, do not have a fixed type.
- It is only the values that they refer to that have a type.
- This means that the type referred to by a variable can change as more statements are interpreted.

```python
y = 'hello'
print('The type of the value referred to by y is ', type(y))
y = 5.0
print('And now the type of the value is ', type(y))
```

```
The type of the value referred to by y is  <class 'str'>
And now the type of the value is  <class 'float'>
```

1.10 Polymorphism

- The meaning of an operator depends on the types we are applying it to.

```python
1 + 1
```

```
2
```

```python
'a' + 'b'
```

```
'ab'
```

```python
'1' + '1'
```

```
'11'
```

1.11 Conditional Statements and Indentation

- The syntax for control structures in Python uses *colons* and *indentation*.
- Beware that white-space affects the semantics of Python code.
- Statements that are indented using the Tab key are grouped together.

1.11.1 `if` statements

```python
x = 5
if x > 0:
    print('x is strictly positive.')
    print(x)

print('finished.')
```

```
x is strictly positive.
5
finished.
```

- Visualize the above on PythonTutor.

1.11.2 Changing indentation

```python
x = 0
if x > 0:
    print('x is strictly positive.')
print(x)

print('finished.')
```

```
0
finished.
```

- Visualize the above on PythonTutor.

1.11.3 `if` and `else`

```python
x = 0
print('Starting.')
if x > 0:
    print('x is strictly positive.')
else:
    if x < 0:
        print('x is strictly negative.')
    else:
        print('x is zero.')
print('finished.')
```

```
Starting.
x is zero.
finished.
```

- Visualize the above on PythonTutor.

1.11.4 `elif`

```python
print('Starting.')
if x > 0:
    print('x is strictly positive')
elif x < 0:
    print('x is strictly negative')
else:
```

```python
    print('x is zero')
print('finished.')
```

```
Starting.
x is zero
finished.
```

1.12 Lists

We can use *lists* to hold an ordered sequence of values.

```python
l = ['first', 'second', 'third']
l
```

```
['first', 'second', 'third']
```

Lists can contain different types of variable, even in the same list.

```python
another_list = ['first', 'second', 'third', 1, 2, 3]
another_list
```

```
['first', 'second', 'third', 1, 2, 3]
```

1.13 Mutable Datastructures

Lists are *mutable*; their contents can change as more statements are interpreted.

```python
l.append('fourth')
l
```

```
['first', 'second', 'third', 'fourth']
```

1.14 References

- Whenever we bind a variable to a value in Python we create a *reference*.
- A reference is distinct from the value that it refers to.
- Variables are names for references.

```python
X = [1, 2, 3]
Y = X
```

1.14.1 Side effects

- The above code creates two different references (named `X` and `Y`) to the *same* value `[1, 2, 3]`
- Because lists are mutable, changing them can have side-effects on other variables.
- If we append something to `X` what will happen to `Y`?

```
X.append(4)
X
```

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

```
Y
```

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

- Visualize the above on PythonTutor.

1.15 State and identity

- The state referred to by a variable is *different* from its identity.
- To compare *state* use the == operator.
- To compare *identity* use the `is` operator.
- When we compare identity we check equality of references.
- When we compare state we check equality of values.

1.15.1 Example

- We will create two *different* lists, with two associated variables.

```
X = [1, 2]
Y = [1]
Y.append(2)
```

- Visualize the above code on PythonTutor.

1.15.2 Comparing state

```
X
```

```
[1, 2]
```

```
Y
```

```
[1, 2]
```

```
X == Y
```

```
True
```

1.15.3 Comparing identity

```python
X is Y
```

```
False
```

1.15.4 Copying data prevents side effects

- In this example, because we have two different lists we avoid side effects

```python
Y.append(3)
X
```

```
[1, 2]
```

```python
X == Y
```

```
False
```

```python
X is Y
```

```
False
```

1.16 Iteration

- We can iterate over each element of a list in turn using a `for` loop:

```python
my_list = ['first', 'second', 'third', 'fourth']
for i in my_list:
    print(i)
```

```
first
second
third
fourth
```

- Visualize the above on PythonTutor.

1.16.1 Including more than one statement inside the loop

```python
my_list = ['first', 'second', 'third', 'fourth']
for i in my_list:
    print("The next item is:")
    print(i)
    print()
```

```
The next item is:
first

The next item is:
second

The next item is:
third
```

```
The next item is:
fourth
```

- Visualize the above code on PythonTutor.

1.16.2 Looping a specified number of times

- To perform a statement a certain number of times, we can iterate over a list of the required size.

```python
for i in [0, 1, 2, 3]:
    print("Hello!")
```

```
Hello!
Hello!
Hello!
Hello!
```

1.16.3 The `range` function

- To save from having to manually write the numbers out, we can use the function `range()` to count for us.
- We count starting at 0 (as in Java and C++).

```python
list(range(4))
```

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

1.16.4 `for` loops with the `range` function

```python
for i in range(4):
    print("Hello!")
```

```
Hello!
Hello!
Hello!
Hello!
```

1.17 List Indexing

- Lists can be indexed using square brackets to retrieve the element stored in a particular position.

```python
my_list
```

```
['first', 'second', 'third', 'fourth']
```

```python
my_list[0]
```

```
'first'
```

```
my_list[1]
```

```
'second'
```

1.18 List Slicing

- We can also a specify a *range* of positions.
- This is called *slicing*.
- The example below indexes from position 0 (inclusive) to 2 (exclusive).

```
my_list[0:2]
```

```
['first', 'second']
```

1.18.1 Indexing from the start or end

- If we leave out the starting index it implies the beginning of the list:

```
my_list[:2]
```

```
['first', 'second']
```

- If we leave out the final index it implies the end of the list:

```
my_list[2:]
```

```
['third', 'fourth']
```

1.18.1.1 Copying a list

- We can conveniently copy a list by indexing from start to end:

```
new_list = my_list[:]
```

```
new_list
```

```
['first', 'second', 'third', 'fourth']
```

```
new_list is my_list
```

```
False
```

```python
new_list == my_list
```

```
True
```

1.19 Negative Indexing

- Negative indices count from the end of the list:

```python
my_list[-1]
```

```
'fourth'
```

```python
my_list[:-1]
```

```
['first', 'second', 'third']
```

1.20 Collections

- Lists are an example of a *collection*.
- A collection is a type of value that can contain other values.
- There are other collection types in Python:
  - `tuple`
  - `set`
  - `dict`

1.20.1 Tuples

- Tuples are another way to combine different values.
- The combined values can be of different types.
- Like lists, they have a well-defined ordering and can be indexed.
- To create a tuple in Python, use round brackets instead of square brackets

```python
tuple1 = (50, 'hello')
tuple1
```

```
(50, 'hello')
```

```python
tuple1[0]
```

```
50
```

```python
type(tuple1)
```

```
tuple
```

1.20.1.1 Tuples are immutable

- Unlike lists, tuples are *immutable*. Once we have created a tuple we cannot add values to it.

```
tuple1.append(2)
```

```
---------------------------------------------------------------------------
```

```
AttributeError                            Traceback (most recent
↪ call last)
```

```
<ipython-input-64-46e3866e32ee> in <module>
----> 1 tuple1.append(2)
```

```
AttributeError: 'tuple' object has no attribute 'append'
```

1.20.2 Sets

- Lists can contain duplicate values.
- A set, in contrast, contains no duplicates.
- Sets can be created from lists using the `set()` function.

```
X = set([1, 2, 3, 3, 4])
X
```

```
{1, 2, 3, 4}
```

```
type(X)
```

```
set
```

- Alternatively we can write a set literal using the { and } brackets.

```
X = {1, 2, 3, 4}
type(X)
```

```
set
```

1.20.2.1 Sets are mutable

- Sets are mutable like lists:

```
X.add(5)
X
```

```
{1, 2, 3, 4, 5}
```

- Duplicates are automatically removed

```
X.add(5)
X
```

```
{1, 2, 3, 4, 5}
```

1.20.2.2 Sets are unordered

- Sets do not have an ordering.
- Therefore we cannot index or slice them:

```
X[0]
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent
↪ call last)

<ipython-input-70-19c40ecbd036> in <module>
----> 1 X[0]

TypeError: 'set' object is not subscriptable
```

1.20.2.3 Operations on sets

- Union: $X \cup Y$

```
X = {1, 2, 3}
Y = {4, 5, 6}
X | Y
```

```
{1, 2, 3, 4, 5, 6}
```

- Intersection: $X \cap Y$:

```
X = {1, 2, 3, 4}
Y = {3, 4, 5}
X & Y
```

```
{3, 4}
```

- Difference $X - Y$:

```
X - Y
```

```
{1, 2}
```

1.20.3 Dictionaries

- A dictionary contains a mapping between *keys*, and corresponding *values*.
  - Mathematically it is a one-to-one function with a finite domain and range.
- Given a key, we can very quickly look up the corresponding value.
- The values can be any type (and need not all be of the same type).
- Keys can be any immutable (hashable) type.
- They are abbreviated by the keyword `dict`.
- In other programming languages they are sometimes called *associative arrays*.

1.20.3.1 Creating a dictionary

- A dictionary contains a set of key-value pairs.
- To create a dictionary:

```
students = { 107564: 'Xu', 108745: 'Ian', 102567: 'Steve' }
```

- The above initialises the dictionary students so that it contains three key-value pairs.
- The keys are the student id numbers (integers).
- The values are the names of the students (strings).
- Although we use the same brackets as for sets, this is a different type of collection:

```
type(students)
```

```
dict
```

1.20.3.2 Accessing the values in a dictionary

- We can access the value corresponding to a given key using the same syntax to access particular elements of a list:

```
students[108745]
```

```
'Ian'
```

- Accessing a non-existent key will generate a `KeyError`:

```
students[123]
```

```
---------------------------------------------------------------------------

KeyError                                  Traceback (most recent
↪ call last)

<ipython-input-77-26e887eb0296> in <module>
----> 1 students[123]

KeyError: 123
```

1.20.3.3 Updating dictionary entries

- Dictionaries are mutable, so we can update the mapping:

```
students[108745] = 'Fred'
print(students[108745])
```

```
Fred
```

- We can also grow the dictionary by adding new keys:

```
students[104587] = 'John'
print(students[104587])
```

```
John
```

1.20.3.4 Dictionary keys can be any immutable type

- We can use any immutable type for the keys of a dictionary
- For example, we can map names onto integers:

```
age = { 'John':21, 'Steve':47, 'Xu': 22 }
```

```
age['Steve']
```

```
47
```

1.20.3.5 Creating an empty dictionary

- We often want to initialise a dictionary with no keys or values.
- To do this call the function `dict()`:

```
result = dict()
```

- We can then progressively add entries to the dictionary, e.g. using iteration:

```
for i in range(5):
    result[i] = i**2
print(result)
```

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

1.20.3.6 Iterating over a dictionary

- We can use a for loop with dictionaries, just as we can with other collections such as sets.
- When we iterate over a dictionary, we iterate over the *keys*.
- We can then perform some computation on each key inside the loop.
- Typically we will also access the corresponding value.

```
for id in students:
    print(students[id])
```

```
Xu
Fred
Steve
John
```

1.20.4 The size of a collection

- We can count the number of values in a collection using the `len` (length) function.
- This can be used with any type of collection (list, set, tuple etc.).

```python
len(students)
```

```
4
```

```python
len(['one', 'two'])
```

```
2
```

```python
len({'one', 'two', 'three'})
```

```
3
```

1.20.4.1 Empty collections

- Empty collections have a size of zero:

```python
empty_list = []
len(empty_list) == 0
```

```
True
```

1.20.5 Arrays

- Python also has arrays which contain a *single* type of value.
- i.e. we *cannot* have different types of value within the same array.
- Arrays are mutable like lists; we can modify the existing elements of an array.
- However, we typically do not change the size of the array; i.e. it has a fixed length.

1.21 The `numpy` module

- Arrays are provided by a separate *module* called numpy. Modules correspond to packages in e.g. Java.
- We can import the module and then give it a shorter *alias*.

```python
import numpy as np
```

- We can now use the functions defined in this package by prefixing them with `np`.
- The function `array()` creates an array given a list.

1.21.1 Creating an array

- We can create an array from a list by using the `array()` function defined in the `numpy` module:

```python
x = np.array([0, 1, 2, 3, 4])
x
```

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

```
type(x)
```

```
numpy.ndarray
```

1.21.2 Functions over arrays

- When we use arithmetic operators on arrays, we create a new array with the result of applying the operator to each element.

```
y = x * 2
y
```

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

- The same goes for functions:

```
x = np.array([-1, 2, 3, -4])
y = abs(x)
y
```

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

1.21.3 Populating Arrays

- To populate an array with a range of values we use the `np.arange()` function:

```
x = np.arange(0, 10)
x
```

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

- We can also use floating point increments.

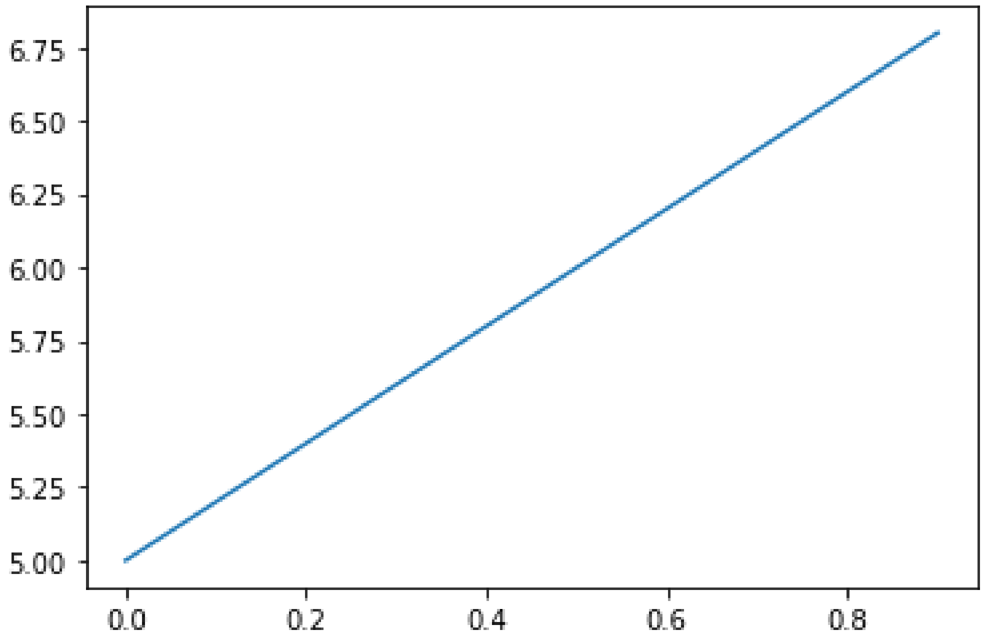
```
x = np.arange(0, 1, 0.1)
x
```

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

1.21.4 Basic Plotting

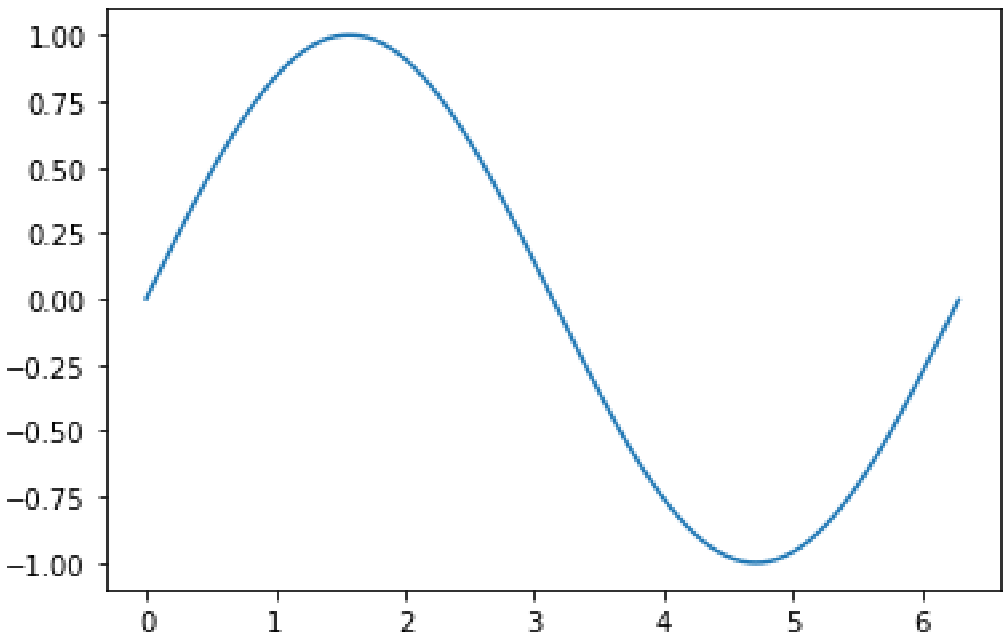
- We will use a module called `matplotlib` to plot some simple graphs.
- This module provides functions which are very similar to MATLAB plotting commands.

```
import matplotlib.pyplot as plt

y = x*2 + 5
plt.plot(x, y)
plt.show()
```



1.21.5 Plotting a sine curve

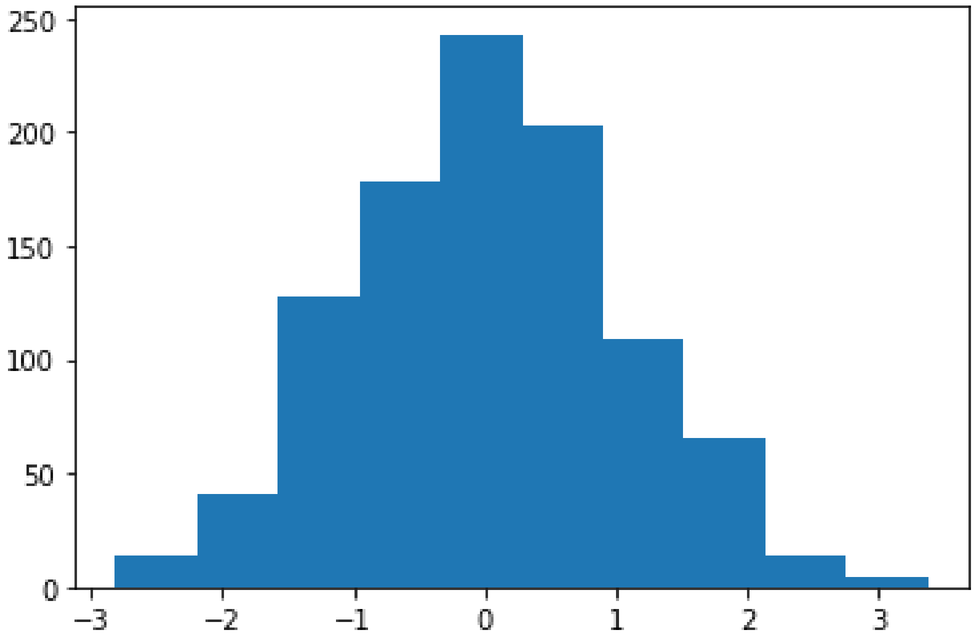
```
from numpy import pi, sin

x = np.arange(0, 2*pi, 0.01)
y = sin(x)
plt.plot(x, y)
plt.show()
```



1.21.6 Plotting a histogram

- We can use the `hist()` function in `matplotlib` to plot a histogram

```
# Generate some random data
data = np.random.randn(1000)

ax = plt.hist(data)
plt.show()
```



1.21.7 Computing histograms as matrices

- The function `histogram()` in the `numpy` module will count frequencies into bins and return the result as a 2-dimensional array.

```
np.histogram(data)
```

```
(array([ 14,  41, 128, 178, 243, 203, 109,  66,  14,   4]),
 array([-2.81515826, -2.19564948, -1.57614071, -0.95663193, -
0.33712315,
         0.28238562,  0.9018944 ,  1.52140318,  2.14091195,
2.76042073,
         3.3799295 ]))
```

1.22 Defining new functions

```
def squared(x):
    return x ** 2

```

```
squared(5)
```

```
25
```

1.23 Local Variables

- Variables created inside functions are *local* to that function.
- They are not accessable to code outside of that function.

```
def squared(x):
    temp = x ** 2
    return temp

squared(5)
```

```
25
```

```
temp
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent
↪ call last)

<ipython-input-102-da77557ed0c8> in <module>
----> 1 temp

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

1.24 Functional Programming

- Functions are first-class citizens in Python.
- They can be passed around just like any other value.

```
squared
```

```
<function __main__.squared(x)>
```

```
y = squared
y
```

```
<function __main__.squared(x)>
```

```
y(5)
```

```
25
```

1.25 Mapping the elements of a collection

- We can apply a function to each element of a collection using the built-in function `map()`.
- This will work with any collection: list, set, tuple or string.
- This will take as an argument *another function*, and the list we want to apply it to.
- It will return the results of applying the function, as a list.

```python
list(map(squared, [1, 2, 3, 4]))
```

```
[1, 4, 9, 16]
```

1.26 List Comprehensions

- Because this is such a common operation, Python has a special syntax to do the same thing, called a *list comprehension*.

```python
[squared(i) for i in [1, 2, 3, 4]]
```

```
[1, 4, 9, 16]
```

- If we want a set instead of a list we can use a set comprehension

```python
{squared(i) for i in [1, 2, 3, 4]}
```

```
{1, 4, 9, 16}
```

1.27 Cartesian product using list comprehensions

image courtesy of Quartl

The Cartesian product of two collections $X = A \times B$ can be expressed by using multiple `for` statements in a comprehension.

1.27.1 example

```python
A = {'x', 'y', 'z'}
B = {1, 2, 3}
{(a,b) for a in A for b in B}
```

```
{('x', 1),
 ('x', 2),
 ('x', 3),
 ('y', 1),
 ('y', 2),
 ('y', 3),
 ('z', 1),
```

```
('z', 2),
('z', 3)}
```

1.28 Cartesian products with other collections

- The syntax for Cartesian products can be used with any collection type.

```python
first_names = ('Steve', 'John', 'Peter')
surnames = ('Smith', 'Doe', 'Rabbit')

[(first_name, surname) for first_name in first_names for surname in
    surnames]
```

```
[('Steve', 'Smith'),
 ('Steve', 'Doe'),
 ('Steve', 'Rabbit'),
 ('John', 'Smith'),
 ('John', 'Doe'),
 ('John', 'Rabbit'),
 ('Peter', 'Smith'),
 ('Peter', 'Doe'),
 ('Peter', 'Rabbit')]
```

1.29 Joining collections using a zip

- The Cartesian product pairs every combination of elements.
- If we want a 1-1 pairing we use an operation called a zip.
- A zip pairs values at the same position in each sequence.
- Therefore:
    - it can only be used with sequences (not sets); and
    - both collections must be of the same length.

```python
list(zip(first_names, surnames))
```

```
[('Steve', 'Smith'), ('John', 'Doe'), ('Peter', 'Rabbit')]
```

1.30 Anonymous Function Literals

- We can also write *anonymous* functions.
- These are function literals, and do not necessarily have a name.
- They are called *lambda expressions* (after the $\lambda-$calculus).

```python
list(map(lambda x: x ** 2, [1, 2, 3, 4]))
```

```
[1, 4, 9, 16]
```

1.31 Filtering data

- We can filter a list by applying a *predicate* to each element of the list.
- A predicate is a function which takes a single argument, and returns a boolean value.
- `filter(p, X)` is equivalent to $\{x : p(x) \; \forall x \in X\}$ in set-builder notation.

```
list(filter(lambda x: x > 0, [-5, 2, 3, -10, 0, 1]))
```

```
[2, 3, 1]
```

We can use both `filter()` and `map()` on other collections such as strings or sets.

```
list(filter(lambda x: x > 0, {-5, 2, 3, -10, 0, 1}))
```

```
[1, 2, 3]
```

1.32 Filtering using a list comprehension

- Again, because this is such a common operation, we can use simpler syntax to say the same thing.
- We can express a filter using a list-comprehension by using the keyword `if`:

```
data = [-5, 2, 3, -10, 0, 1]
[x for x in data if x > 0]
```

```
[2, 3, 1]
```

- We can also filter and then map in the same expression:

```
from numpy import sqrt
[sqrt(x) for x in data if x > 0]
```

```
[1.4142135623730951, 1.7320508075688772, 1.0]
```

1.33 The reduce function

- The `reduce()` function recursively applies another function to pairs of values over the entire list, resulting in a *single* return value.

```
from functools import reduce
reduce(lambda x, y: x + y, [0, 1, 2, 3, 4, 5])
```

```
15
```

1.34 Big Data

- The `map()` and `reduce()` functions form the basis of the map-reduce programming model.
- Map-reduce is the basis of modern highly-distributed large-scale computing frameworks.
- It is used in BigTable, Hadoop and Apache Spark.
- See these examples in Python for Apache Spark.

2 Numerical Computing in Python

(c) 2019 Steve Phelps

2.1 Overview

- Floating-point representation
- Arrays and Matrices with `numpy`
- Basic plotting with `matplotlib`
- Pseudo-random variates with `numpy.random`

2.2 Representing continuous values

- Digital computers are inherently *discrete*.
- Real numbers $x \in R$ cannot always be represented exactly in a digital computer.
- They are stored in a format called *floating-point*.
- IEEE Standard 754 specifies a universal format across different implementations.
  - As always there are deviations from the standard.
- There are two standard sizes of floating-point numbers: 32-bit and 64-bit.
- 64-bit numbers are called *double precision*, are sometimes called *double* values.
- IEEE floating-point calculations are performed in *hardware* on modern computers.
- How can we represent aribitrary real values using only 32 bits?

2.3 Fixed-point verses floating-point

- One way we could discretise continuous values is to represent them as two integers x and y.
- The final value is obtained by e.g. $r = x + y \times 10^{-5}$.
- So the number 500.4421 would be represented as the tuple $x = 500$, $y = 44210$.
- The exponent 5 is fixed for all computations.
- This number represents the *precision* with which we can represent real values.
- It corresponds to the where we place we place the decimal *point*.
- This scheme is called fixed precision.
- It is useful in certain circumstances, but suffers from many problems, in particular it can only represent a very limited *range* of values.
- In practice, we use variable precision, also known as *floating* point.

2.4 Scientific Notation

- Humans also use a form of floating-point representation.
- In Scientific notation, all numbers are written in the form $m \times 10^n$.
- When represented in ASCII, we abbreviate this `<m>e<n>`, for example `6.72e+11` = 6.72×10^{11}.
- The integer m is called the *significand* or *mantissa*.
- The integer n is called the *exponent*.
- The integer 10 is the *base*.

2.5 Scientific Notation in Python

- Python uses Scientific notation when it displays floating-point numbers:

```python
print(672000000000000000.0)
```

```
6.72e+17
```

- Note that internally, the value is not *represented* exactly like this.
- Scientific notation is a convention for writing or rendering numbers, *not* representing them digitally.

2.6 Floating-point representation

- Floating point numbers use a base of 2 instead of 10.
- Additionally, the mantissa are exponent are stored in binary.
- Therefore we represent floating-point numbers as $m \times 2^e$.
- The integers m (mantissa) and e (exponent) are stored in binary.
- The mantissa uses two's complement to represent positive and negative numbers.
  - One bit is reserved as the sign-bit: 1 for negative, 0 for positive values.
- The mantissa is normalised, so we assume that it starts with the digit 1 (which is not stored).

2.7 Bias

- We also need to represent signed *exponents*.
- The exponent does *not* use two's complement.
- Instead a *bias* value is subtracted from the stored exponent (s) to obtain the final value (e).
- Double-precision values use a bias of $b = 1023$, and single-precision uses a bias value of $b = 127$.
- The actual exponent is given by $e = s - b$ where s is the stored exponent.
- The stored exponent values $s = 0$ and $s = 1024$ are reserved for special values- discussed later.
- The stored exponent s is represented in binary *without using a sign bit*.

2.8 Double and single precision formats

The number of bits allocated to represent each integer component of a float is given below:

Format	Sign	Exponent	Mantissa	Total
single	1	8	23	32
double	1	11	52	64

- By default, Python uses 64-bit precision.
- We can specify alternative precision by using the numpy numeric data types.

2.9 Loss of precision

- We cannot represent every value in floating-point.
- Consider single-precision (32-bit).
- Let's try to represent $4,039,944,879$.

2.10 Loss of precision

- As a binary integer we write $4,039,944,879$ as:

```
11110000 11001100 10101010 10101111
```

- This already takes up 32-bits.
- The mantissa only allows us to store 24-bit integers.
- So we have to *round*. We store it as:

```
+1.1110000 11001100 10101011e+31
```

- Which gives us

```
+11110000 11001100 10101011 0000000
```

$= 4,039,944,960$

2.11 Ranges of floating-point values

In single precision arithmetic, we *cannot* represent the following values:

- Negative numbers less than $-(2-2^{-23}) \times 2^{127}$
- Negative numbers greater than -2^{-149}
- Positive numbers less than 2^{-149}
- Positive numbers greater than $(2-2^{-23}) \times 2^{127}$

Attempting to represent these numbers results in *overflow* or *underflow*.

2.12 Effective floating-point range

Format	Binary	Decimal
single	$\pm(2-2^{-23}) \times 2^{127}$	$\approx \pm 10^{38.53}$
double	$\pm(2-2^{-52}) \times 2^{1023}$	$\approx \pm 10^{308.25}$

```python
import sys
sys.float_info.max
```

```
1.7976931348623157e+308
```

```python
sys.float_info.min
```

```
2.2250738585072014e-308
```

```python
sys.float_info
```

```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp
=308, min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig
=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

2.13 Range versus precision

- With a fixed number of bits, we have to choose between:
  - maximising the range of values (minimum to maximum) we can represent,
  - maximising the precision with-which we can represent each indivdual value.
- These are conflicting objectives:
  - we can increase range, but only by losing precision,
  - we can increase precision, but only by decreasing range.
- Floating-point addresses this dilemma by allowing the precision to *vary* ("float") according to the magnitude of the number we are trying to represent.

2.14 Floating-point density

- Floating-point numbers are unevenly-spaced over the line of real-numbers.
- The precision *decreases* as we increase the magnitude.

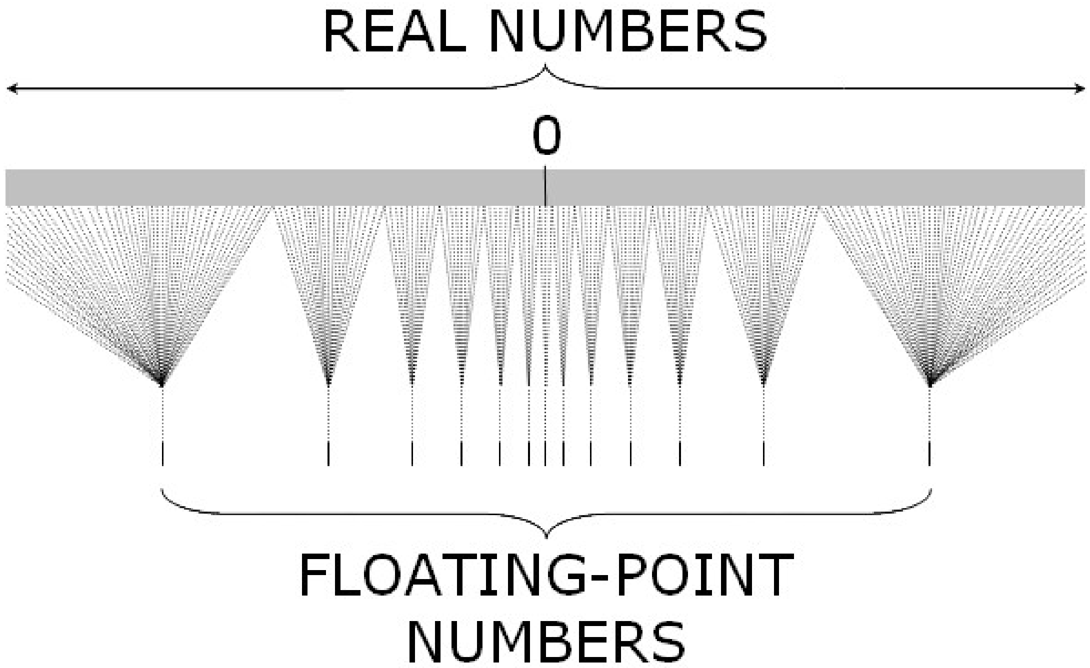


Figure 2.1: density of floats

2.15 Representing Zero

- Zero cannot be represented straightforwardly because we assume that all mantissa values start with the digit 1.
- Zero is stored as a special-case, by setting mantissa *and* exponent both to zero.
- The sign-bit can either be set or unset, so there are distinct positive and negative representations of zero.

2.16 Zero in Python

```
x = +0.0
x
```

```
0.0
```

```
y = -0.0
y
```

```
-
0.0
```

- However, these are considered equal:

```
x == y
```

```
True
```

2.17 Infinity

- Positive overflow results in a special value of infinity (in Python `inf`).
- This is stored with an exponent consiting of all 1s, and a mantissa of all 0s.
- The sign-bit allows us to differentiate between negative and positive overflow: $-\infty$ and $+\infty$.
- This allows us to carry on calculating past an overflow event.

2.18 Infinity in Python

```
x = 1e300 * 1e100
x
```

```
inf
```

```
x = x + 1
x
```

```
inf
```

2.19 Negative infinity in Python

```
x > 0
```

```
True
```

```
y = -x
y
```

```
-
inf
```

```
y < x
```

```
True
```

2.20 Not A Number (NaN)

- Some mathematical operations on real numbers do not map onto real numbers.
- These results are represented using the special value to `NaN` which represents "not a (real) number".
- `NaN` is represented by an exponent of all 1s, and a non-zero mantissa.

2.21 `NaN` in Python

```
from numpy import sqrt, inf, isnan, nan
x = sqrt(-1)
x
```

```
/home/sphelps/anaconda3/lib/python3.6/site-packages/ipykernel_launcher
.py:2: RuntimeWarning: invalid value encountered in sqrt
```

```
nan
```

```
y = inf - inf
y
```

```
nan
```

2.22 Comparing nan values in Python

- Beware of comparing `nan` values

```
x == y
```

```
False
```

- To test whether a value is `nan` use the `isnan` function:

```
isnan(x)
```

```
True
```

2.23 NaN is not the same as None

- `None` represents a *missing* value.
- `NaN` represents an *invalid* floating-point value.
- These are fundamentally different entities:

```
nan is None
```

```
False
```

```
isnan(None)
```

```
    ---------------------------------------------------------------------------
```

```
    TypeError                                 Traceback (most recent
↪ call last)
```

```
<ipython-input-20-4a0142ec4134> in <module>()
----> 1 isnan(None)
```

```
TypeError: ufunc 'isnan' not supported for the input types, and the
↪  inputs could not be safely coerced to any supported types
↪ according to the casting rule ''safe''
```

2.24 Beware finite precision

```
x = 0.1 + 0.2
```

```
x == 0.3
```

```
False
```

```
x
```

```
0.30000000000000004
```

2.25 Relative and absolute error

- Consider a floating point number x_{fp} which represents a real number $x \in \mathbf{R}$.
- In general, we cannot precisely represent the real number; that is $x_{fp} \neq x$.
- The absolute error r is $r = x - x_{fp}$.
- The relative error R is:

$$R = \frac{x - x_{fp}}{x} \tag{2.1}$$

2.26 Numerical Methods

- In e.g. simulation models or quantiative analysis we typically repeatedly update numerical values inside long loops.
- Programs such as these implement *numerical algorithms*.
- It is very easy to introduce bugs into code like this.

2.27 Numerical stability

- The round-off error associated with a result can be compounded in a loop.
- If the error increases as we go round the loop, we say the algorithm is numerically *unstable*.
- Mathematicians design numerically stable algorithms using numerical analysis.

2.28 Catastrophic Cancellation

- Suppose we have two real values x, and $y = x + \epsilon$.
- ϵ is very small and x is very large.
- x has an *exact* floating point representation
- However, because of lack of precision x and y have the same floating point representation.
  - i.e. they are represented as the same sequence of 64-bits
- Consider wheat happens when we compute $y - x$ in floating-point.

2.29 Catestrophic Cancellation and Relative Error

- Catestrophic cancellation results in very large relative error.
- If we calculate $y - x$ in floating-point we will obtain the result 0.
- The correct value is $(x + \epsilon) - x = \epsilon$.
- The relative error is

$$\frac{\epsilon - 0}{\epsilon} = 1 \tag{2.2}$$

- That is, the relative error is 100%.
- This can result in catastrophy.

2.30 Catastrophic Cancellation in Python

```
x = 3.141592653589793
x
```

```
3.141592653589793
```

```
y = 6.022e23
x = (x + y) - y
```

```
x
```

```
0.0
```

2.30.1 Cancellation versus addition

- Addition, on the other hand, is not catestrophic.

```
z = x + y
z
```

```
6.022e+23
```

- The above result is still inaccurate with an absolute error $r \approx \pi$.
- However, let's examine the *relative* error:

$$R = \frac{1.2044 \times 10^{24} - (1.2044 \times 10^{24} + \pi)}{1.2044 \times 10^{24} + \pi} \approx 10^{-24} \tag{2.3}$$

- Here we see that that the relative error from the addition is miniscule compared with the cancellation.

2.30.2 Floating-point arithemetic is nearly always inaccurate.

- You can hardly-ever eliminate absolute rounding error when using floating-point.
- The best we can do is to take steps to minimise error, and prevent it from increasing as your calculation progresses.
- Cancellation can be *catastrophic*, because it can greatly increase the *relative* error in your calculation.

2.31 Use a well-tested library for numerical algorithms.

- Avoid subtracting two nearly-equal numbers.
- Especially in a loop!
- Better-yet use a well-validated existing implementation in the form of a numerical library.

2.32 Importing numpy

- Functions for numerical computiing are provided by a separate *module* called `numpy`.
- Before we use the numpy module we must import it.
- By convention, we import `numpy` using the alias `np`.
- Once we have done this we can prefix the functions in the numpy library using the prefix `np.`

```
import numpy as np
```

- We can now use the functions defined in this package by prefixing them with `np`.

2.33 Arrays

- Arrays represent a collection of values.
- In contrast to lists:
  - arrays typically have a *fixed length*
    - they can be resized, but this involves an expensive copying process.
  - and all values in the array are of the *same type*.
    - typically we store floating-point values.
- Like lists:
  - arrays are *mutable*;
  - we can change the elements of an existing array.

2.34 Arrays in `numpy`

- Arrays are provided by the `numpy` module.
- The function `array()` creates an array given a list.

```
import numpy as np
x = np.array([0, 1, 2, 3, 4])
x
```

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

2.35 Array indexing

- We can index an array just like a list

```
x[4]
```

```
4
```

```
x[4] = 2
x
```

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

2.36 Arrays are not lists

- Although this looks a bit like a list of numbers, it is a fundamentally different type of value:

```
type(x)
```

```
numpy.ndarray
```

- For example, we cannot append to the array:

```
x.append(5)
```

```
---------------------------------------------------------------------------
```

```
AttributeError                            Traceback (most recent
↪ call last)
```

```
<ipython-input-32-7e52d4acf950> in <module>()
----> 1 x.append(5)
```

```
AttributeError: 'numpy.ndarray' object has no attribute 'append'
```

2.37 Populating Arrays

- To populate an array with a range of values we use the `np.arange()` function:

```
x = np.arange(0, 10)
print(x)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

- We can also use floating point increments.

```
x = np.arange(0, 1, 0.1)
print(x)
```

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
```

2.38 Functions over arrays

- When we use arithmetic operators on arrays, we create a new array with the result of applying the operator to each element.

```
y = x * 2
y
```

```
array([0. , 0.2, 0.4, 0.6, 0.8, 1. , 1.2, 1.4, 1.6, 1.8])
```

- The same goes for numerical functions:

```
x = np.array([-1, 2, 3, -4])
y = abs(x)
y
```

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

2.39 Vectorized functions

- Note that not every function automatically works with arrays.
- Functions that have been written to work with arrays of numbers are called *vectorized* functions.
- Most of the functions in `numpy` are already vectorized.
- You can create a vectorized version of any other function using the higher-order function `numpy.vectorize()`.

2.40 `vectorize` example

```
def myfunc(x):
    if x >= 0.5:
        return x
    else:
        return 0.0

fv = np.vectorize(myfunc)
```

```
x = np.arange(0, 1, 0.1)
x
```

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

```
fv(x)
```

```
array([0. , 0. , 0. , 0. , 0. , 0.5, 0.6, 0.7, 0.8, 0.9])
```

2.41 Testing for equality

- Because of finite precision we need to take great care when comparing floating-point values.
- The numpy function `allclose()` can be used to test equality of floating-point numbers within a relative tolerance.
- It is a vectorized function so it will work with arrays as well as single floating-point values.

```
x = 0.1 + 0.2
y = 0.3
x == y
```

```
False
```

```
np.allclose(x, y)
```

```
True
```

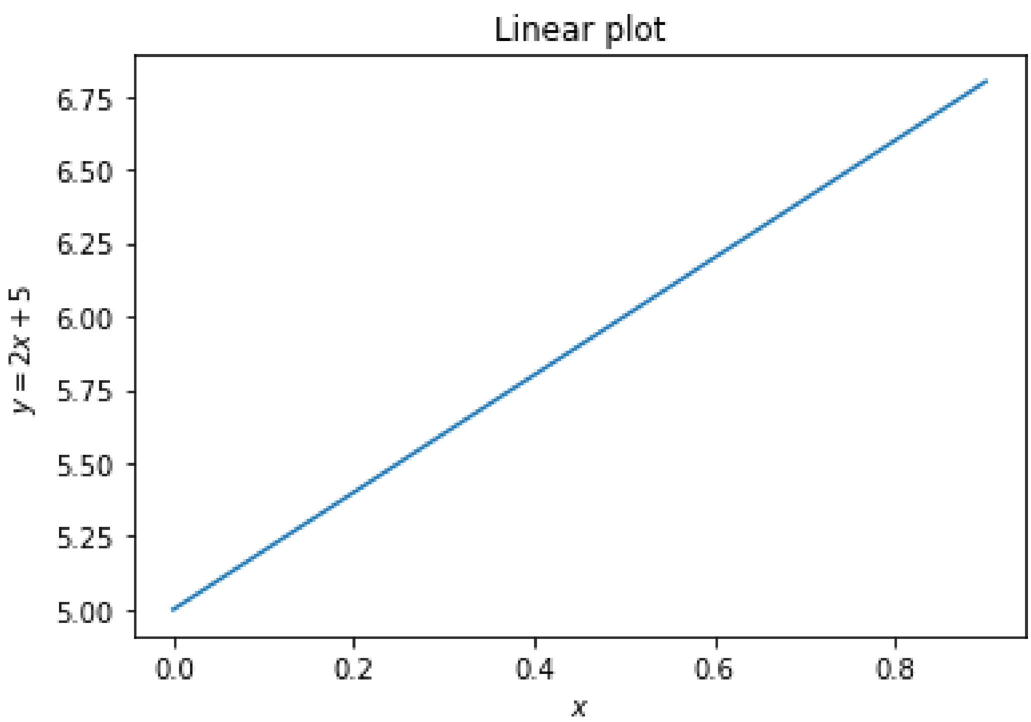
2.42 Plotting with `matplotlib`

- We will use a module called `matplotlib` to plot some simple graphs.
- This module has a nested module called `pyplot`.
- By convention we import this with the alias `plt`.
- This module provides functions which are very similar to MATLAB plotting commands.

```
import matplotlib.pyplot as plt
```

2.42.1 A simple linear plot

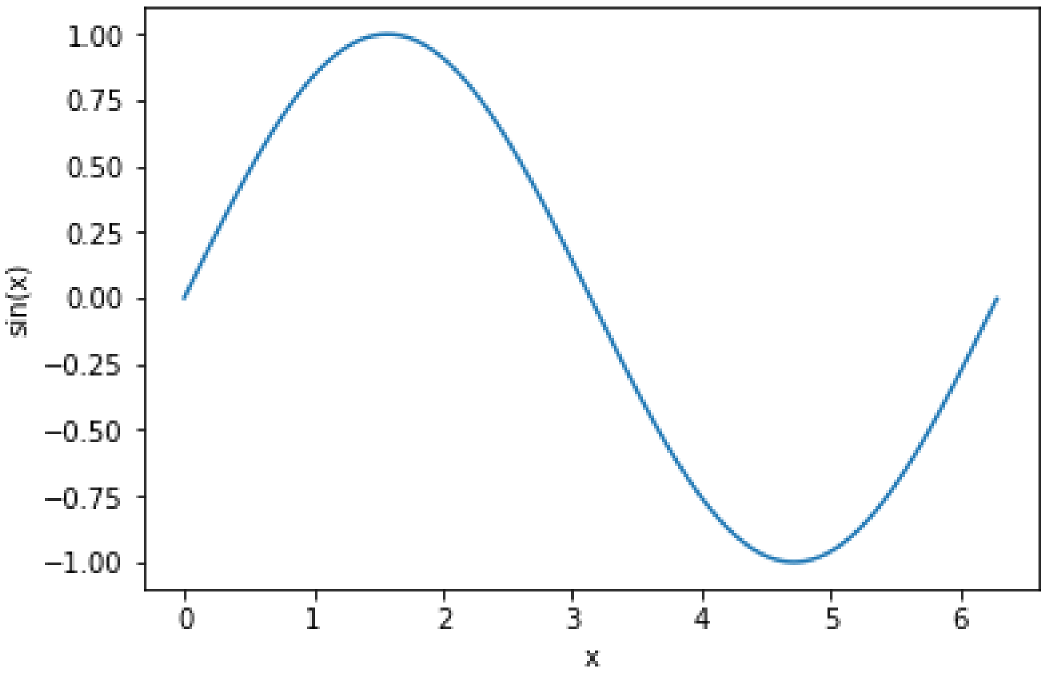
```
x = np.arange(0, 1, 0.1)
y = x*2 + 5
plt.plot(x, y)
plt.xlabel('$x$')
plt.ylabel('$y = 2x + 5$')
plt.title('Linear plot')
plt.show()
```



2.42.2 Plotting a sine curve

```
from numpy import pi, sin

x = np.arange(0, 2*pi, 0.01)
y = sin(x)
plt.plot(x, y)
plt.xlabel('x')
```

```
plt.ylabel('sin(x)')
plt.show()
```



2.43 Multi-dimensional data

- Numpy arrays can hold multi-dimensional data.
- To create a multi-dimensional array, we can pass a list of lists to the `array()` function:

```
import numpy as np

x = np.array([[1,2], [3,4]])
x
```

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

2.43.1 Arrays containing arrays

- A multi-dimensional array is an array of an arrays.
- The outer array holds the rows.
- Each row is itself an array:

```
x[0]
```

```
array([1, 2])
```

```
x[1]
```

```
array([3, 4])
```

- So the element in the second row, and first column is:

```
x[1][0]
```

```
3
```

2.43.2 Matrices

- We can create a matrix from a multi-dimensional array.

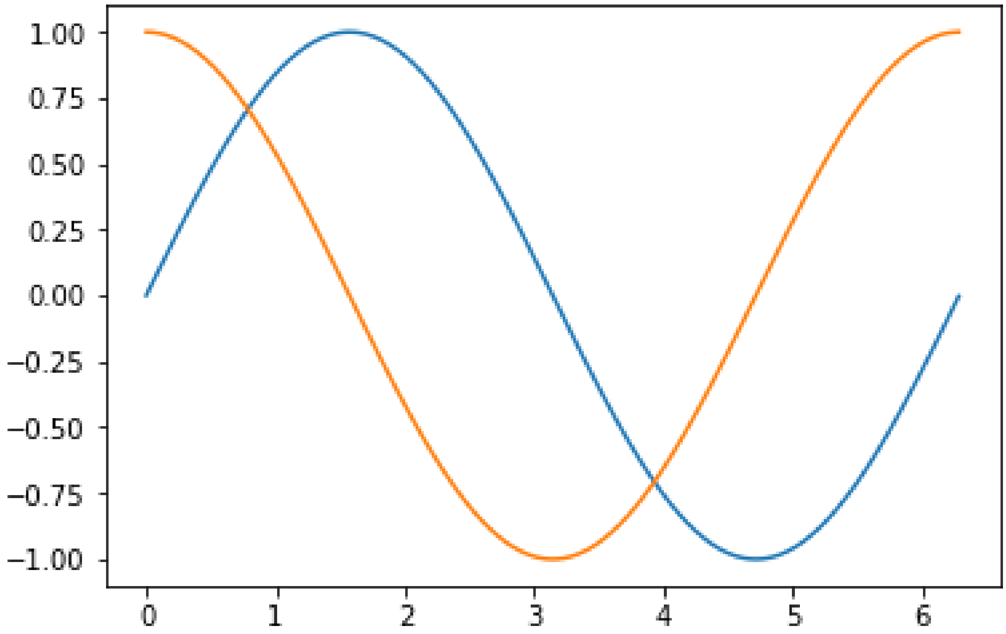
```
M = np.matrix(x)
M
```

```
matrix([[1, 2],
        [3, 4]])
```

2.43.3 Plotting multi-dimensional with matrices

- If we supply a matrix to `plot()` then it will plot the y-values taken from the *columns* of the matrix (notice the transpose in the example below).

```
from numpy import pi, sin, cos

x = np.arange(0, 2*pi, 0.01)
y = sin(x)
ax = plt.plot(x, np.matrix([sin(x), cos(x)]).T)
plt.show()
```



2.43.4 Performance

- When we use `numpy` matrices in Python the corresponding functions are linked with libraries written in C and FORTRAN.
- For example, see the BLAS (Basic Linear Algebra Subprograms) library.
- These libraries are very fast.
- Vectorised code can be more easily ported to frameworks like TensorFlow so that operations are performed in parallel using GPU hardware.

2.43.5 Matrix Operators

- Once we have a matrix, we can perform matrix computations.
- To compute the transpose and inverse use the `T` and `I` attributes:

To compute the transpose M^T

```
M.T
```

```
matrix([[1, 3],
        [2, 4]])
```

To compute the inverse M^{-1}

```
M.I
```

```
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
```

2.43.6 Matrix Dimensions

- The total number of elements, and the dimensions of the array:

```
M.size
```

```
4
```

```
M.shape
```

```
(2, 2)
```

```
len(M.shape)
```

```
2
```

2.43.7 Creating Matrices from strings

- We can also create arrays directly from strings, which saves some typing:

```
I2 = np.matrix('2 0; 0 2')
I2
```

```
matrix([[2, 0],
        [0, 2]])
```

- The semicolon starts a new row.

2.43.8 Matrix Multiplication

Now that we have two matrices, we can perform matrix multiplication:

```
M * I2
```

```
matrix([[2, 4],
        [6, 8]])
```

2.43.9 Matrix Indexing

- We can index and slice matrices using the same syntax as lists.

```
M[:,1]
```

```
matrix([[2],
        [4]])
```

2.43.10 Slices are references

- If we use this is an assignment, we create a *reference* to the sliced elements, *not* a copy.

```
V = M[:,1]  # This does not make a copy of the elements!
V
```

```
matrix([[2],
        [4]])
```

```
M[0,1] = -2
V
```

```
matrix([[-2],
        [ 4]])
```

2.43.11 Copying matrices and vectors

- To copy a matrix, or a slice of its elements, use the function `np.copy()`:

```
M = np.matrix('1 2; 3 4')
V = np.copy(M[:,1])  # This does copy the elements.
V
```

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

```
M[0,1] = -2
V
```

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

2.44 Sums

One way we *could* sum a vector or matrix is to use a `for` loop.

```
vector = np.arange(0.0, 100.0, 10.0)
vector
```

```
array([ 0., 10., 20., 30., 40., 50., 60., 70., 80., 90.])
```

```
result = 0.0
for x in vector:
    result = result + x
result
```

```
450.0
```

- This is not the most *efficient* way to compute a sum.

2.45 Efficient sums

- Instead of using a `for` loop, we can use a numpy function `sum()`.
- This function is written in the C language, and is very fast.

```
vector = np.array([0, 1, 2, 3, 4])
print(np.sum(vector))
```

```
10
```

2.46 Summing rows and columns

- When dealing with multi-dimensional data, the 'sum()' function has a named-argument `axis` which allows us to specify whether to sum along, each rows or columns.

```
matrix = np.matrix('1 2 3; 4 5 6; 7 8 9')
print(matrix)
```

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

2.46.1 To sum along rows:

```
np.sum(matrix, axis=0)
```

```
matrix([[12, 15, 18]])
```

2.46.2 To sum along columns:

```
np.sum(matrix, axis=1)
```

```
matrix([[ 6],
        [15],
        [24]])
```

2.47 Cumulative sums

- Suppose we want to compute $y_n = \sum_{i=1}^{n} x_i$ where $\vec{x}$ is a vector.

```
import numpy as np
x = np.array([0, 1, 2, 3, 4])
y = np.cumsum(x)
print(y)
```

```
[ 0  1  3  6 10]
```

2.48 Cumulative sums along rows and columns

```
x = np.matrix('1 2 3; 4 5 6; 7 8 9')
print(x)
```

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

```
y = np.cumsum(x)
np.cumsum(x, axis=0)
```

```
matrix([[ 1,  2,  3],
        [ 5,  7,  9],
        [12, 15, 18]])
```

```
np.cumsum(x, axis=1)
```

```
matrix([[ 1,  3,  6],
        [ 4,  9, 15],
        [ 7, 15, 24]])
```

2.49 Cumulative products

- Similarly we can compute $y_n = \Pi_{i=1}^{n} x_i$ using `cumprod()`:

```
import numpy as np
x = np.array([1, 2, 3, 4, 5])
np.cumprod(x)
```

```
array([  1,   2,   6,  24, 120])
```

- We can compute cummulative products along rows and columns using the `axis` parameter, just as with the `cumsum()` example.

2.50 Generating (pseudo) random numbers

- The nested module `numpy.random` contains functions for generating random numbers from different probability distributions.

```
from numpy.random import normal, uniform, exponential, randint
```

- Suppose that we have a random variable $\epsilon \sim N(0, 1)$.
- In Python we can draw from this distribution like so:

```
epsilon = normal()
epsilon
```

```
0.1465312427787133
```

- If we execute another call to the function, we will make a *new* draw from the distribution:

```
epsilon = normal()
epsilon
```

```
-
2.3135050810408773
```

2.51 Pseudo-random numbers

- Strictly speaking, these are not random numbers.
- They rely on an initial state value called the *seed*.
- If we know the seed, then we can predict with total accuracy the rest of the sequence, given any "random" number.
- Nevertheless, statistically they behave like independently and identically-distributed values.
  - Statistical tests for correlation and auto-correlation give insignificant results.
- For this reason they called *pseudo*-random numbers.
- The algorithms for generating them are called Pseudo-Random Number Generators (PRNGs).
- Some applications, such as cryptography, require genuinely unpredictable sequences.
  - never use a standard PRNG for these applications!

2.52 Managing seed values

- In some applications we need to reliably reproduce the same sequence of pseudo-random numbers that were used.
- We can specify the seed value at the beginning of execution to achieve this.
- Use the function `seed()` in the `numpy.random` module.

2.53 Setting the seed

```
from numpy.random import seed

seed(5)
```

```
normal()
```

```
0.44122748688504143
```

```
normal()
```

```
-
0.33087015189408764
```

```
seed(5)
```

```
normal()
```

```
0.44122748688504143
```

```
normal()
```

```
-
0.33087015189408764
```

2.54 Drawing multiple variates

- To generate more than number, we can specify the `size` parameter:

```
normal(size=10)
```

```
array([ 2.43077119, -0.25209213,  0.10960984,  1.58248112, -0.9092324 ,
       -0.59163666,  0.18760323, -0.32986996, -1.19276461, -
0.20487651])
```

- If you are generating very many variates, this will be *much* faster than using a for loop
- We can also specify more than one dimension:

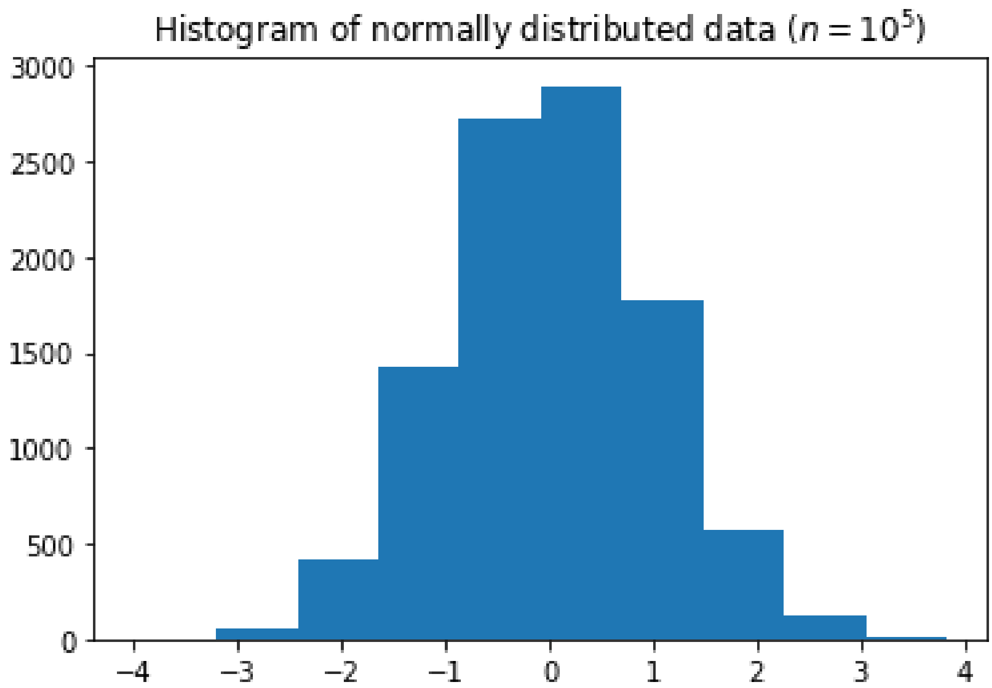
```
normal(size=(5,5))
```

```
array([[-0.35882895,  0.6034716 , -1.66478853, -0.70017904,
1.15139101],
       [ 1.85733101, -1.51117956,  0.64484751, -0.98060789, -
0.85685315],
       [-0.87187918, -0.42250793,  0.99643983,  0.71242127,
0.05914424],
       [-0.36331088,  0.00328884, -0.10593044,  0.79305332, -
0.63157163],
       [-0.00619491, -0.10106761, -0.05230815,  0.24921766,
0.19766009]])
```

2.55 Histograms

- We can plot a histograms of randomly-distributed data using the `hist()` function from matplotlib:

```
import matplotlib.pyplot as plt

data = normal(size=10000)
ax = plt.hist(data)
plt.title('Histogram of normally distributed data ($n=10^5$)')
plt.show()
```



2.56 Computing histograms as matrices

- The function `histogram()` in the `numpy` module will count frequencies into bins and return the result as a 2-dimensional array.

```
import numpy as np
np.histogram(data)
```

```
(array([  23,   136,   618,  1597,  2626,  2635,  1620,   599,   130,    16]),
 array([-3.59780883, -2.87679609, -2.15578336, -1.43477063, -
0.71375789,
         0.00725484,  0.72826758,  1.44928031,  2.17029304,
2.89130578,
         3.61231851]))
```

2.57 Descriptive statistics

- We can compute the descriptive statistics of a sample of values using the numpy functions `mean()` and `var()` to compute the sample mean $\bar{X}$ and sample variance σ_X^2 .

```
np.mean(data)
```

```
-
0.00045461080333497925
```

```
np.var(data)
```

```
1.0016048722546331
```

- These functions also have an `axis` parameter to compute mean and variances of columns or rows of a multi-dimensional data-set.

2.58 Descriptive statistics with `nan` values

- If the data contains `nan` values, then the descriptive statistics will also be `nan`.

```
from numpy import nan
import numpy as np
data = np.array([1, 2, 3, 4, nan])
np.mean(data)
```

```
nan
```

- To omit `nan` values from the calculation, use the functions `nanmean()` and `nanvar()`:

```
np.nanmean(data)
```

```
2.5
```

2.59 Discrete random numbers

- The `randint()` function in `numpy.random` can be used to draw from a uniform discrete probability distribution.
- It takes two parameters: the low value (inclusive), and the high value (exclusive).
- So to simulate one roll of a die, we would use the following Python code.

```
die_roll = randint(0, 6) + 1
die_roll
```

```
4
```

- Just as with the `normal()` function, we can generate an entire sequence of values.
- To simulate a Bernoulli process with $n = 20$ trials:

```
bernoulli_trials = randint(0, 2, size = 20)
bernoulli_trials
```

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

Acknowledgements

The early sections of this notebook were adapted from an online article by Steve Hollasch.

3 Financial data with data frames

(c) 2019 Steve Phelps

3.1 Data frames

- The `pandas` module provides a powerful data-structure called a data frame.
- It is similar, but not identical to:
  - a table in a relational database,
  - an Excel spreadsheet,
  - a dataframe in R.

3.1.1 Types of data

Data frames can be used to represent:

- Panel data
- Time series data
- Relational data

3.1.2 Loading data

- Data frames can be read and written to/from:
  - financial web sites
  - database queries
  - database tables
  - CSV files
  - json files
- Beware that data frames are memory resident;
  - If you read a large amount of data your PC might crash
  - With big data, typically you would read a subset or summary of the data via e.g. a select statement.

3.2 Importing pandas

- The pandas module is usually imported with the alias `pd`.

```
import pandas as pd
```

3.3 Series

- A Series contains a one-dimensional array of data, *and* an associated sequence of labels called the *index*.
- The index can contain numeric, string, or date/time values.
- When the index is a time value, the series is a time series.
- The index must be the same length as the data.
- If no index is supplied it is automatically generated as `range(len(data))`.

3.3.1 Creating a series from an array

```python
import numpy as np
data = np.random.randn(5)
data
```

```
array([ 0.03245675,  0.41263151, -0.27993028, -0.95398035, -
0.01473876])
```

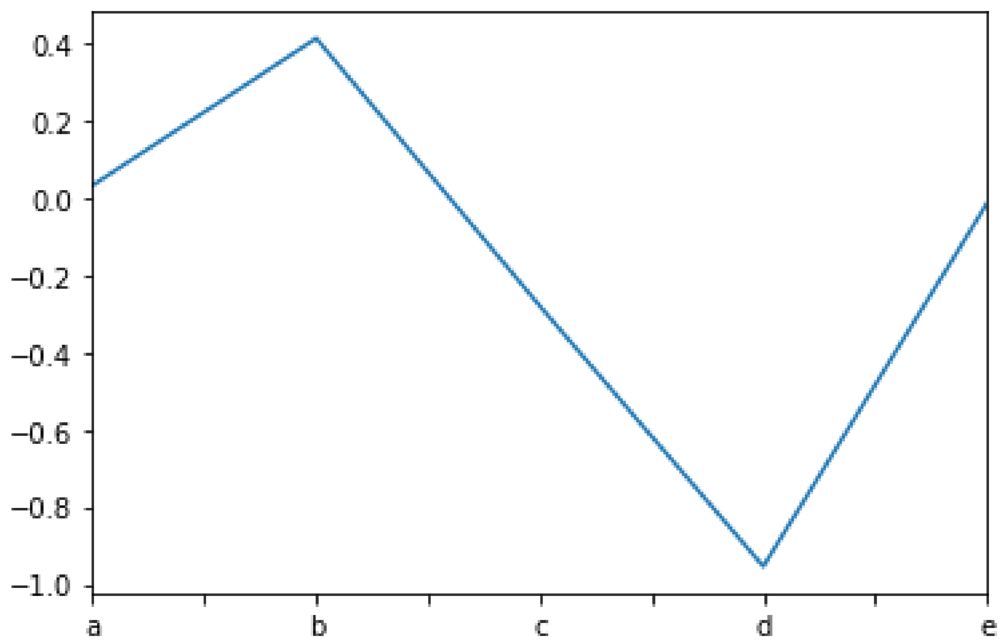
```python
my_series = pd.Series(data, index=['a', 'b', 'c', 'd', 'e'])
my_series
```

```
a     0.032457
b     0.412632
c    -0.279930
d    -0.953980
e    -0.014739
dtype: float64
```

3.3.2 Plotting a series

- We can plot a series by invoking the `plot()` method on an instance of a `Series` object.
- The x-axis will autimatically be labelled with the series index.

```python
import matplotlib.pyplot as plt
my_series.plot()
plt.show()
```



3.3.3 Creating a series with automatic index

- In the following example the index is creating automatically:

```
pd.Series(data)
```

```
0     0.032457
1     0.412632
2    -0.279930
3    -0.953980
4    -0.014739
dtype: float64
```

3.3.4 Creating a Series from a `dict`

```
d = {'a' : 0., 'b' : 1., 'c' : 2.}
my_series = pd.Series(d)
my_series
```

```
a     0.0
b     1.0
c     2.0
dtype: float64
```

3.3.5 Indexing a series with `[]`

- Series can be accessed using the same syntax as arrays and dicts.
- We use the labels in the index to access each element.

```
my_series['b']
```

```
1.0
```

- We can also use the label like an attribute:

```
my_series.b
```

```
1.0
```

3.3.6 Slicing a series

- We can specify a range of labels to obtain a slice:

```
my_series[['b', 'c']]
```

```
b     1.0
c     2.0
dtype: float64
```

3.4 Arithmetic and vectorised functions

- numpy vectorization works for series objects too.

```
d = {'a' : 0., 'b' : 1., 'c' : 2.}
squared_values = pd.Series(d) ** 2
squared_values
```

```
a    0.0
b    1.0
c    4.0
dtype: float64
```

```
x = pd.Series({'a' : 0., 'b' : 1., 'c' : 2.})
y = pd.Series({'a' : 3., 'b' : 4., 'c' : 5.})
x + y
```

```
a    3.0
b    5.0
c    7.0
dtype: float64
```

3.5 Time series

```
dates = pd.date_range('1/1/2000', periods=5)
dates
```

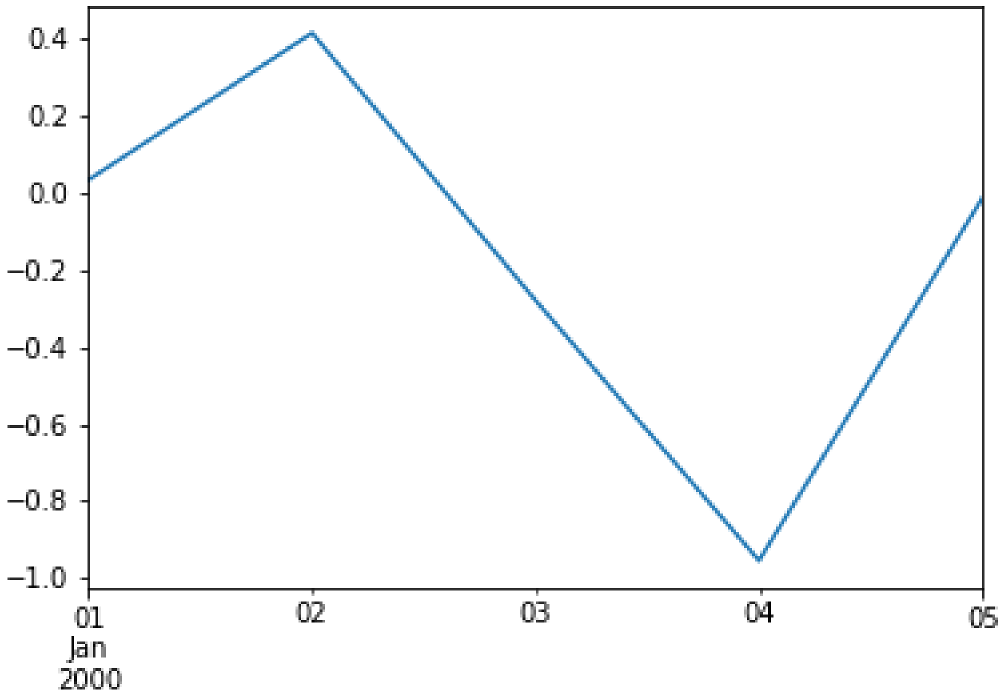
```
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04',
               '2000-01-05'],
              dtype='datetime64[ns]', freq='D')
```

```
time_series = pd.Series(data, index=dates)
time_series
```

```
2000-01-01    0.032457
2000-01-02    0.412632
2000-01-03   -0.279930
2000-01-04   -0.953980
2000-01-05   -0.014739
Freq: D, dtype: float64
```

3.5.1 Plotting a time-series

```
ax = time_series.plot()
```



3.6 Missing values

- Pandas uses `nan` to represent missing data.
- So `nan` is used to represent missing, invalid or unknown data values.
- It is important to note that this only convention only applies within pandas.
  - Other frameworks have very different semantics for these values.

3.7 DataFrame

- A data frame has multiple columns, each of which can hold a *different* type of value.
- Like a series, it has an index which provides a label for each and every row.
- Data frames can be constructed from:
  - dict of arrays,
  - dict of lists,
  - dict of dict
  - dict of Series
  - 2-dimensional array
  - a single Series
  - another DataFrame

3.8 Creating a dict of series

```
series_dict = {
        'x' :
            pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
```

```
            'y' :
                pd.Series([4., 5., 6., 7.], index=['a', 'b', 'c', 'd']),
            'z' :
                pd.Series([0.1, 0.2, 0.3, 0.4], index=['a', 'b', 'c', 'd'])
}

series_dict
```

```
{'x': a    1.0
 b    2.0
 c    3.0
 dtype: float64, 'y': a    4.0
 b    5.0
 c    6.0
 d    7.0
 dtype: float64, 'z': a    0.1
 b    0.2
 c    0.3
 d    0.4
 dtype: float64}
```

3.9 Converting the dict to a data frame

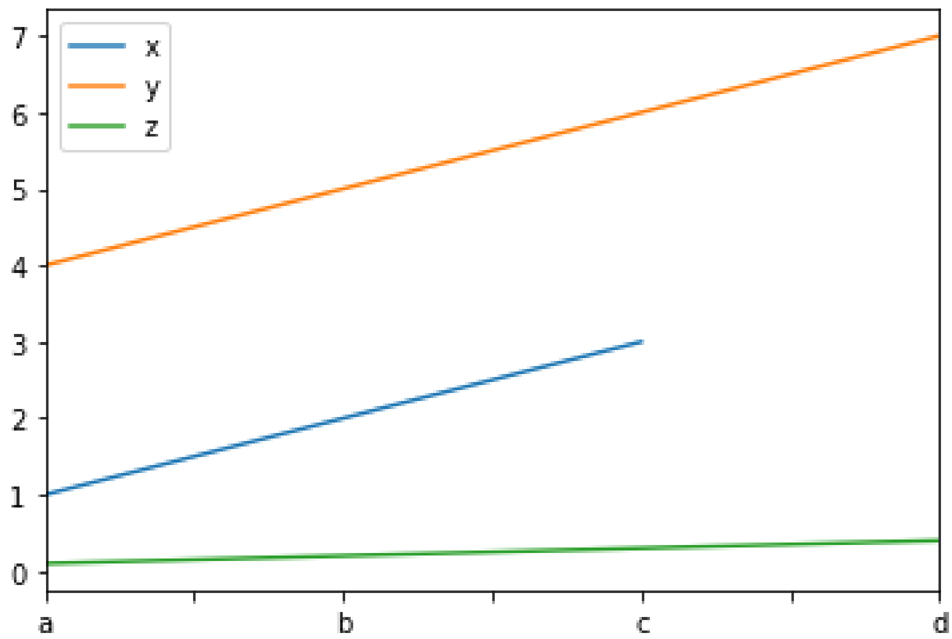
```
df = pd.DataFrame(series_dict)
df
```

```
     x    y    z
a  1.0  4.0  0.1
b  2.0  5.0  0.2
c  3.0  6.0  0.3
d  NaN  7.0  0.4
```

3.10 Plotting data frames

- When plotting a data frame, each column is plotted as its own series on the same graph.
- The column names are used to label each series.
- The row names (index) is used to label the x-axis.

```
ax = df.plot()
```



3.11 Indexing

- The outer dimension is the column index.
- When we retrieve a single column, the result is a Series

```
df['x']
```

```
a     1.0
b     2.0
c     3.0
d     NaN
Name: x, dtype: float64
```

```
df['x']['b']
```

```
2.0
```

```
df.x.b
```

```
2.0
```

3.12 Projections

- Data frames can be sliced just like series.
- When we slice columns we call this a *projection*, because it is analogous to specifying a subset of attributes in a relational query, e.g. `SELECT x FROM table`.
- If we project a single column the result is a series:

```python
slice = df['x'][['b', 'c']]
slice
```

```
b     2.0
c     3.0
Name: x, dtype: float64
```

```python
type(slice)
```

```
pandas.core.series.Series
```

3.13 Projecting multiple columns

- When we include multiple columns in the projection the result is a DataFrame.

```python
slice = df[['x', 'y']]
slice
```

```
     x    y
a  1.0  4.0
b  2.0  5.0
c  3.0  6.0
d  NaN  7.0
```

```python
type(slice)
```

```
pandas.core.frame.DataFrame
```

3.14 Vectorization

- Vectorized functions and operators work just as with series objects:

```python
df['x'] + df['y']
```

```
a     5.0
b     7.0
c     9.0
d     NaN
dtype: float64
```

```python
df ** 2
```

```
     x     y     z
a  1.0  16.0  0.01
b  4.0  25.0  0.04
c  9.0  36.0  0.09
d  NaN  49.0  0.16
```

3.15 Logical indexing

- We can use logical indexing to retrieve a subset of the data.

```
df['x'] >= 2
```

```
a    False
b     True
c     True
d    False
Name: x, dtype: bool
```

```
df[df['x'] >= 2]
```

```
     x    y    z
b  2.0  5.0  0.2
c  3.0  6.0  0.3
```

3.16 Descriptive statistics

- To quickly obtain descriptive statistics on numerical values use the `describe` method.

```
df.describe()
```

```
         x         y         z
count  3.0  4.000000  4.000000
mean   2.0  5.500000  0.250000
std    1.0  1.290994  0.129099
min    1.0  4.000000  0.100000
25%    1.5  4.750000  0.175000
50%    2.0  5.500000  0.250000
75%    2.5  6.250000  0.325000
max    3.0  7.000000  0.400000
```

3.17 Accessing a single statistic

- The result is itself a DataFrame, so we can index a particular statistic like so:

```
df.describe()['x']['mean']
```

```
2.0
```

3.18 Accessing the row and column labels

- The row labels (index) and column labels can be accessed:

```
df.index
```

```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

```
df.columns
```

```
Index(['x', 'y', 'z'], dtype='object')
```

3.19 Head and tail

- Data frames have `head()` and `tail()` methods which behave analgously to the Unix commands of the same name.

3.20 Financial data

- Pandas was originally developed to analyse financial data.
- We can download tabulated data in a portable format called Comma Separated Values (CSV).

```
import pandas as pd
googl = pd.read_csv('data/GOOGL.csv')
```

3.20.1 Examining the first few rows

- When working with large data sets it is useful to view just the first/last few rows in the dataset.
- We can use the `head()` method to retrieve the first rows:

```
googl.head()
```

```
         Date        Open        High         Low       Close    Adj
Close  \
0  2013-11-13  503.878876  516.941956  503.753754  516.751770
516.751770
1  2013-11-14  517.477478  520.395386  515.690674  518.133118
518.133118
2  2013-11-15  517.952942  519.519531  515.670654  517.297302
517.297302
3  2013-11-18  518.393372  524.894897  515.135132  516.291321
516.291321
4  2013-11-19  516.376404  517.892883  512.037048  513.113098
513.113098

    Volume
0  3155600
1  2331000
2  2550000
3  3515800
4  2260900
```

3.20.2 Examining the last few rows

```
googl.tail()
```

```
             Date         Open         High          Low        Close  \
1505   2019-11-06  1290.089966  1292.989990  1282.270020  1291.010010
1506   2019-11-07  1294.280029  1322.650024  1293.750000  1306.939941
1507   2019-11-08  1301.520020  1317.109985  1301.520020  1309.000000
1508   2019-11-11  1304.000000  1304.900024  1295.869995  1298.280029
1509   2019-11-12  1298.569946  1309.349976  1294.239990  1297.209961

        Adj Close   Volume
1505  1291.010010  1231300
1506  1306.939941  2257000
1507  1309.000000  1519600
1508  1298.280029   861700
1509  1297.209961  1442600
```

3.20.3 Converting to datetime values

- So far, the `Date` attribute is of type string.

```
googl.Date[0]
```

```
'2013-11-13'
```

```
type(googl.Date[0])
```

```
str
```

- In order to work with time-series data, we need to construct an index containing time values.
- Time values are of type `datetime` or `Timestamp`.
- We can use the function `to_datetime()` to convert strings to time values.

```
pd.to_datetime(googl['Date']).head()
```

```
0   2013-11-13
1   2013-11-14
2   2013-11-15
3   2013-11-18
4   2013-11-19
Name: Date, dtype: datetime64[ns]
```

3.20.4 Setting the index

- Now we need to set the index of the data-frame so that it contains the sequence of dates.

```
googl.set_index(pd.to_datetime(googl['Date']), inplace=True)
googl.index[0]
```

```
Timestamp('2013-11-13 00:00:00')
```

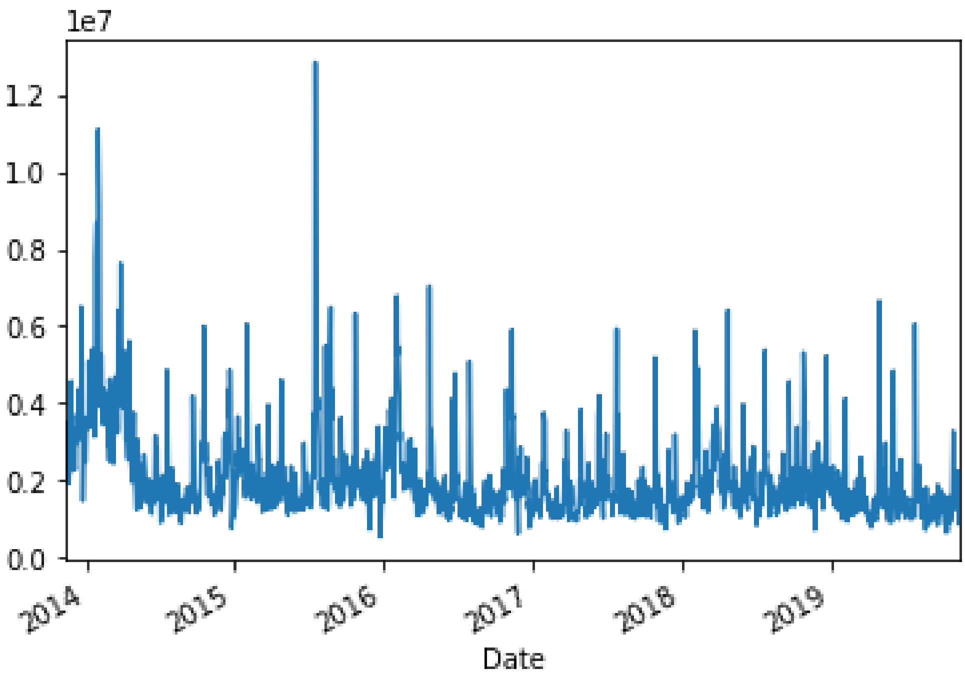
```python
type(googl.index[0])
```

```
pandas._libs.tslibs.timestamps.Timestamp
```

3.20.5 Plotting series

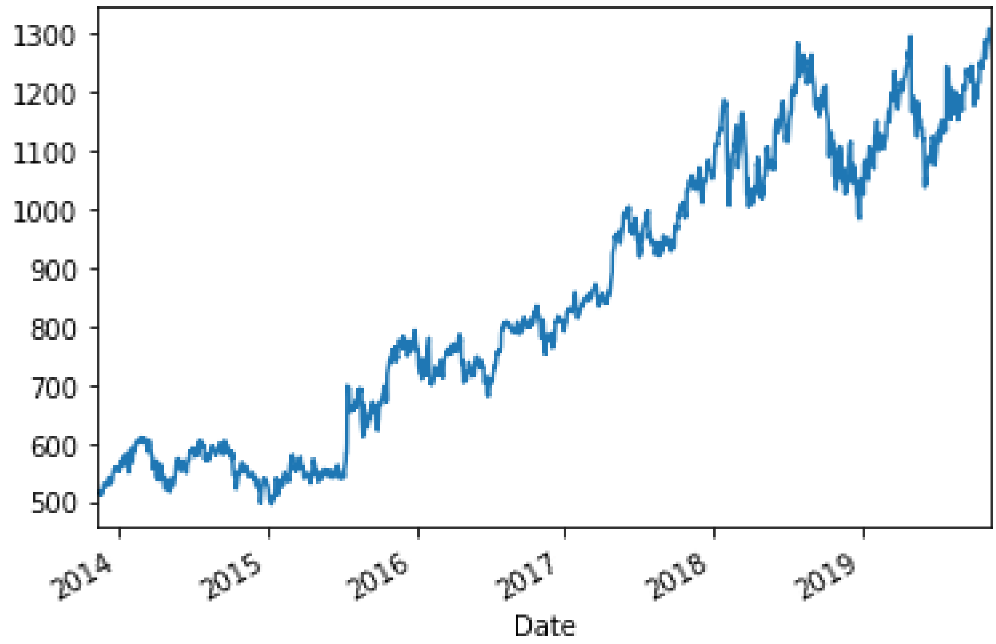
- We can plot a series in a dataframe by invoking its `plot()` method.
- Here we plot a time-series of the daily traded volume:

```python
ax = googl['Volume'].plot()
plt.show()
```



3.20.6 Adjusted closing prices as a time series

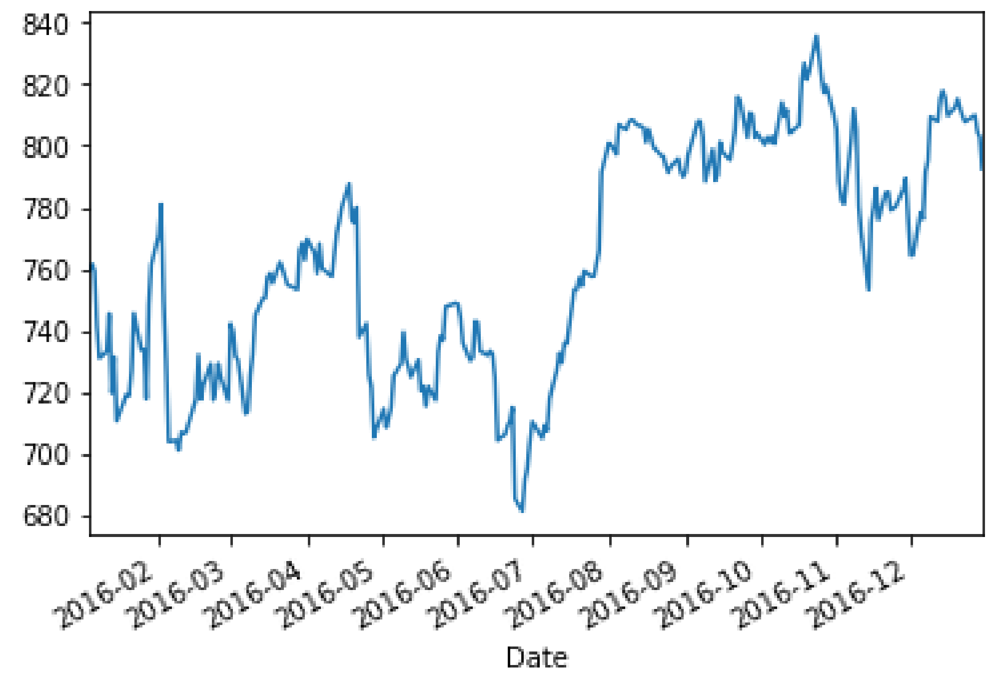
```python
googl['Adj Close'].plot()
plt.show()
```



3.20.7 Slicing series using date/time stamps

- We can slice a time series by specifying a range of dates or times.
- Date and time stamps are specified strings representing dates in the required format.

```
googl['Adj Close']['1-1-2016':'1-1-2017'].plot()
plt.show()
```



3.20.8 Resampling

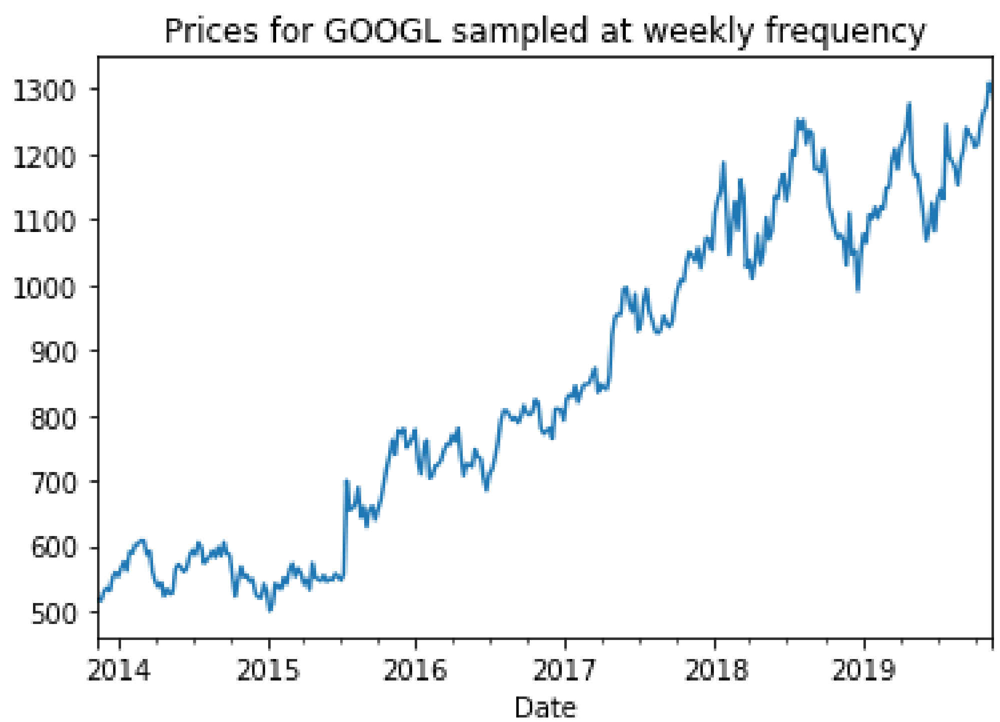
- We can *resample* to obtain e.g. weekly or monthly prices.
- In the example below the `'W'` denotes weekly.
- See the documentation for other frequencies.
- We group data into weeks, and then take the last value in each week.
- For details of other ways to resample the data, see the documentation.

3.20.8.1 Resampled time-series plot

```
weekly_prices = googl['Adj Close'].resample('W').last()
weekly_prices.head()
```

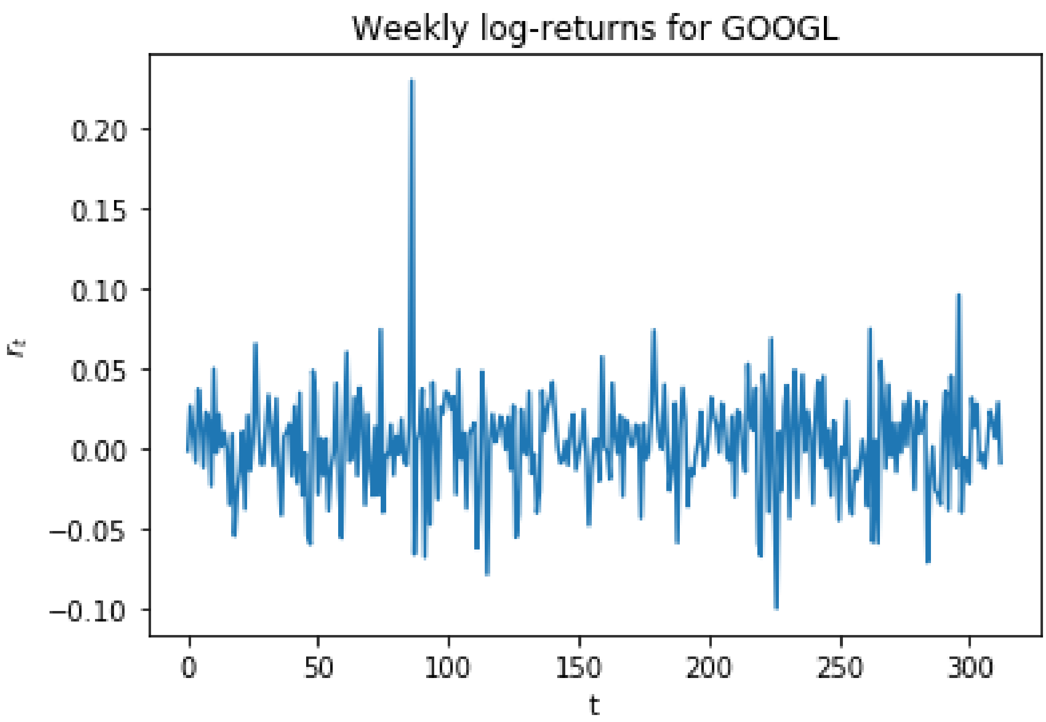
```
Date
2013-11-17    517.297302
2013-11-24    516.461487
2013-12-01    530.325317
2013-12-08    535.470459
2013-12-15    530.925903
Freq: W-SUN, Name: Adj Close, dtype: float64
```

```
weekly_prices.plot()
plt.title('Prices for GOOGL sampled at weekly frequency')
plt.show()
```



3.20.9 Converting prices to log returns

```
weekly_rets = np.diff(np.log(weekly_prices))
plt.plot(weekly_rets)
plt.xlabel('t'); plt.ylabel('$r_t$')
plt.title('Weekly log-returns for GOOGL')
plt.show()
```



3.20.10 Converting the returns to a series

- Notice that in the above plot the time axis is missing the dates.
- This is because the `np.diff()` function returns an array instead of a data-frame.

```
type(weekly_rets)
```

```
numpy.ndarray
```

- We can convert it to a series thus:

```
weekly_rets_series = pd.Series(weekly_rets, index=weekly_prices.
    ↪ index[1:])
weekly_rets_series.head()
```

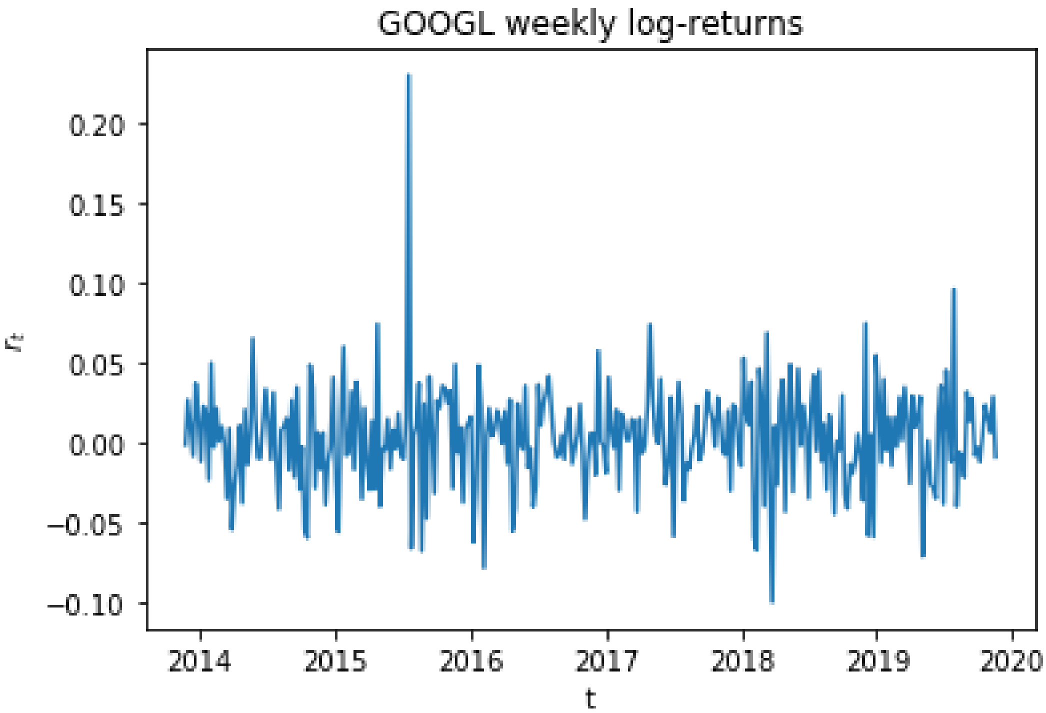
```
Date
2013-11-24   -0.001617
2013-12-01    0.026490
2013-12-08    0.009655
2013-12-15   -0.008523
2013-12-22    0.036860
```

```
Freq: W-SUN, dtype: float64
```

3.20.10.1 Plotting with the correct time axis

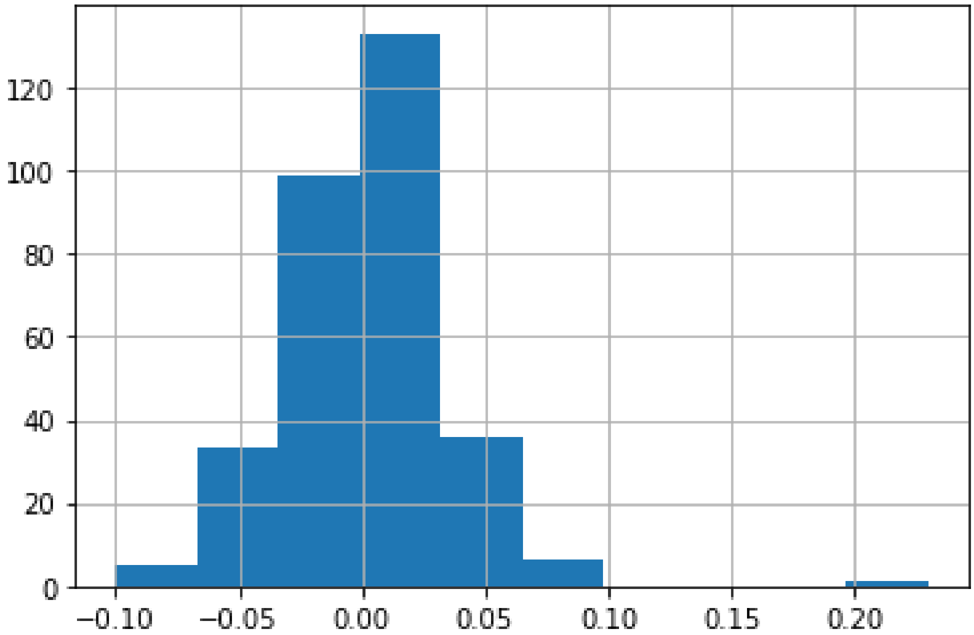
Now when we plot the series we will obtain the correct time axis:

```
plt.plot(weekly_rets_series)
plt.title('GOOGL weekly log-returns'); plt.xlabel('t'); plt.ylabel('
    ↪ $r_t$')
plt.show()
```



3.20.11 Plotting a return histogram

```
weekly_rets_series.hist()
plt.show()
```




```
weekly_rets_series.describe()
```

```
count    313.000000
mean       0.002937
std        0.032039
min       -0.099918
25%       -0.013341
50%        0.004653
75%        0.021327
max        0.229571
dtype: float64
```

4 Statistics and optimization with SciPy

4.1 The SciPy library

- SciPy is a library that provides several modules for scientific computing.
- You can read more about it by reading the reference guide.
- It provides modules for:
  - Solving optimization problems.
  - Linear algebra.
  - Interpolation.
  - Statistical inference.
  - Fourier transform.
  - Numerical differentation and integration.

4.2 Overview

1. loading data with pandas,
2. computing returns,
3. Quantile-Quantile (q-q) plots,
4. The Jarque-Bera test for normally-distributed data,
5. ordinary-least squares (OLS) regression.
6. Portfolio optimization

4.3 Loading data into a pandas dataframe

- We will first obtain some data from Yahoo finance using the pandas library.
- First we will import the functions and modules we need.

```python
import matplotlib.pyplot as plt
import datetime
import pandas as pd
import numpy as np
```

4.4 Downloading price data using as CSV

- Here we obtain price data on Microsoft Corporation Common Stock, so we specify the symbol MSFT.

```python
def prices_from_csv(fname):
    df = pd.read_csv(fname)
    df.set_index(pd.to_datetime(df['Date']), inplace=True)
    return df
```

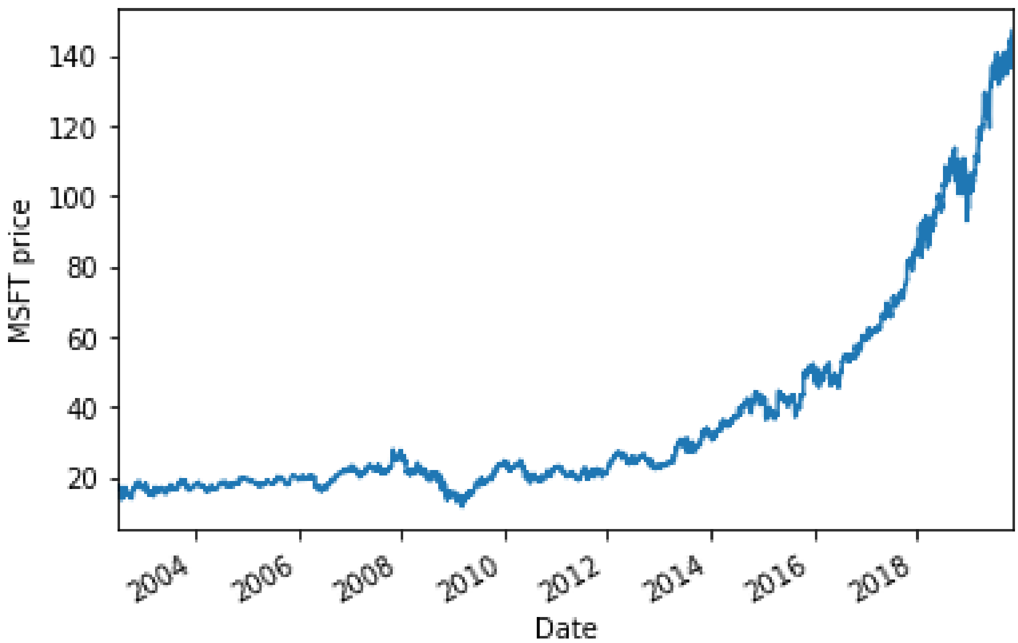
```python
msft = prices_from_csv('data/MSFT.csv')
msft.head()
```

```
                  Date       Open       High        Low      Close
Adj Close  \
Date
2002-07-01  2002-07-01  27.059999  27.195000  26.290001  26.330000
```

```
16.997240
2002-07-02  2002-07-02  26.190001  26.459999  25.665001  25.719999
16.603462
2002-07-03  2002-07-03  25.620001  26.260000  25.225000  25.920000
16.732574
2002-07-05  2002-07-05  26.545000  27.450001  26.525000  27.424999
17.704121
2002-07-08  2002-07-08  27.205000  27.465000  26.290001  26.459999
17.081167

               Volume
Date
2002-07-01  66473800
2002-07-02  82814200
2002-07-03  80936600
2002-07-05  35673600
2002-07-08  63199400
```

4.5 Plotting the price of the stock

```
msft['Adj Close'].plot()
plt.ylabel('MSFT price')
plt.show()
```

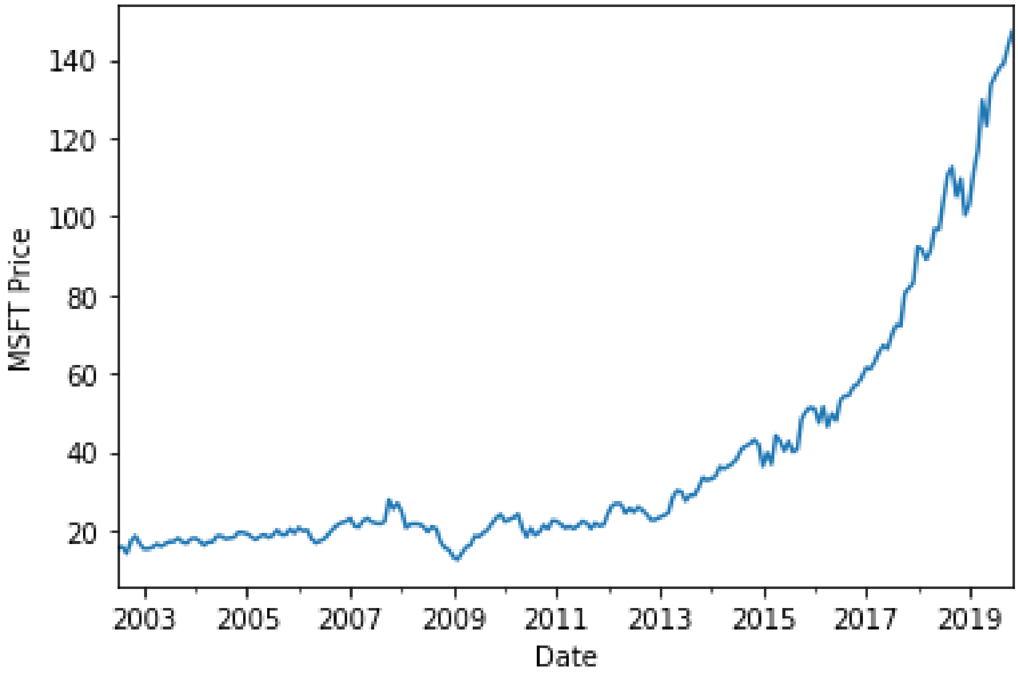


4.6 Converting to monthly data

- We will resample the data at a frequency of one calendar month.
- The code below takes the last price in every month.

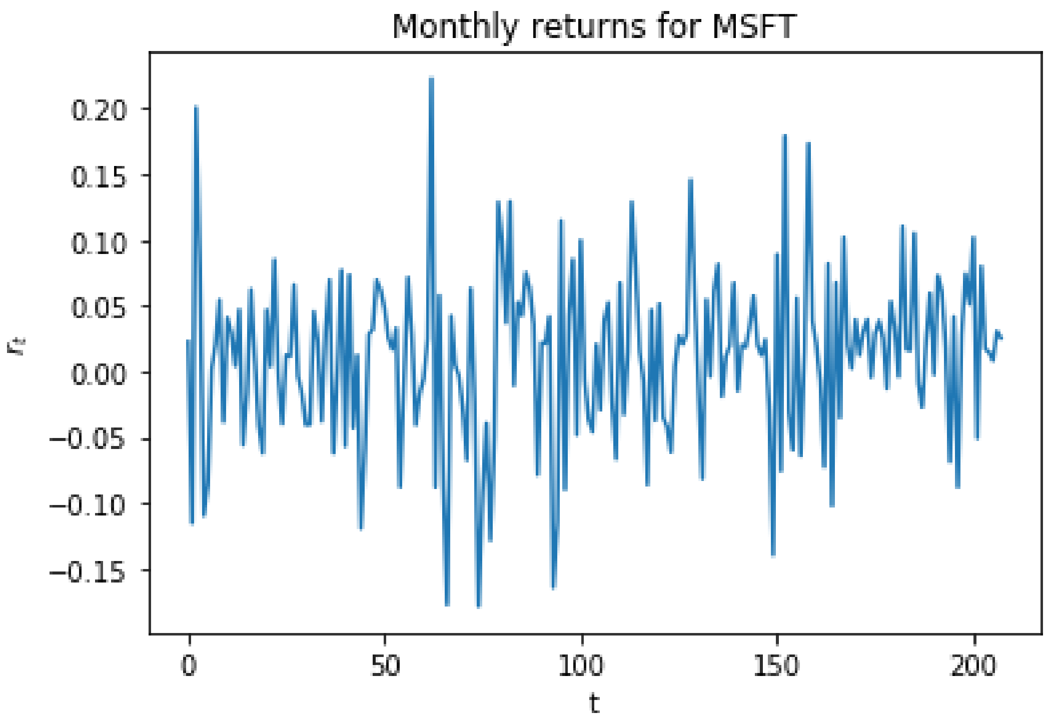
```
daily_prices = msft['Adj Close']
```

```
monthly_prices = daily_prices.resample('M').last()
monthly_prices.plot()
plt.ylabel('MSFT Price')
plt.show()
```



4.7 Calculating log returns

```
stock_returns = np.diff(np.log(monthly_prices))
plt.plot(stock_returns)
plt.xlabel('t'); plt.ylabel('$r_t$')
plt.title('Monthly returns for MSFT')
plt.show()
```

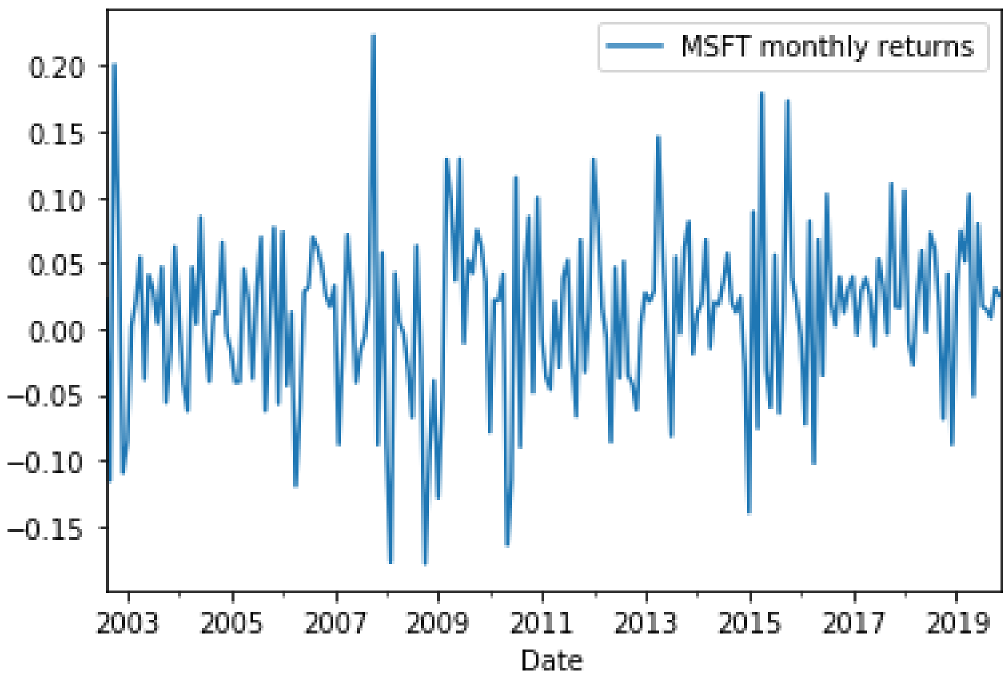


4.8 Converting the returns to a data frame

```
stock_returns_df = pd.DataFrame({'MSFT monthly returns':
    ↪ stock_returns}, index=monthly_prices.index[1:])
stock_returns_df.tail()
```

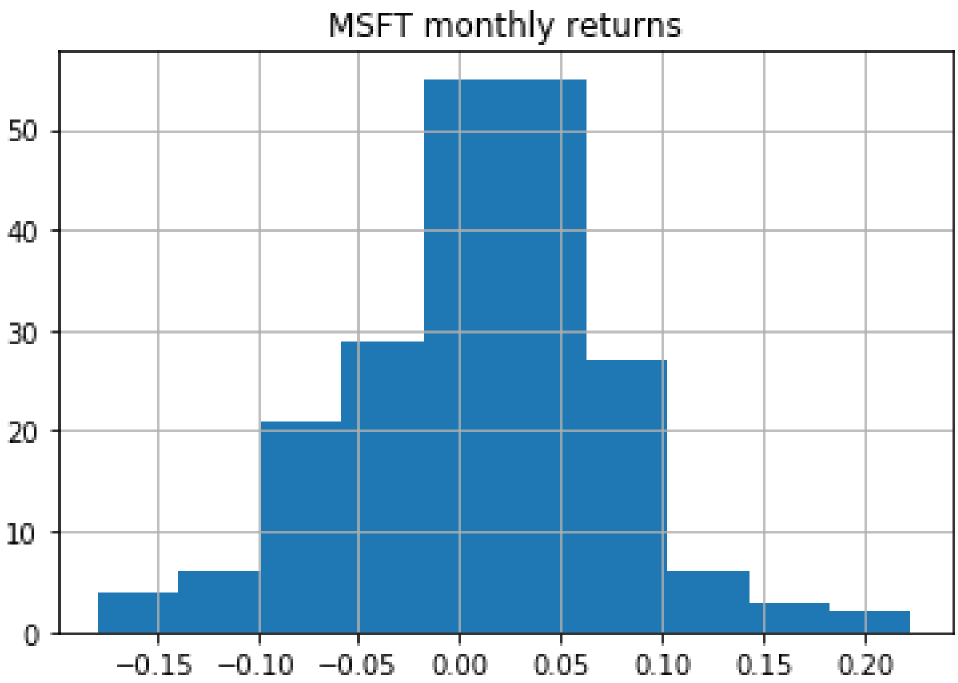
```
               MSFT monthly returns
Date
2019-07-31                 0.017097
2019-08-31                 0.014925
2019-09-30                 0.008451
2019-10-31                 0.030739
2019-11-30                 0.025480
```

```
stock_returns_df.plot()
plt.show()
```



4.9 Return histogram

```
stock_returns_df.hist()
plt.show()
```



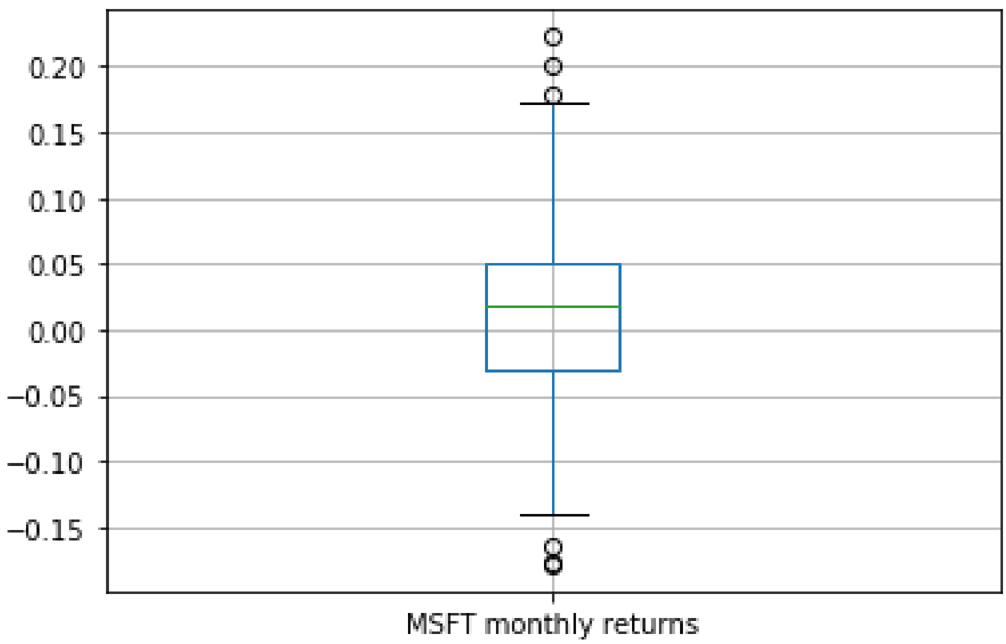
4.10 Descriptive statistics of the return distribution

```
stock_returns_df.describe()
```

```
       MSFT monthly returns
count            208.000000
mean               0.010822
std                0.064673
min               -0.178358
25%               -0.031284
50%                0.018196
75%                0.051398
max                0.222736
```

4.11 Summarising the distribution using a boxplot

```
stock_returns_df.boxplot()
plt.show()
```

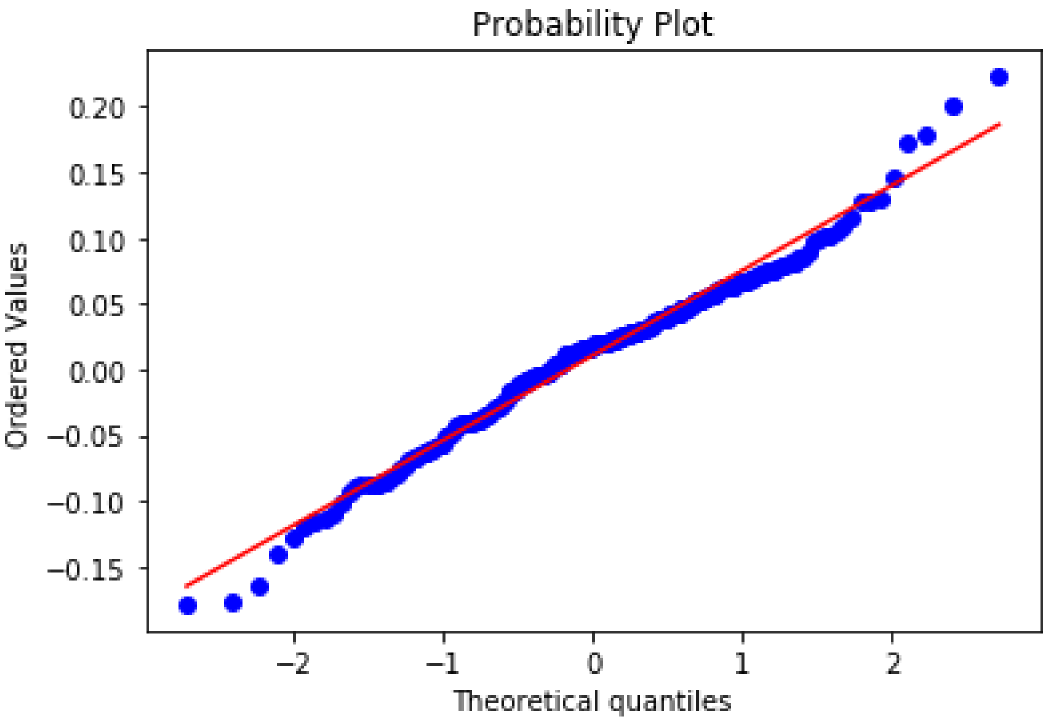


4.12 Q-Q plots

- Quantile-Quantile (Q-Q) plots are a useful way to compare distributions.
- We plot empirical quantiles against the quantiles computed the inverted c.d.f. of a specified theoretical distribution.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats

stats.probplot(stock_returns, dist="norm", plot=plt)
```

```
plt.show()
```



4.13 The Jarque-Bera Test

- The Jarque-Bera (JB) test is a statistical test that can be used to test whether a given sample was drawn from a normal distribution.
- The null hypothesis is that the data have the same skewness (0) and kurtosis (3) as a normal distribution.
- The test statistic is:

$$JB = \frac{n}{6}(S^2 + \frac{1}{4}(K-3)^2) \tag{4.1}$$

where S is the sample skewness, K is the sample kurtosis, and n is the number of observations.

- It is implemented in `scipy.stats.jarque_bera()`.

4.13.0.1 References

Jarque, C. and Bera, A. (1980) "Efficient tests for normality, homoscedasticity and serial independence of regression residuals", 6 Econometric Letters 255-259.

4.14 The Jarque-Bera test using a bootstrap

- We can test against the *null hypothesis* of S=0 and K=3.
- A finite sample can exhibit non-zero skewness and excess kurtosis simply due to sample noise, even if the distribution is Guassian.

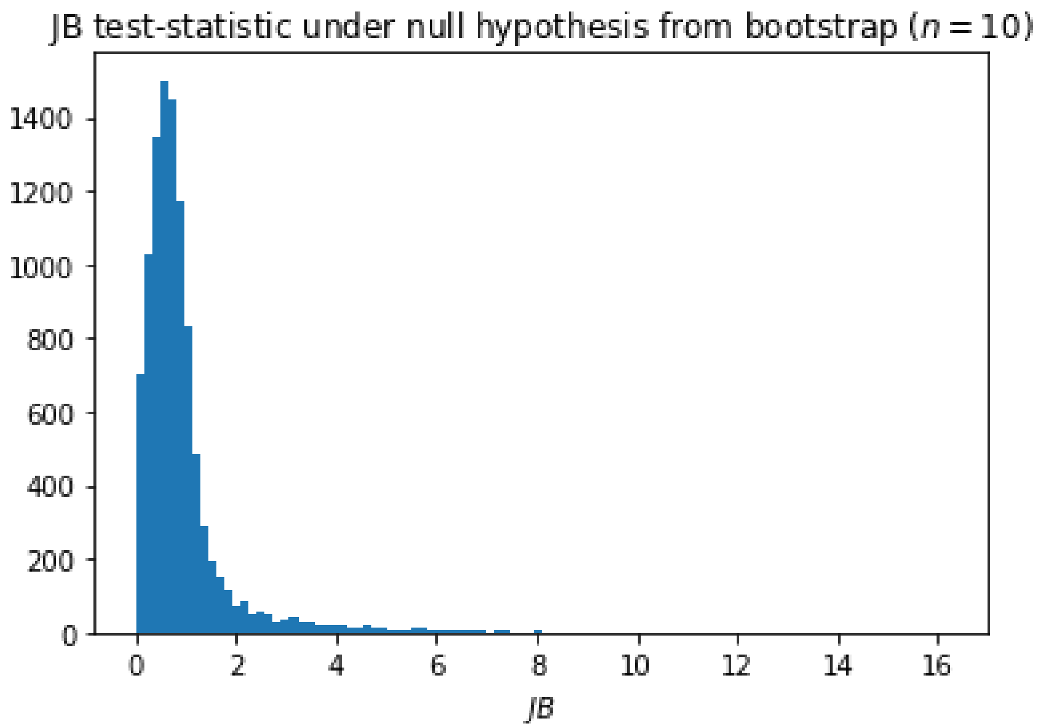
- What is the distribution of the sum of the squared sample skewness and kurtosis under repeated sampling?
- We can answer this question using a Monte-Caro method called bootstrapping.
  - Note that this is very expensive, and we would not always do this in practice (see the subsequent slides).

4.14.1 Bootstrap code

```
from scipy.stats import skew, kurtosis

def jb(n, s, k):
    return  n / 6. * (s**2 + (((k - 3.)**2) / 4.))

def jb_from_samples(n, bootstrap_samples):
    s = skew(bootstrap_samples)
    k = kurtosis(bootstrap_samples, fisher=False)
    return jb(n, s, k)
```

4.15 The distribution of the test-statistic under the null hypothesis

```
bootstrap_replications = 10000
n = 10  # Sample size
test_statistic_null = jb_from_samples(n, np.random.normal(size=(n,
    ↪ bootstrap_replications)))
plt.hist(test_statistic_null, bins=100)
plt.title('JB test-statistic under null hypothesis from bootstrap (
    ↪ $n=10$)'); plt.xlabel('$JB$')
plt.show()
```



4.16 The critical value

- The 95-percentile can be computed from the bootstrap data.
- This is called the *critical value* for $p = 0.05$.

```
critical_value = np.percentile(test_statistic_null, 95)
critical_value
```

```
2.5370999432580295
```

- This is the value of JB_{crit} such that area underneath the p.d.f. over the interval $[0, JB_{crit}]$ sums to 0.95 (95% of the area under the curve).
- The corresponding p-value is $1 - 0.95 = 0.05$.

4.17 Rejecting the null hypothesis

- When we test an empirical sample, we compute its sample skewness and kurtosis, and the corresponding value of the test statistic JB_{data}.
- We reject the null hypothesis iff. $JB_{data} > JB_{crit}$:

```
def jb_critical_value(n, bootstrap_samples, p):
    return np.percentile(jb_from_samples(n, bootstrap_samples), (1.
↪   - p) * 100.)

def jb_test(data_sample, bootstrap_replications=100000, p=0.05):
    sample_size = len(data_sample)
    bootstrap_samples = np.random.normal(size=(sample_size,
```

```
        ↪ bootstrap_replications))
    critical_value = jb_critical_value(sample_size,
        ↪ bootstrap_samples, p)
    empirical_jb = jb(sample_size, skew(data_sample), kurtosis(
        ↪ data_sample, fisher=False))
    return (empirical_jb > critical_value, empirical_jb,
        ↪ critical_value)
```

4.17.1 Test data from a normal distribution

```
x = np.random.normal(size=2000)
jb_test(x)
```

```
(False, 0.3436442928512375, 5.958443087793155)
```

4.17.2 Test data from a log-normal distribution

```
jb_test(np.exp(x))
```

```
(True, 589546.3684834961, 5.968889385720822)
```

4.18 Critical-values from a Chi-Squared table

- The code on the previous slide is not very efficient, since we have to perform a lengthly bootstrap operation each time we test a data sample.
- For $n > 2000$, the distribution of the test statistic follows a Chi-squared distribution with two degrees of freedom ($k = 2$), so we can look up the critical values for any given confidence level (p-value) using a Chi-Squared table.
- For smaller n we must resort to a bootstrap.

4.19 Producing a table of table critical values from a bootstrap

```
n = 10
bootstrap_samples = np.random.normal(size=(n, 300000))
confidence_levels = np.array([0.025, 0.05, 0.10, 0.20])
critical_values = np.vectorize(lambda p: jb_critical_value(n,
    ↪ bootstrap_samples, p))(confidence_levels)
critical_values_df = pd.DataFrame({'critical value (n=10)':
    ↪ critical_values}, index=confidence_levels)
critical_values_df.index.name = 'p-value'
critical_values_df
```

```
          critical value (n=10)
p-value
0.025                  3.728356
0.050                  2.518205
0.100                  1.620483
0.200                  1.125069
```

- If we save this data-frame permanently, then we do not need to re-compute the critical value for the given sample size.
- We can simply calculate the test-statistic from the data sample, and see whether the value thus obtained exceeds the critical value for the chosen level of confidence (p-value).

4.20 Using the `jarque_bera` function in scipy

- The function `scipy.stats.jarque_bera()` contains code already written to implement the Jarque-Bera (JB) test.
- It computes the p-value from the cdf. of the Chi-Squared distribution and the empirical test-statistic.
- This assumes a large sample $n \geq 2000$.
- The variable `test_statistic` returned below is the value of JB calculated from the *empirical* data sample.
- If the p-value in the result is ≤ 0.05 then we reject the null hypothesis at 95% confidence.
- The null hypothesis is that the data are drawn from a distribution with skew 0 and kurtosis 3.

```python
import scipy.stats
x = np.random.normal(size=2000)
(test_statistic, p_value) = scipy.stats.jarque_bera(x)
print("JB test statistic = %f" % test_statistic)
print("p-value = %f" % p_value)
```

```
JB test statistic = 0.096519
p-value = 0.952887
```

4.21 Testing the empirical data

```python
len(stock_returns)
```

```
208
```

```python
stats.jarque_bera(stock_returns)
```

```
(6.19438331072041, 0.04517589389305776)
```

4.22 The single-index model

$$r_{i,t} - r_f = \alpha_i + \beta_i(r_{m,t} - r_f) + \epsilon_{i,t} \tag{4.2}$$

$$\epsilon_{i,t} \sim N(0, \sigma_i) \tag{4.3}$$

- $r_{i,t}$ is return to stock i in period t.
- r_f is the risk-free rate.
- $r_{m,t}$ is the return to the market portfolio.

Elton, E. J., & Gruber, M. J. (1997). *Modern portfolio theory, 1950 to date*. Journal of Banking and Finance, 21(11–12), 1743–1759. https://doi.org/10.1016/S0378-4266(97)00048-4

4.23 Estimating the single-index model

- We will first obtain data on the market index: in this case the NASDAQ:

```
nasdaq_index = prices_from_csv('data/^NDX.csv')
nasdaq_index.head()
```

```
                  Date        Open        High         Low
Close  \
Date
2002-07-01  2002-07-01  1044.479980  1049.880005   997.969971
998.169983
2002-07-02  2002-07-02   989.250000   993.989990   961.760010
963.659973
2002-07-03  2002-07-03   957.260010   995.950012   950.330017
995.679993
2002-07-05  2002-07-05  1018.630005  1061.050049  1018.630005
1060.890015
2002-07-08  2002-07-08  1051.270020  1066.280029  1008.780029
1014.330017

              Adj Close      Volume
Date
2002-07-01   998.169983  2320650000
2002-07-02   963.659973  2722550000
2002-07-03   995.679993  2661060000
2002-07-05  1060.890015  1120960000
2002-07-08  1014.330017  1708150000
```

4.24 Converting to monthly data

- As before, we can resample to obtain monthly data.

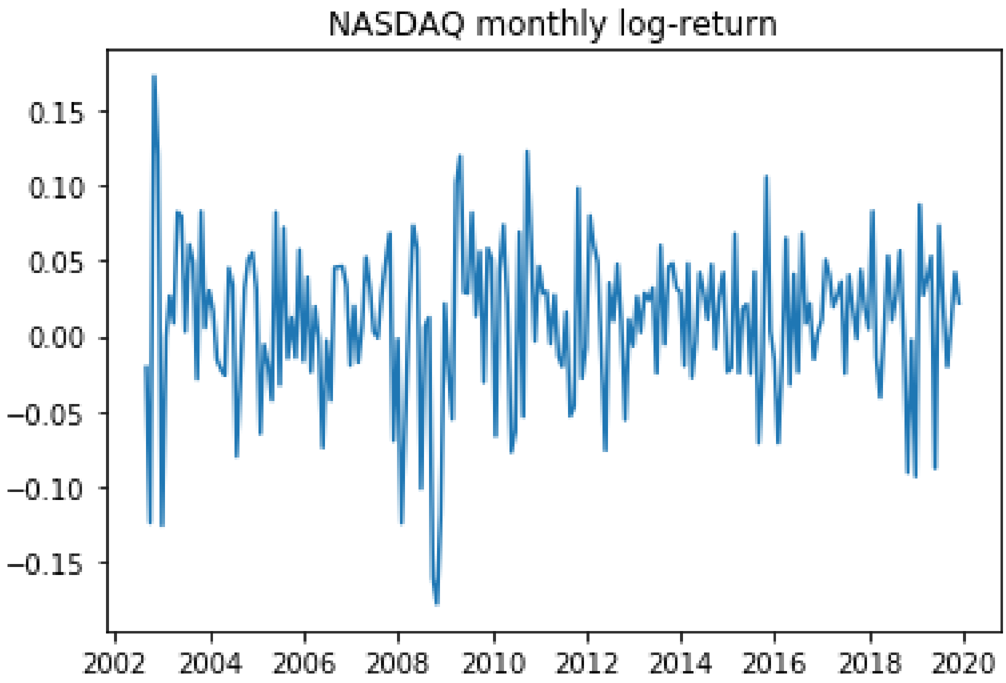
```
nasdaq_monthly_prices = nasdaq_index['Adj Close'].resample('M').
    ↪ last()
nasdaq_monthly_prices.head()
```

```
Date
2002-07-31      962.099976
2002-08-31      942.380005
2002-09-30      832.520020
2002-10-31      989.539978
2002-11-30     1116.099976
Freq: M, Name: Adj Close, dtype: float64
```

4.25 Plotting monthly returns

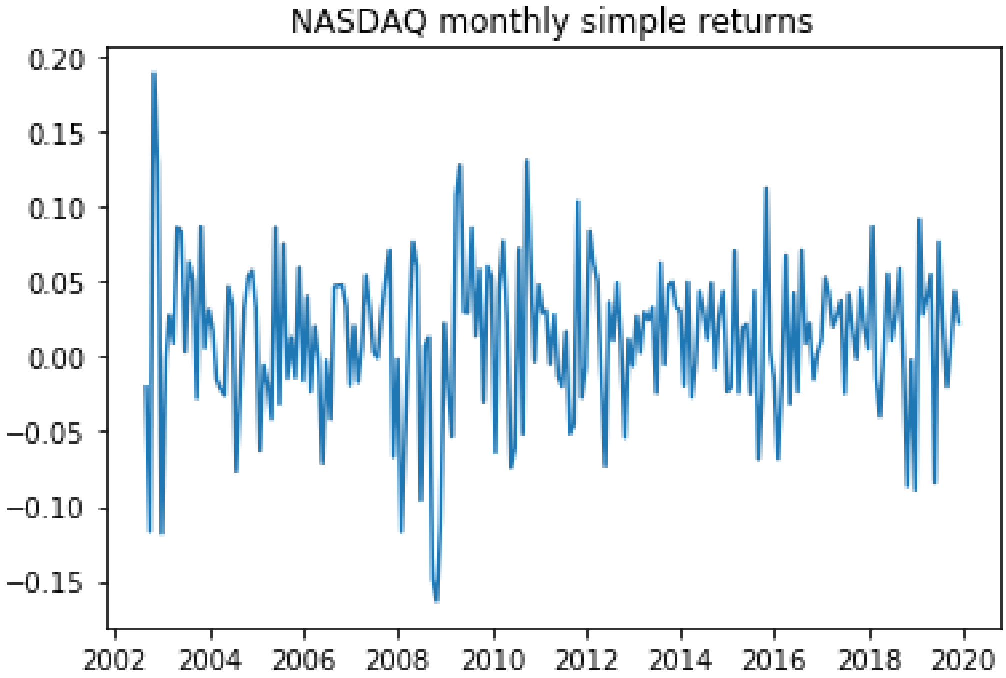
```
index_log_returns = np.diff(np.log(nasdaq_monthly_prices))
index_log_returns_df = pd.DataFrame({'NASDAQ monthly returns':
    ↪ index_log_returns}, index=nasdaq_monthly_prices.index[1:])
plt.plot(index_log_returns_df)
```

```python
plt.title('NASDAQ monthly log-return')
plt.show()
```



4.26 Converting to simple returns

```python
index_simple_returns_df = np.exp(index_log_returns_df) - 1.
plt.plot(index_simple_returns_df)
plt.title('NASDAQ monthly simple returns')
plt.show()
```




```
stock_simple_returns_df = np.exp(stock_returns_df) - 1.
```

4.27 Concatenating data into a single data frame

- We will now concatenate the data into a single data fame.
- We can use `pd.concat()`, specifying an axis of 1 to merge data along columns.
- This is analogous to performing a `zip()` operation.

```
comparison_df = pd.concat([index_simple_returns_df,
    ↪ stock_simple_returns_df], axis=1)
comparison_df.head()
```

```
                NASDAQ monthly returns  MSFT monthly returns
Date
2002-08-31                   -0.020497              0.022926
2002-09-30                   -0.116577             -0.108802
2002-10-31                    0.188608              0.222450
2002-11-30                    0.127898              0.078736
2002-12-31                   -0.118027             -0.103676
```

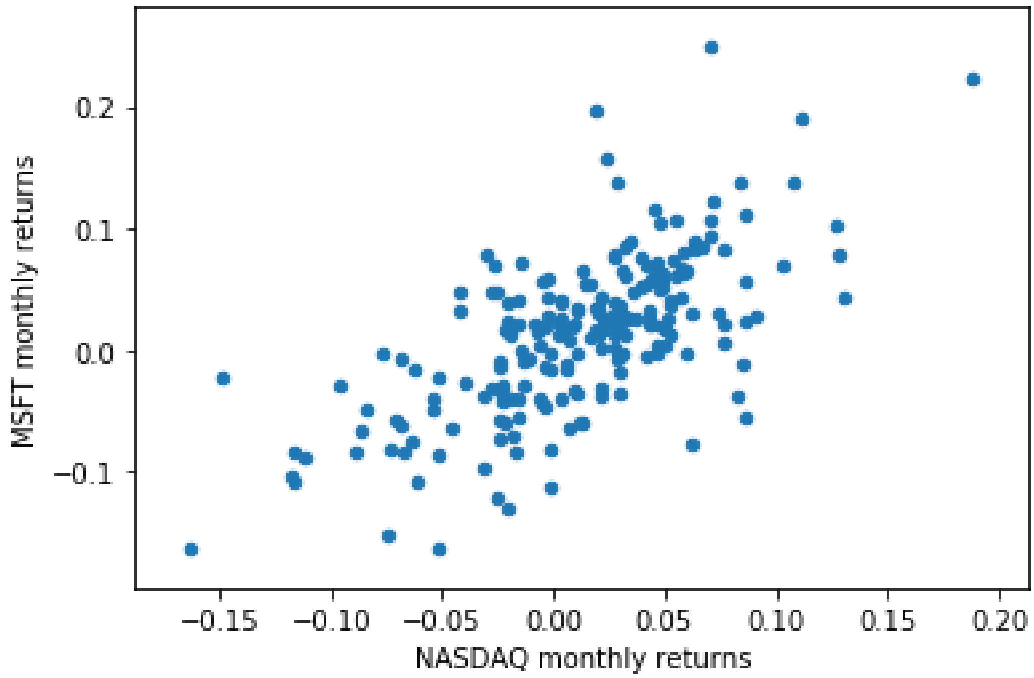
4.28 Scatter plots

- We can produce a scatter plot to see whether there is any relationship between the stock returns, and the index returns.
- There are two ways to do this:
  1. Use the function `scatter()` in `matplotlib.pyplot`
  2. Invoke the `plot()` method on a data frame, passing `kind='scatter'`

4.29 Scatter plots using the `plot()` method of a data frame

- In the example below, the `x` and `y` named arguments refer to column numbers of the data frame.
- Notice that the `plot()` method is able to infer the labels automatically.

```
comparison_df.plot(x=0, y=1, kind='scatter')
plt.show()
```



4.30 Computing the correlation matrix

- For random variables X and Y, the Pearson correlation coefficient is:

$$\rho_{X,Y} = \frac{\mathrm{cov}(X,Y)}{\sigma_X \sigma_Y} \tag{4.4}$$

$$= \frac{E[(X-\mu_x)(Y-\mu_Y)]}{\sigma_X \sigma_Y} \tag{4.5}$$

(4.6)

4.31 Covariance and correlation of a data frame

- We can invoke the `cov()` and `corr()` methods on a data frame.

```
comparison_df.cov()
```

```
                      NASDAQ monthly returns  MSFT monthly returns
NASDAQ monthly returns              0.002675              0.002243
```

```
MSFT monthly returns                          0.002243              0.004283
```

```
comparison_df.corr()
```

```
                        NASDAQ monthly returns  MSFT monthly returns
NASDAQ monthly returns                1.000000              0.662686
MSFT monthly returns                  0.662686              1.000000
```

4.32 Comparing multiple attributes in a data frame

- It is often useful to work with more than two variables.
- We can add columns (attributes) to our data frame.
- Many of the methods we are using will automatically incorporate the additional variables into the analysis.

4.33 Using a function to compute returns

- The code below defines a function defines a function which will return a data frame containing a single series of returns for the specified symbol, and sampled over the specified frequency.

```
def returns_df(symbol, frequency='M'):
    df = prices_from_csv('~/Downloads/%s.csv' % symbol)
    prices = df['Adj Close'].resample(frequency).last()
    column_name = symbol + ' returns (' + frequency + ')'
    return pd.DataFrame({column_name: np.exp(np.diff(np.log(prices)
    ↪ )) - 1.},
                                    index=prices.index[1:])
```

```
apple_returns = returns_df('AAPL')
apple_returns.head()
```

```
            AAPL returns (M)
Date
2002-08-31         -0.033421
2002-09-30         -0.016949
2002-10-31          0.108276
2002-11-30         -0.035470
2002-12-31         -0.075484
```

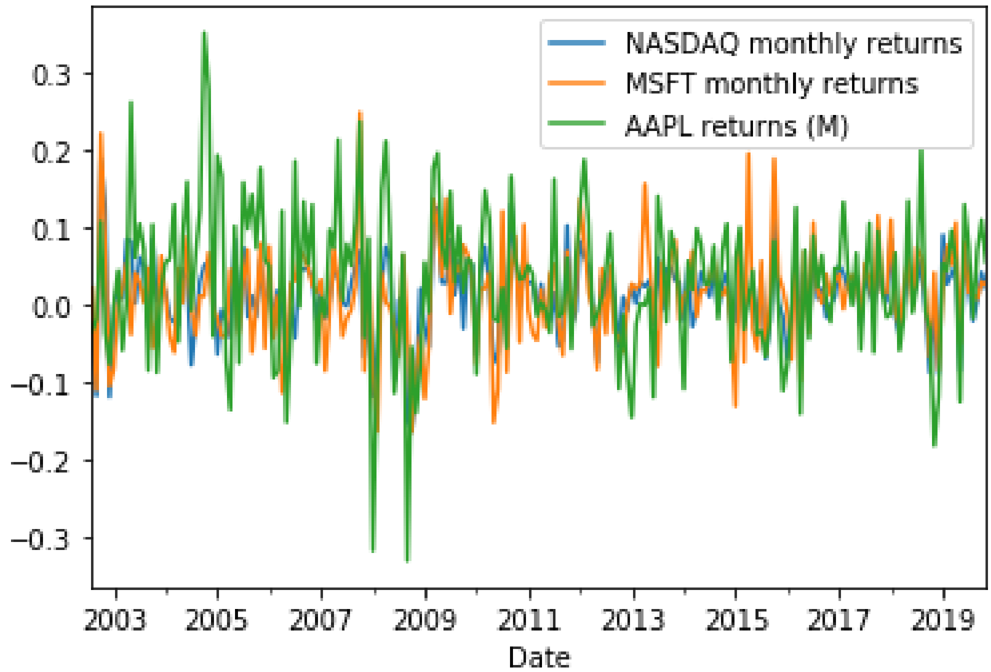
4.34 Adding another stock to the portfolio

```
comparison_df = pd.concat([comparison_df, apple_returns], axis=1)
comparison_df.head()
```

```
            NASDAQ monthly returns  MSFT monthly returns  AAPL returns
 (M)
Date
2002-08-31               -0.020497              0.022926             -
0.033421
2002-09-30               -0.116577             -0.108802             -
```

```
0.016949
2002-10-31              0.188608              0.222450
0.108276
2002-11-30              0.127898              0.078736          -
0.035470
2002-12-31             -0.118027             -0.103676          -
0.075484
```

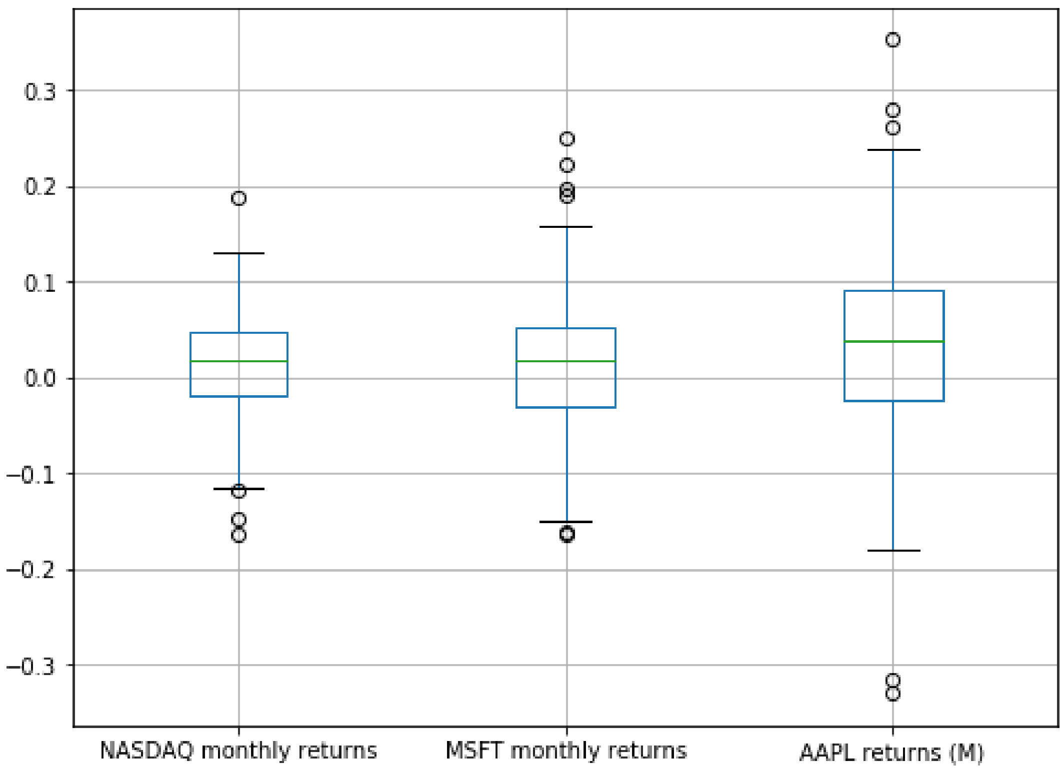
```
comparison_df.plot()
plt.show()
```




```
comparison_df.corr()
```

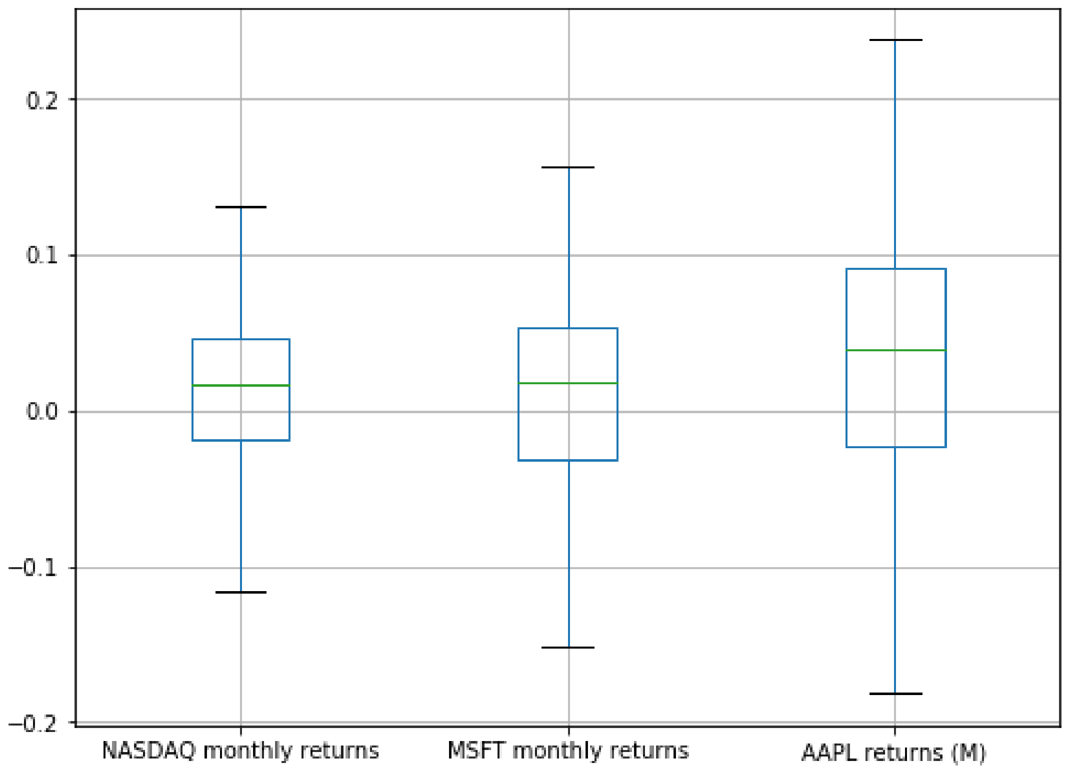
```
                        NASDAQ monthly returns  MSFT monthly returns  \
NASDAQ monthly returns                1.000000              0.662686
MSFT monthly returns                  0.662686              1.000000
AAPL returns (M)                      0.629116              0.375185

                        AAPL returns (M)
NASDAQ monthly returns          0.629116
MSFT monthly returns            0.375185
AAPL returns (M)                1.000000
```

```
plt.figure(figsize=(8, 6))
comparison_df.boxplot()
plt.show()
```



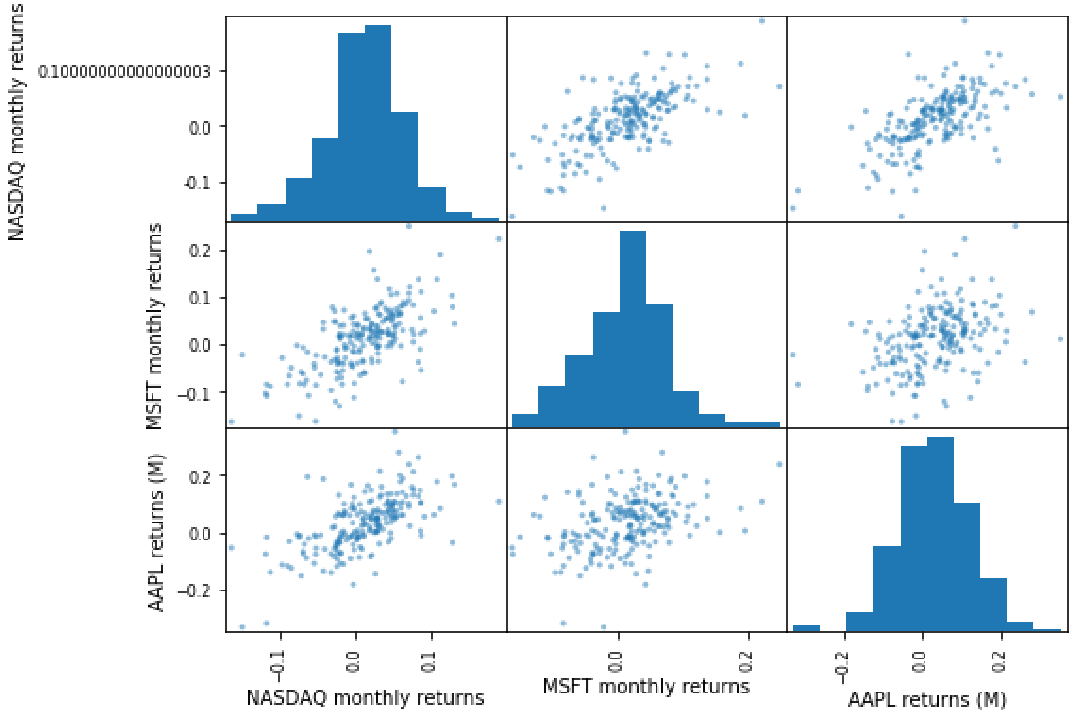
4.35 Boxplots without outliers

```
plt.figure(figsize=(8, 6))
comparison_df.boxplot(showfliers=False)
plt.show()
```



4.36 Scatter matrices

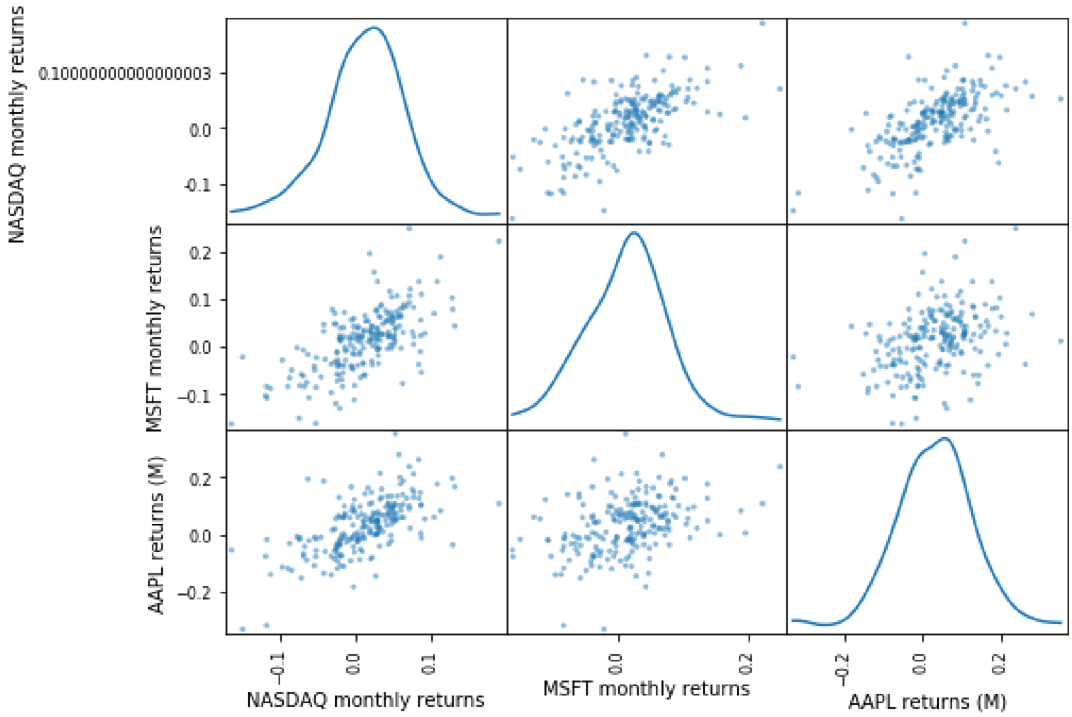
```
pd.plotting.scatter_matrix(comparison_df, figsize=(8, 6))
plt.show()
```



4.37 Scatter matrices with Kernel-density plots

- We can use Kernel density estimation (KDE) to plot an approximation of the pdf.

```
pd.plotting.scatter_matrix(comparison_df, diagonal='kde', figsize
    ↪ =(8, 6))
plt.show()
```



4.38 Ordinary-least squares

- For n observations $(x_{1,j}, y_1), (x_{2,j}, y_2), \ldots, (x_{n,j}, y_n)$ over $j \in \{1, 2, \ldots p\}$ regressors:

$$y_i = \alpha_i + \beta_1 x_{i,1} + \beta_2 x_{i,2} + x_{i,2} + \ldots + \beta_p x_{i,p} + \epsilon_i \tag{4.7}$$

4.39 Ordinary-least squares estimation in Python

- First we import the stats module:

```
import scipy.stats as stats
```

- Now we prepare the data set:

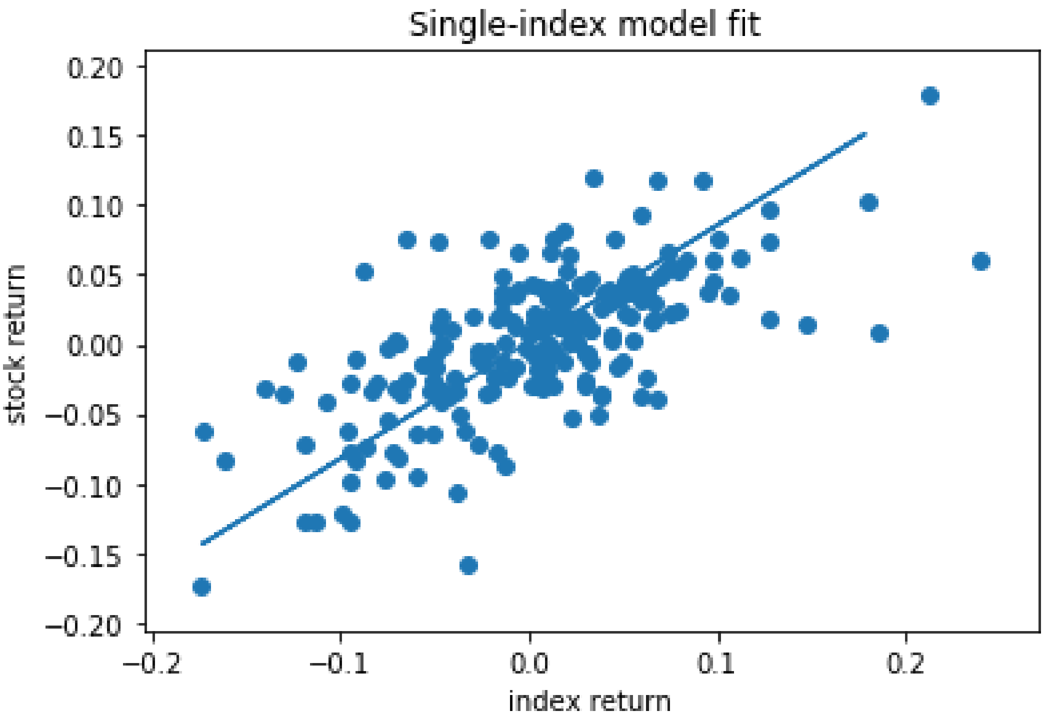
```
rr = 0.01 # risk-free rate
xdata = stock_simple_returns_df.values[:, 0] - rr
ydata = index_simple_returns_df.values[:, 0] - rr
```

```
regression_result = (beta, alpha, rvalue, pvalue, stderr) = \
    stats.linregress(ydata, xdata)
print(regression_result)
```

```
LinregressResult(slope=0.8385169376111149, intercept
=0.0015326890415947839, rvalue=0.6626859398568364, pvalue
=1.1260177711868404e-27, stderr=0.06602262800537023)
```

4.40 Plotting the fitted model

```
plt.scatter(x=xdata, y=ydata)
plt.plot(ydata, alpha + beta * ydata)
plt.xlabel('index return')
plt.ylabel('stock return')
plt.title('Single-index model fit ')
plt.show()
```



4.41 Regressing attributes of a data frame

- First we will create a new data frame containing the excess returns.

```
excess_returns_df = comparison_df - rr
excess_returns_df.head()
```

```
             NASDAQ monthly returns  MSFT monthly returns  AAPL returns
 (M)
Date
2002-08-31              -0.030497              0.012926            -
0.043421
2002-09-30              -0.126577             -0.118802            -
0.026949
2002-10-31               0.178608              0.212450
0.098276
2002-11-30               0.117898              0.068736            -
0.045470
```

```
2002-12-31                    -0.128027                -0.113676           -
0.085484
```

4.42 Renaming the columns of a data frame

- We will now rename the columns to make the variable names easier to work with.

```
excess_returns_df.rename(columns={'NASDAQ monthly returns': 'index'
    ↪ ,
                                   'MSFT monthly returns': 'msft',
                                    'AAPL returns (M)': 'aapl'},
                          inplace=True)
excess_returns_df.head()
```

```
                 index       msft       aapl
Date
2002-08-31 -0.030497  0.012926 -0.043421
2002-09-30 -0.126577 -0.118802 -0.026949
2002-10-31  0.178608  0.212450  0.098276
2002-11-30  0.117898  0.068736 -0.045470
2002-12-31 -0.128027 -0.113676 -0.085484
```

4.42.1 Fitting the model

```
import statsmodels.formula.api as sm
result = sm.ols(formula = 'msft ~ index', data=excess_returns_df).
    ↪ fit()
```

4.42.2 The full regression results

```
print(result.summary())
```

```
                            OLS Regression Results
==============================================================================

Dep. Variable:                   msft   R-squared:
   0.439
Model:                            OLS   Adj. R-squared:
   0.436
Method:                 Least Squares   F-statistic:
   161.3
Date:                Thu, 28 Nov 2019   Prob (F-statistic):
1.13e-27
Time:                        11:52:55   Log-Likelihood:
  332.64
No. Observations:                 208   AIC:
  -661.3
Df Residuals:                     206   BIC:
  -654.6
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
```

```
                  coef    std err          t      P>|t|      [0.025
  0.975]------------------------------------------------------------------------------
---

Intercept       0.0015      0.003      0.450      0.653      -0.005
   0.008
index           0.8385      0.066     12.700      0.000       0.708
   0.969
==============================================================================

Omnibus:                       11.619   Durbin-Watson:
   2.369
Prob(Omnibus):                  0.003   Jarque-Bera (JB):
  21.780
Skew:                           0.241   Prob(JB):
1.86e-05
Kurtosis:                       4.510   Cond. No.
    19.4
==============================================================================


Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is
 correctly specified.
```

4.42.3 The intercept and coefficient

```python
print(result.params)
```

```
Intercept    0.001533
index        0.838517
dtype: float64
```

```python
coefficient = result.params['index']
coefficient
```

```
0.8385169376111143
```

4.43 Portfolio optimization

- For a column vector $\mathbf{w}$ of portfolio weights, the portfolio return r_p is given by:

$$r_p = \sum_{i=1}^{n} w_i r_i \tag{4.8}$$

- The portfolio variance σ_p is given by:

$$\sigma_p = \sum_{i=1}^{n} \sum_{j=1}^{n} w_i w_j k_{i,j} \sigma_i \sigma_j = \mathbf{w}^T \cdot \mathbf{K} \cdot \mathbf{w} \tag{4.9}$$

where $\mathbf{K}$ is the covariance matrix.

4.43.1 Portfolio mean and variance in Python

- We can write the equation from the previous slide as a Python function:

```
def portfolio_mean_var(w, R, K):
    portfolio_mean = np.mean(R, axis=0) * w
    portfolio_var = w.T * K * w
    return portfolio_mean.item(), portfolio_var.item()
```

- The `item()` method is required to convert a one-dimensional matrix into a scalar.

4.43.2 Obtaining portfolio data in Pandas

```
portfolio = pd.concat([returns_df(s) for s in ['AAPL', 'ATVI', '
    ↪ MSFT', 'VRSN', 'WDC']], axis=1)
portfolio.head()
```

```
              AAPL returns (M)  ATVI returns (M)  MSFT returns (M)  \
Date
2002-08-31           -0.033421         -0.029596          0.022926
2002-09-30           -0.016949         -0.141371         -0.108802
2002-10-31            0.108276         -0.143335          0.222450
2002-11-30           -0.035470          0.053658          0.078736
2002-12-31           -0.075484         -0.324537         -0.103676

              VRSN returns (M)  WDC returns (M)
Date
2002-08-31            0.121875        -0.087838
2002-09-30           -0.296657         0.160493
2002-10-31            0.594059         0.317022
2002-11-30            0.305590         0.365105
2002-12-31           -0.236917        -0.243787
```

4.43.3 Computing the covariance matrix

```
portfolio.cov()
```

```
                  AAPL returns (M)  ATVI returns (M)  MSFT returns (M)
  \
AAPL returns (M)          0.009047          0.003175          0.002335
ATVI returns (M)          0.003175          0.009093          0.001402
MSFT returns (M)          0.002335          0.001402          0.004283
VRSN returns (M)          0.002891          0.001898          0.002008
WDC returns (M)           0.004194          0.002607          0.002413

                  VRSN returns (M)  WDC returns (M)
AAPL returns (M)          0.002891         0.004194
ATVI returns (M)          0.001898         0.002607
MSFT returns (M)          0.002008         0.002413
VRSN returns (M)          0.011007         0.005368
WDC returns (M)           0.005368         0.016106
```

4.43.4 Converting to matrices

```
R = np.matrix(portfolio)
K = np.matrix(portfolio.cov())
```

```
K
```

```
matrix([[0.00904665, 0.00317498, 0.00233534, 0.00289089, 0.00419387],
        [0.00317498, 0.00909339, 0.00140182, 0.00189845, 0.00260658],
        [0.00233534, 0.00140182, 0.00428272, 0.00200831, 0.00241259],
        [0.00289089, 0.00189845, 0.00200831, 0.0110073 , 0.00536781],
        [0.00419387, 0.00260658, 0.00241259, 0.00536781, 0.01610639]])
```

4.43.5 An example portfolio

- Let's construct a single portfolio by specifying a weight vector:

```
w = np.matrix('0.4; 0.2; 0.2; 0.1; 0.1')
w
```

```
matrix([[0.4],
        [0.2],
        [0.2],
        [0.1],
        [0.1]])
```

```
np.sum(w)
```

```
1.0
```

```
portfolio_mean_var(w, R, K)
```

```
(0.023266799902568663, 0.004278614805731206)
```

4.43.6 Optimizing portfolios

- We can use the `scipy.optimize` module to solve the portfolio optimization problem.
- First we import the module:

```
import scipy.optimize as sco
```

4.43.6.1 Defining an objective function

- Next we define an objective function.
- This function will be minimized.
- In this example, we linearly weight each of our optimization objectives, mean and variance, using a risk-aversion parameter.

```
def portfolio_performance(w_list, R, K, risk_aversion):
    w = np.matrix(w_list).T
    mean, var = portfolio_mean_var(w, R, K)
    return risk_aversion * var -  (1 -risk_aversion) * mean
```

4.43.6.2 Computing the performance of a given portfolio

```
def uniform_weights(n):
    return [1. / float(n) for i in range(n)]
```

```
uniform_weights(5)
```

```
[0.2, 0.2, 0.2, 0.2, 0.2]
```

```
portfolio_performance(uniform_weights(5), R, K, risk_aversion=0.5)
```

```
-
0.008482461529679837
```

4.43.6.3 Finding optimal portfolio weights

```
def optimal_portfolio(R, K, risk_aversion):
    n = R.shape[1]
    constraints = ({'type': 'eq', 'fun': lambda x: np.sum(x) - 1})
    bounds = tuple( (0,1) for asset in range(len(w)))
    result = sco.minimize(portfolio_performance, uniform_weights(n)
    ↪ , args=(R, K, risk_aversion),
                         method='SLSQP', bounds=bounds, constraints=
    ↪ constraints)
    return np.matrix(result.x).T
```

```
optimal_portfolio(R, K, risk_aversion=0.5)
```

```
matrix([[9.04156453e-01],
        [7.28583860e-17],
        [0.00000000e+00],
        [9.58435472e-02],
        [0.00000000e+00]])
```

4.43.7 Computing the Pareto frontier

- First we define our risk aversion coefficients

```
risk_aversion_coefficients = np.arange(0.0, 1.1, 0.1)
risk_aversion_coefficients
```

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

4.43.7.1 The optimal portfolios

```python
optimal_portfolios = [optimal_portfolio(R, K, ra) for ra in
    ↪ risk_aversion_coefficients]
```

The least risk-averse portfolio:

```python
optimal_portfolios[0]
```

```
matrix([[1.0000e+00],
        [2.9577e-16],
        [0.0000e+00],
        [9.2981e-16],
        [3.7730e-16]])
```

The most risk-averse portfolio:

```python
optimal_portfolios[-1]
```

```
matrix([[0.0933],
        [0.2077],
        [0.5684],
        [0.1134],
        [0.0173]])
```

4.43.7.2 The efficient frontier

- Now we can map from the optimal portfolio weights for each level of risk-aversion onto mean-variance space:

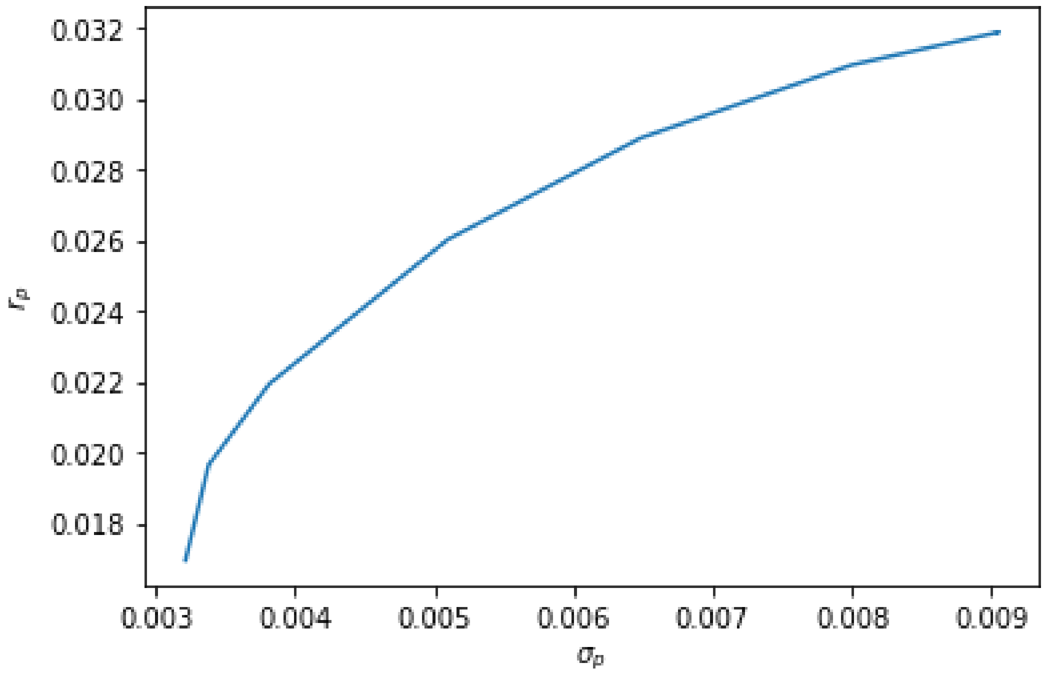
```python
pareto_frontier = np.matrix([portfolio_mean_var(w, R, K) for w in
    ↪ optimal_portfolios])
```

```python
pareto_frontier
```

```
matrix([[0.0319, 0.009 ],
        [0.0319, 0.009 ],
        [0.0319, 0.009 ],
        [0.0319, 0.009 ],
        [0.0319, 0.009 ],
        [0.031 , 0.008 ],
        [0.0289, 0.0065],
        [0.026 , 0.0051],
        [0.022 , 0.0038],
        [0.0197, 0.0034],
        [0.017 , 0.0032]])
```

4.43.8 Plotting the Pareto frontier

```python
plt.plot(pareto_frontier[:, 1], pareto_frontier[:, 0])
plt.xlabel('$\sigma_p$');  plt.ylabel('$r_p$')
plt.show()
```


0.032
0.030
0.028
0.026
0.024
0.022
0.020
0.018
r_p
0.003
0.004
0.005
0.006
0.007
0.008
0.009
σ_p

5 Monte-Carlo Methods

5.1 Quantitative Models

- A mathematical model uses variables to represent quantities in the real-world.
  - e.g. security prices
- Variables are related to each other through mathematical equations.
- Variables can be divided into:
  - Input variables: parameters (independent variables)
    - Inititial conditions: parameters which specify the initial values of in a time-varying (dynamic) model.
  - Output variables: dependent-variables (e.g. payoff of an option).

5.2 Monte-Carlo Methods

- Financial models are typically *stochastic*.
- Stochastic models make use of random variables.
- If the dependent variables are stochastic, we typically want to compute their *expectation*.
  - Note, however, that in some models, dependent variables are deterministic, even when parameters are random.
- If the parameters are stochastic, we can use Monte-Carlo methods to *estimate* the expected values of dependent variables.

5.3 The Monte-Carlo Casino

- "Monte-Carlo" was the secret code-name of a project which used the earliest Monte-Carlo methods to solve problems of neurton-diffusion during the development of the first-atomic bomb.
- It was named after the Monte-Carlo casino.

5.4 Pseudo-code

- We will illustrate a simple Monte-Carlo method for analysing a stochastic model.
- We will make use of *pseudo-code*.
- Pseudo-code is written for people.
- It is not executable by machines.
- It is written to *illustrate* exactly how something is done.
- Exact specifications of the steps required to compute mathematical values are called *algorithms*.
- Pseudo-code can be used to write down algorithms.

5.5 A simple Monte-Carlo method

- Here we consider a simple model with one input variable X, and one output variable Y, related by a function $Y = f(X)$.
- X and Y are random variables.
- X is iid. distributed with some known distribution.
- We want to compute the expected value of the dependent variable $E(Y)$.
- We do so by drawing a random sample of n random variates $(x_1, x_2, \ldots, x_n)$ from the specified distribution.
- We map these values onto a sample $\mathbf{y}$ of the dependent variable Y: $\mathbf{y} = (f(x_1), f(x_2), \ldots, f(x_n))$.
- We can use the sample mean $\bar{\mathbf{y}} = \sum_i f(x_i)/n$ to estimate $E(Y)$.
- Provided that n is sufficiently large, our estimate will be accurate by the law of large numbers.
- $\bar{\mathbf{y}}$ is called *the Monte-Carlo estimator*.

5.6 In Pseudo-code

- The pseudo-code below illustrates the method specified on the previous slide using iteration:

```
sample = []
for i in range(n):
    x = draw_random_value(distribution)
    y = f(input_variable)
    sample.append(y)
result = mean(sample)
```

- We can write this more concisely using a comprehension:

```
inputs = draw_random_value(distribution, size=n)
result = mean([f(x) for x in inputs])
```

5.7 A Monte-Carlo algorithm for computing π

1. Inscribe a circle in a square.
2. Randomly generate points (X, Y) in the square.
3. Determine the number of points in the square that are also in the circle.
4. Let R be the number of points in the circle divided by the number of points in the square, then $\pi = 4 \times E(R)$.

See this tutorial.

```
import numpy as np

def f(x, y):
        if x*x + y*y < 1:
            return 1.
        else:
            return 0.

n = 1000000
X = np.random.random(size=n)
```

```python
Y = np.random.random(size=n)
pi_approx = 4 * np.mean([f(x, y) for (x, y) in zip(X,Y)])
print("Pi is approximately %f" % pi_approx)
```

```
Pi is approximately 3.144072
```

5.8 Monte-Carlo Integration

The expectation of a random variable $X \in \mathbb{R}$ with pdf. $f(x)$ can be written:

$$E[X] = \int_{-\infty}^{+\infty} x f(x)\, dx \tag{5.1}$$

For a continuous uniform distribution over $U(0,1)$, the pdf. is $f(x) = 1$, and:

$$E[X] = \int_0^1 x\, dx \tag{5.2}$$

5.9 Estimating π using Monte-Carlo integration

Consider:

$$E[\sqrt{1-X^2}] = \int_0^1 \sqrt{1-x^2}\, dx \tag{5.3}$$

If we draw a finite random sample $x_1, x_2, \ldots, x_n$ from $U(0,1)$, then

$$\bar{\mathbf{x}} \approx E[X] \quad = \quad \int_0^1 \sqrt{1-x^2} dx \tag{5.4}$$

$$\int \sqrt{1-x^2} dx \quad = \quad \frac{1}{2}(x\sqrt{1-x^2} + \arcsin(x)). \tag{5.5}$$

$$\tag{5.6}$$

Therefore:

$$\bar{\mathbf{x}} \approx E[X] = \frac{\pi}{4} \tag{5.7}$$

5.10 Estimation error

- By the law of large numbers $\lim_{n\to\infty} \bar{\mathbf{x}} = E(X)$.
- However, for finite values of n we will have an estimation error.
- Can we quantify the estimation error as a function of n?

5.11 Computing the error numerically

- If we draw from a standard normal distribution, we know that $E(X) = 0$.
- Therefore we can easily compute the estimation error in any given sample.

5.12 The error for a small random sample.

- Here $X \sim N(0,1)$, and we draw a random sample $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ of size $n = 5$.
- We will compute $\epsilon_{\mathbf{x}} = |\bar{\mathbf{x}} - E(X)| = |\bar{\mathbf{x}}|$

```
x = np.random.normal(size=5)
x
```

```
array([ 0.1388361 ,  0.38725229,  0.32960095,  0.75778728, -
0.20427589])
```

```
np.mean(x)
```

```
0.28184014536845586
```

```
estimation_error = np.sqrt(np.mean(x)**2)
estimation_error
```

```
0.28184014536845586
```

- If we draw a different sample, will the error be different or the same?

```
x = np.random.normal(size=5)
estimation_error = np.sqrt(np.mean(x)**2)
estimation_error
```

```
0.4203236264247142
```

```
x = np.random.normal(size=5)
estimation_error = np.sqrt(np.mean(x)**2)
estimation_error
```

```
0.472181171295761
```

```
x = np.random.normal(size=5)
estimation_error = np.sqrt(np.mean(x)**2)
estimation_error
```

```
0.6013672431458685
```

- The error $\epsilon_{\mathbf{x}}$ is itself a random variable.
- How can we compute $E(\epsilon_{\mathbf{x}})$?

5.13 Monte-Carlo estimation of the sampling error

```
def sampling_error(n):
    errors = [np.sqrt(np.mean(np.random.normal(size=n))**2) \
                for i in range(100000)]
    return np.mean(errors)

sampling_error(5)
```

```
0.35638525509003804
```

- Notice that this estimate is relatively stable:

```
sampling_error(5)
```

```
0.3568948241598915
```

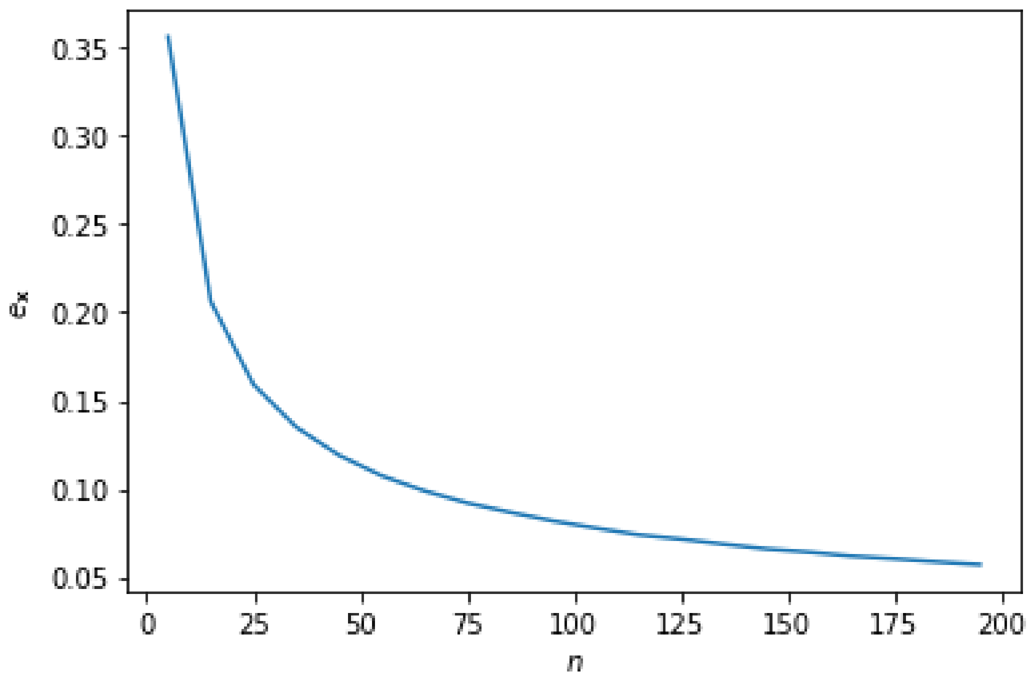
```
sampling_error(5)
```

```
0.35572945022084923
```

5.14 Monte-Carlo estimation of the standard error

- We can now examine the relationship between sample size n and the expected error using a Monte-Carlo method.

```
import matplotlib.pyplot as plt
n = np.arange(5, 200, 10)
plt.plot(n, np.vectorize(sampling_error)(n))
plt.xlabel('$n$');  plt.ylabel('$e_\mathbf{x}$')
plt.show()
```



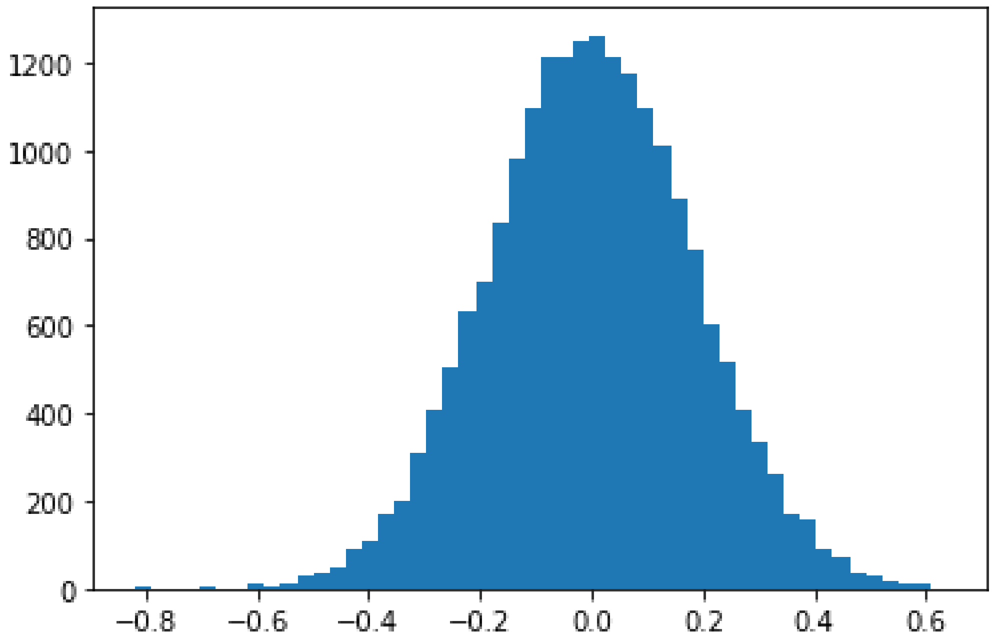
5.15 The sampling distribution of the mean

- The variance in the error occurs because the sample mean is a random variable.
- What is the distribution of the sample mean?

5.16 The sampling distribution of the mean

- Let's fix the sample size at $n = 30$, and look at the empirical distribution of the sample means.

```
# Sample size
n = 30
# Number of repeated samples
N = 20000

means_30 = [np.mean(np.random.normal(size=n)) for i in range(N)]
ax = plt.hist(means_30, bins=50, label='$n=30$')
plt.show()
```

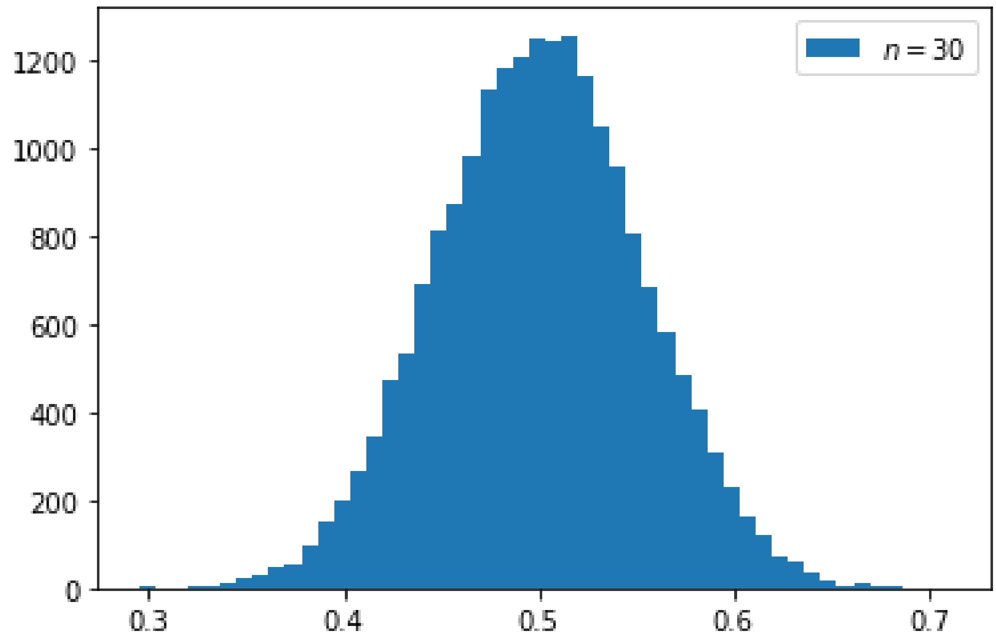


5.17 The sampling distribution of the mean

- Now let's do this again for a variable sampled from a *different* distribution: $X \sim U(0, 1)$.

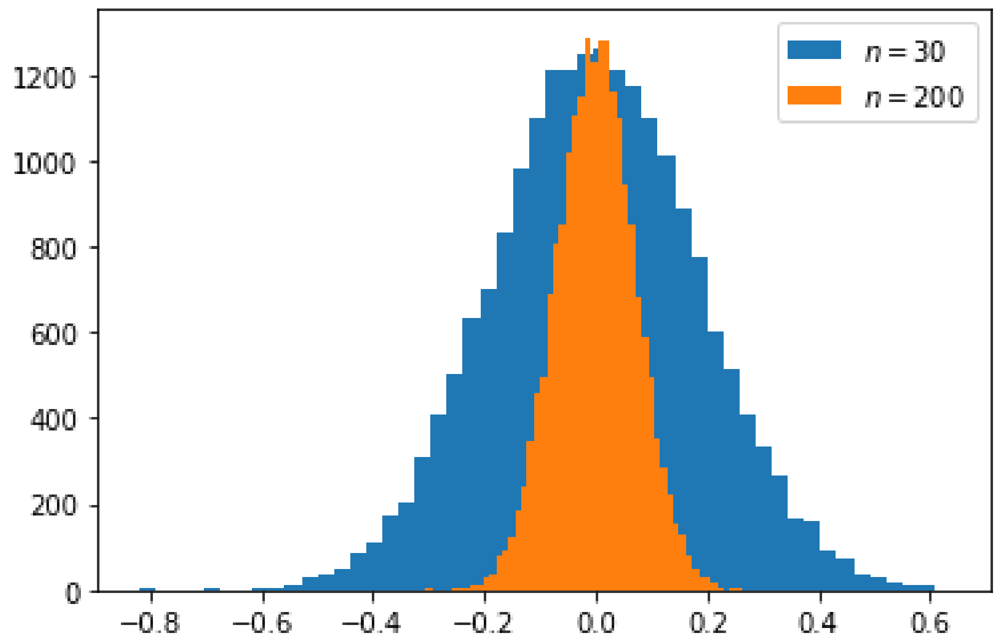
```
# Sample size
n = 30
# Number of repeated samples
N = 20000
means_30_uniform = [np.mean(np.random.uniform(size=n)) for i in
    ↪ range(N)]
```

```
plt.hist(means_30_uniform, bins=50, label='$n=30$')
plt.legend();  plt.show()
```



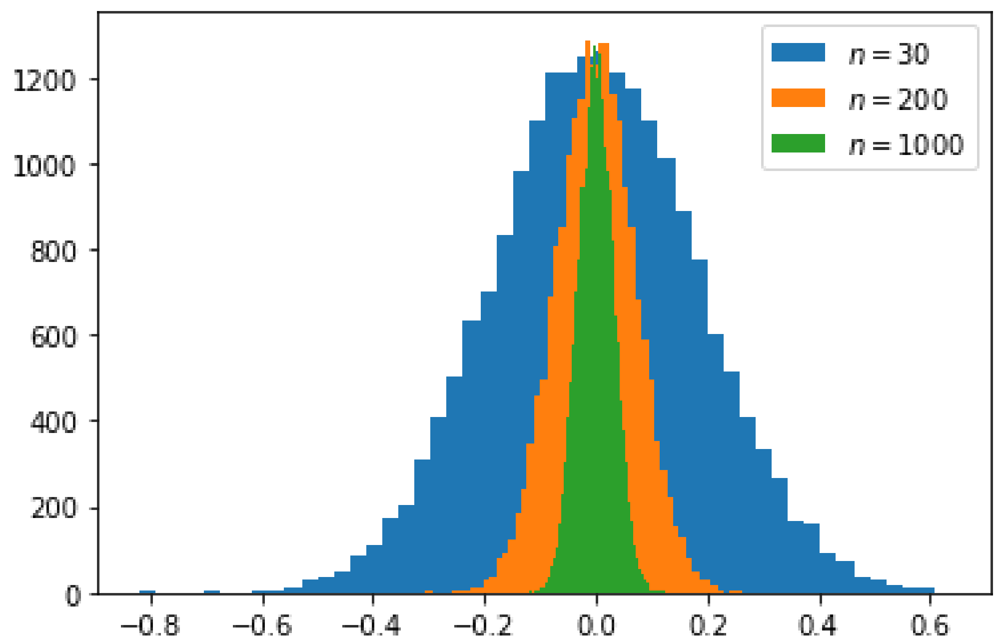
5.18 Increasing the sample size

```
# Sample size
n = 200

means_200 = [np.mean(np.random.normal(size=n)) for i in range(N)]
plt.hist(means_30, bins=50, label='$n=30$')
plt.hist(means_200, bins=50, label='$n=200$')
plt.legend()
plt.show()
```



5.18.1 Increasing the sample size further

```
# Sample size
n = 1000
means_1000 = [np.mean(np.random.normal(size=n)) for i in range(N)]
plt.hist(means_30, bins=50, label='$n=30$')
plt.hist(means_200, bins=50, label='$n=200$')
plt.hist(means_1000, bins=50, label='$n=1000$')
plt.legend(); plt.show()
```



5.19 The sampling distribution of the mean

- In general the sampling distribution of the mean approximates a normal distribution.
- If $X \sim N(\mu, \sigma^2)$ then $\mathbf{\bar{x}_n} \sim N(\mu, \frac{\sigma^2}{n})$.
- The *standard error* of the mean is $\sigma_{\bar{\mathbf{x}}} = \frac{\sigma}{\sqrt{n}}$.
- Therefore sample size must be quadrupled to achieve half the measurement error.

5.20 Summary

- Monte-Carlo methods can be used to analyse quantitative models.
- Any problem in which the solution can be written as an expectation of random variable(s) can be solved using a Monte-Carlo approach.
- We write down an *estimator* for the problem; a variable whose expectation represents the solution.
- We then repeatedly sample input variables, and calculate the estimator numerically (in a computer program).
- The sample mean of this variable can be used as an approximation of the solution; that is, it is an estimate.
- The larger the sample size, the more accurate the estimate.
- There is an inverse-square relationship between sample size and the estimation error.

6 Random walks in Python

(c) 2019 Steve Phelps

6.1 A Simple Random Walk

- Imagine a board-game in which we move a counter either up or down on an infinite grid based on the flip of a coin.
- We start in the center of the grid at position $y_1 = 0$.
- Each turn we flip a coin. If it is heads we move up one square, otherwise we move down.
- How will the counter behave over time? Let's simulate this in Python.
- First we create a variable y to hold the current position

```
y = 0
```

6.2 Movements as Bernoulli trials

- Now we will generate a Bernoulli sequence representing the moves
- Each movement is an i.i.d. discrete random variable ϵ_t distributed with $p(\epsilon_t = 0) = \frac{1}{2}$ and $p(\epsilon_t = 1) = \frac{1}{2}$.
- We will generate a sequence $(\epsilon_1, \epsilon_2, \ldots, \epsilon_{t_{max}})$ such movements, with $t_{max} = 100$.
- The time variable is also discrete, hence this is a *discrete-time* model.
- This means that time values can be represented as integers.

6.2.1 Simulating a Bernoulli process in Python

```
import numpy as np
from numpy.random import randint

max_t = 100
movements = randint(0, 2, size=max_t)
print(movements)
```

```
[0 0 1 0 0 1 1 0 1 1 0 1 0 1 0 1 0 0 0 0 1 0 0 1 0 0 0 1 1 1 0 1 1 1 0
 0 0
 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 0 1 1 0 1 1 1 1 0 1 0 1 0 0 1 0 1
 1 1
 1 1 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0 1 1 0 0 1 1 1 1 1]
```

6.3 An integer random-walk in Python

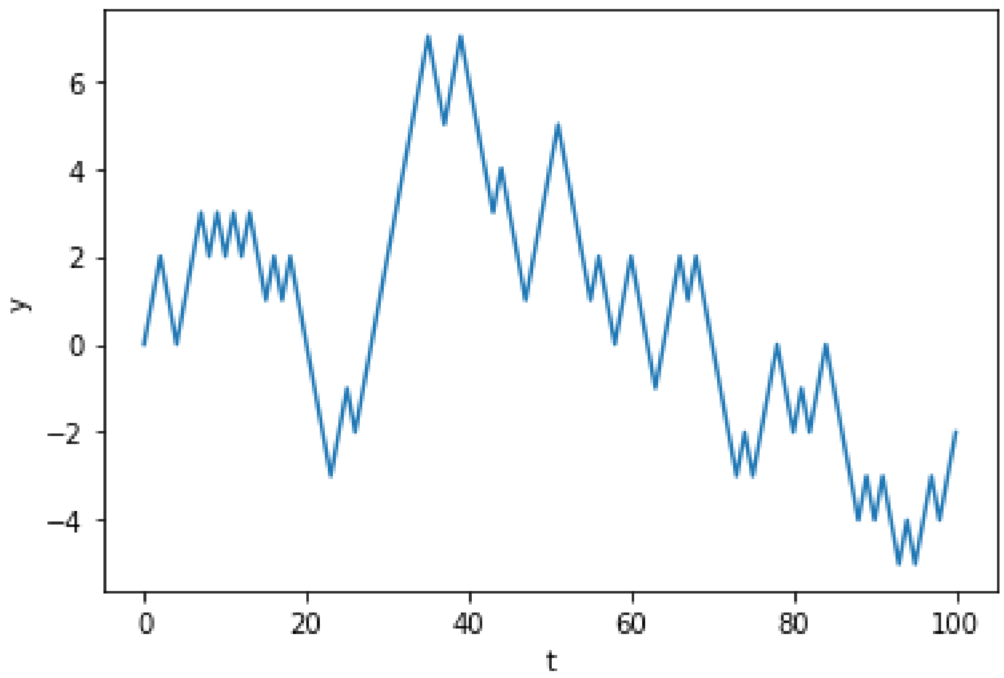
- Each time we move the counter, we move it in the upwards direction if we flip a 1, and downwards for a 0.
- So we add 1 to y_t for a 1, and subtract 1 for a 0.

6.3.1 an integer random-walk using a loop

```python
import numpy as np
import matplotlib.pyplot as plt
from numpy.random import randint, normal, uniform
max_t = 100
movements = randint(0, 2, size=max_t)
y = 0
values = [y]
for movement in movements:
    if movement == 1:
        y = y + 1
    else:
        y = y - 1
    values.append(y)
```

6.3.1.1 Plot of a single integer random walk

```python
plt.xlabel('t')
plt.ylabel('y')
ax = plt.plot(values)
plt.show()
```



6.4 A random-walk as a cumulative sum

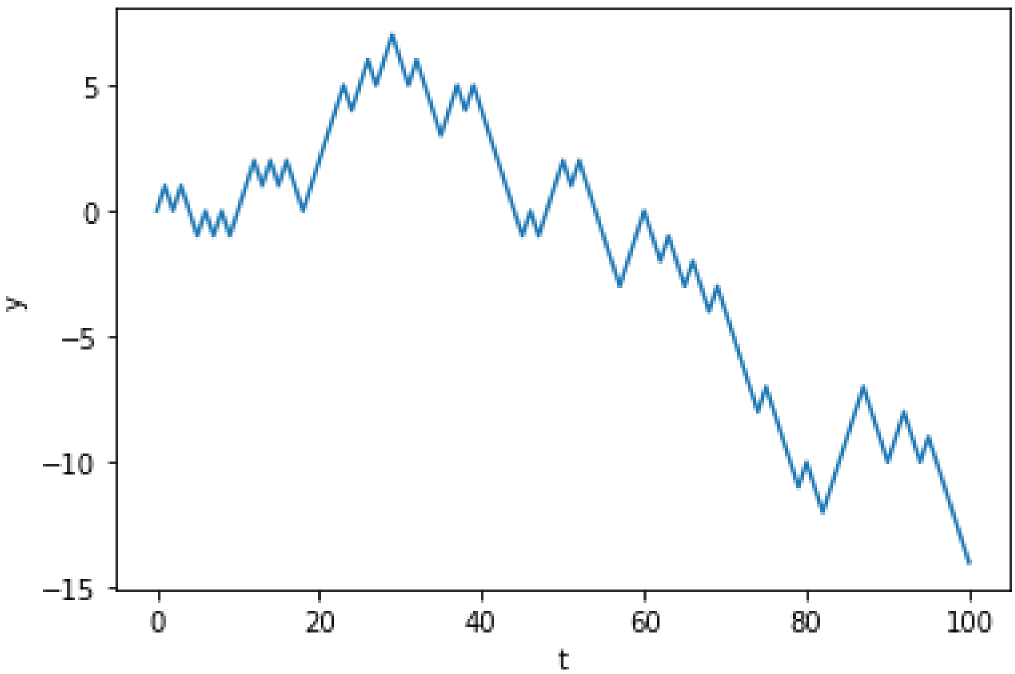
- Notice that the value of y_t is simply the cumulative sum of movements randomly chosen from -1 or $+1$.
- So if $p(\epsilon = -1) = \frac{1}{2}$ and $p(\epsilon = +1) = \frac{1}{2}$ then
- We can define our game as a simple *stochastic process* : $y_t = \sum_{t=1}^{t_{max}} \epsilon_t$
- We can use numpy's `where()` function to replace all zeros with -1.

6.4.1 an integer random-walk using an accumulator

```
t_max = 100
random_numbers = randint(0, 2, size=t_max)
steps = np.where(random_numbers == 0, -1, +1)
y = 0
values = [0]
for step in steps:
    y = y + step
    values.append(y)
```

6.4.1.1 Plot of a single integer random walk

```
plt.xlabel('t')
plt.ylabel('y')
plt.plot(values)
plt.show()
```



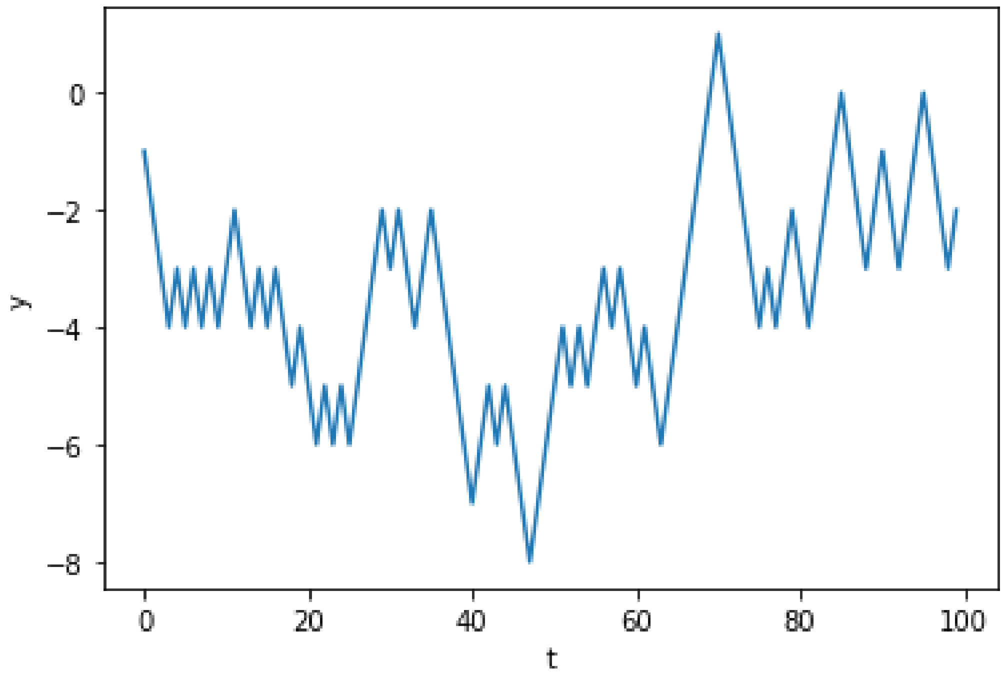
6.5 A random-walk using arrays

- We can make our code more efficient by using the `cumsum()` function instead of a loop.
- This way we can work entirely with *arrays*.
- Remember that vectorized code can be much faster than iterative code.

6.5.1 an integer random-walk using vectorization

```
# Vectorized random-walk with arrays to improve efficiency
t_max = 100
random_numbers = randint(0, 2, size=t_max)
steps = np.where(random_numbers == 0, -1, +1)
```

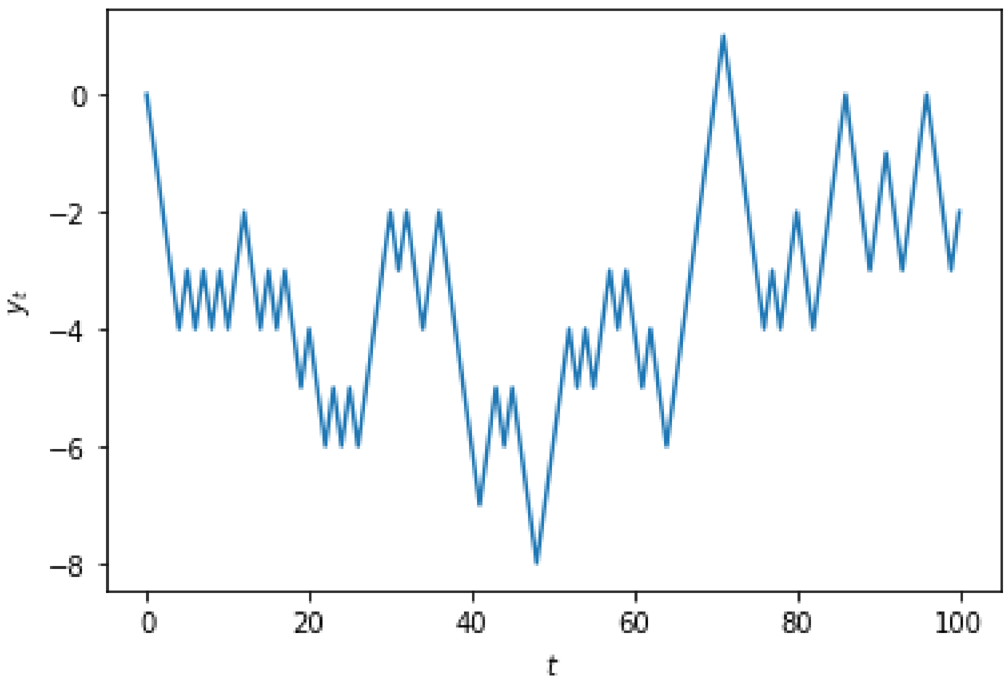
```
values = np.cumsum(steps)
plt.xlabel('t')
plt.ylabel('y')
ax = plt.plot(values)
```



6.5.2 Using concatenate to prepend the initial value

- If we want to include the initial position $y_0 = 0$, we can concatenate this value to the computed values from the previous slide.
- The `numpy.concatenate()` function takes a *single* argument containing a *sequence* of arrays, and returns a new array which contains all values in a single array.

```
plt.plot(np.concatenate(([0], values)))
plt.xlabel('$t$')
plt.ylabel('$y_t$')
plt.show()
```



6.6 Multiple realisations of a stochastic process

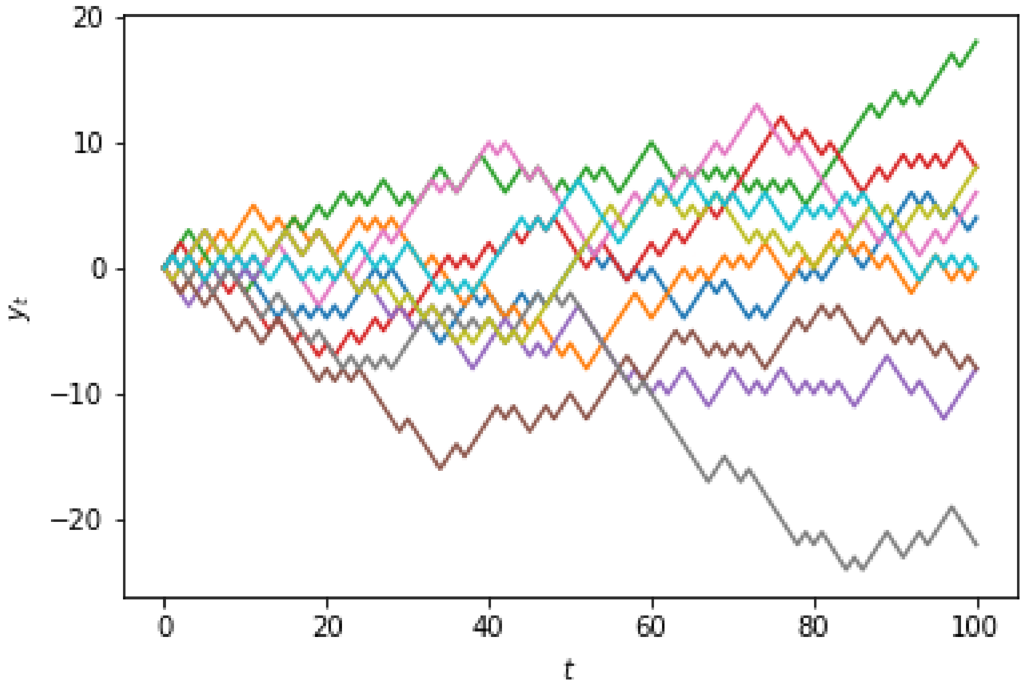
- Because we are making use of random numbers, each time we execute this code we will obtain a different result.
- In the case of a random-walk, the result of the simulation is called a *path*.
- Each path is called a *realisation* of the model.
- We can generate multiple paths by using a 2-dimensional array (a matrix).
- Suppose we want $n = 10$ paths.
- In Python we can pass two values for the size argument in the `randint()` function to specify the dimensions (rows and columns):

```
t_max = 100
n = 10
random_numbers = randint(0, 2, size=(t_max, n))
steps = np.where(random_numbers == 0, -1, +1)
```

6.7 Using `cumsum()`

We can then tell `cumsum()` to sum the rows using the `axis` argument:

```
values = np.cumsum(steps, axis=0)
values = np.concatenate((np.matrix(np.zeros(n)), values), axis=0)
plt.xlabel('$t$')
plt.ylabel('$y_t$')
ax = plt.plot(values)
```



6.8 Multiplicative Random Walks

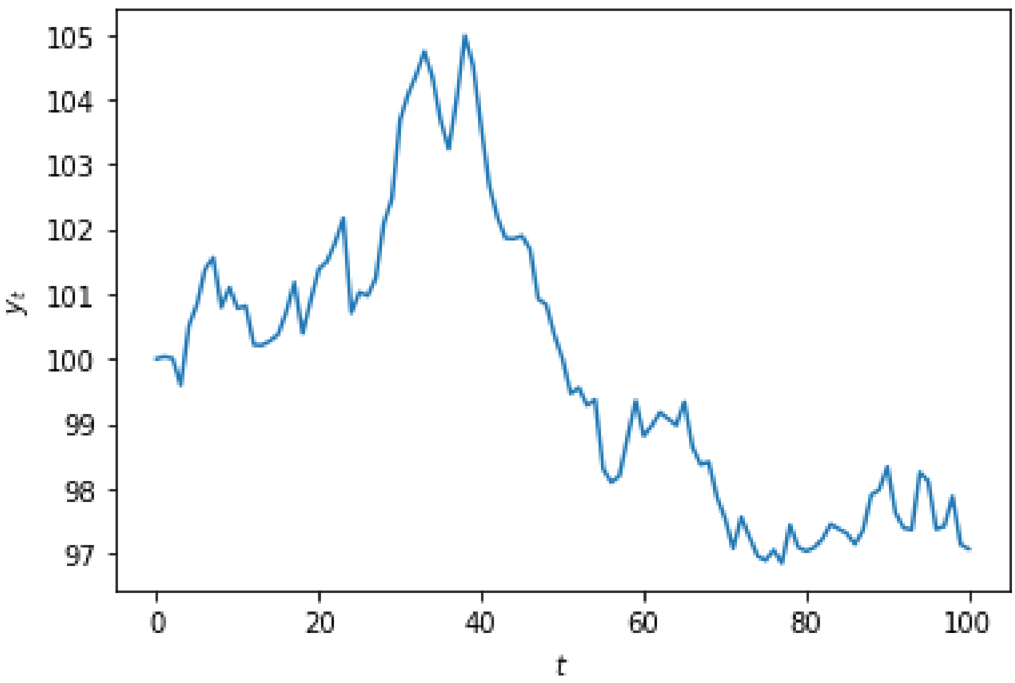
- The series of values we have looked at do *not* closely resemember how security prices change over time.
- In order to obtain a more realistic model of how prices change over time, we need to *multiply* instead of add.
- Let r_t denote an *i.i.d.* random variable distributed $r_t \sim N(0, \sigma^2)$
- Define a strictly positive intitial value $y_0 \in \mathbf{R}$; e.g. $y_0 = 10$.
- Subsequent values are given by $y_t = y_{t-1} \times (1 + r_t)$
- We can write this as a cumulative product:

$y_t = y_0 \times \Pi_{t=1}^{t_{max}} \epsilon_t$

6.9 Using `cumprod()`

- This can be computed efficiently using numpy's `cumprod()` function.

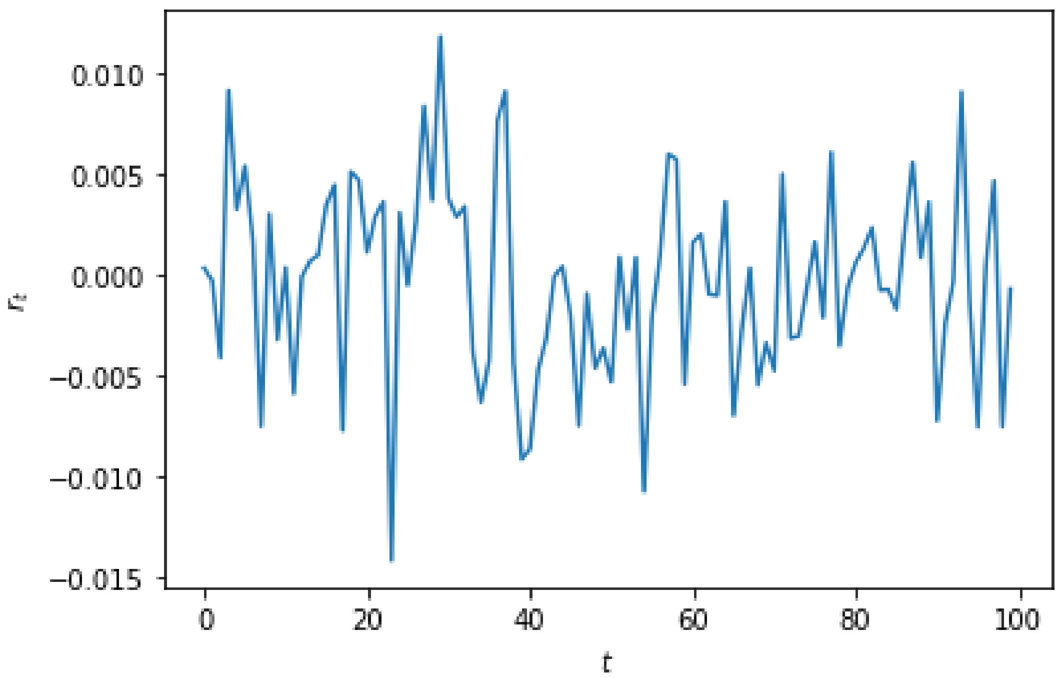
```
initial_value = 100.0
random_numbers = normal(size=t_max) * 0.005
multipliers = 1 + random_numbers
values = initial_value * np.cumprod(multipliers)
plt.xlabel('$t$')
plt.ylabel('$y_t$')
ax = plt.plot(np.concatenate(([initial_value], values)))
```



6.10 Random walk variates as a time-series

- Now let's plot the random perturbations over time

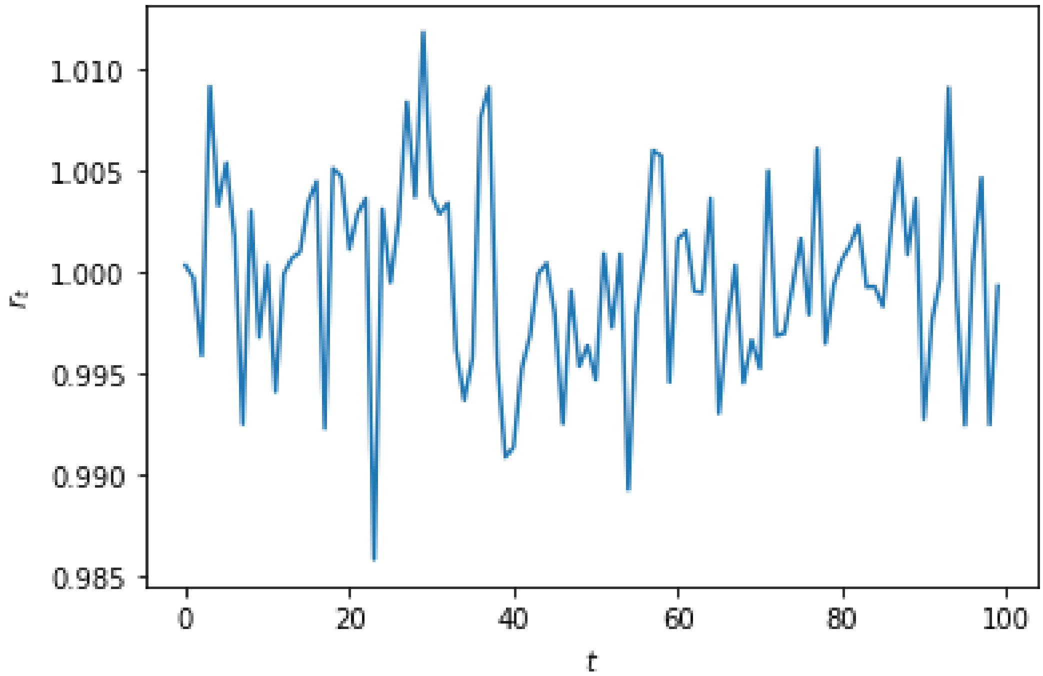
```
plt.xlabel('$t$')
plt.ylabel('$r_t$')
ax = plt.plot(random_numbers)
```



6.11 Gross returns

- If we take $100 \times \epsilon_t$, then these represent the percentage changes in the value at discrete time intervals.
- If the values represent prices that have been adjusted to incorporate dividends, then the multipliers are called simple *returns*.
- The gross return is obtained by adding 1.

```
plt.xlabel('$t$')
plt.ylabel('$r_t$')
ax = plt.plot(random_numbers + 1)
```



6.12 Continuously compounded, or log returns

- A simple return R_t is defined as

$R_t = (y_t - y_{t-1})/y_{t-1} = y_t/y_{t-1} - 1$

where y_t is the adjusted price at time t.

- The gross return is $R_t + 1$
- A *continuously* compounded return r_t, or *log-return*, is defined as:

$r_t = \log(y_t/y_{t-1}) = \log(y_t) - \log(y_{t-1})$

- In Python:

```
from numpy import diff, log

prices = values
log_returns = diff(log(prices))
```

6.13 Aggregating returns

- Simple returns aggregate across *assets*- the return on a *portfolio* of assets is the weighted average of the simple returns of its individual securities.
- Log returns arregate across *time*.
- If return in year one is $r_1 = \log(p_1/p_0) = \log(p_1) - \log(p_0)$
- and return in year two is $r_2 = \log(p_2/p_1) = \log(p_2) - \log(p_1)$,
- then return over two years is $r_1 + r_2 = \log(p_2) - \log(p_0)$

6.14 Converting between simple and log returns

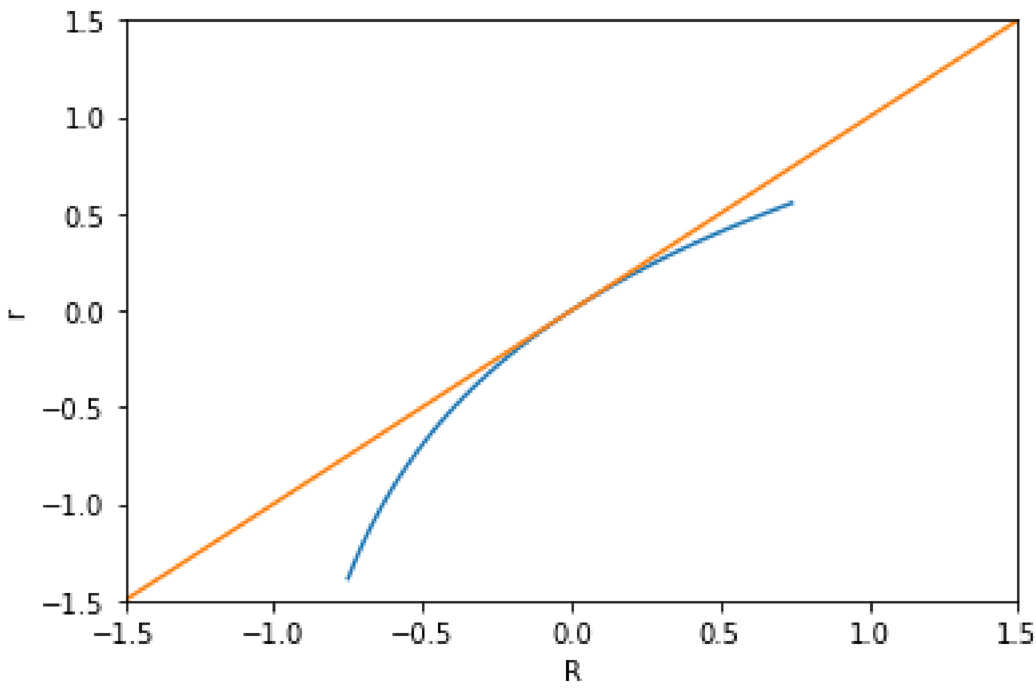
- A simple return R_t can be converted into a log-return r_t:

$r_t = \log(R_t + 1)$

6.15 Comparing simple and log returns

- For small values of r_t then $R_t \approx r_t$.
- We can examine the error for larger values:

```
simple_returns = np.arange(-0.75, +0.75, 0.01)
log_returns = np.log(simple_returns + 1)
plt.xlim([-1.5, +1.5]); plt.ylim([-1.5, +1.5])
plt.plot(simple_returns, log_returns)
x = np.arange(-1.5, 1.6, 0.1)
plt.xlabel('R'); plt.ylabel('r')
plt.plot(x, x); plt.show()
```

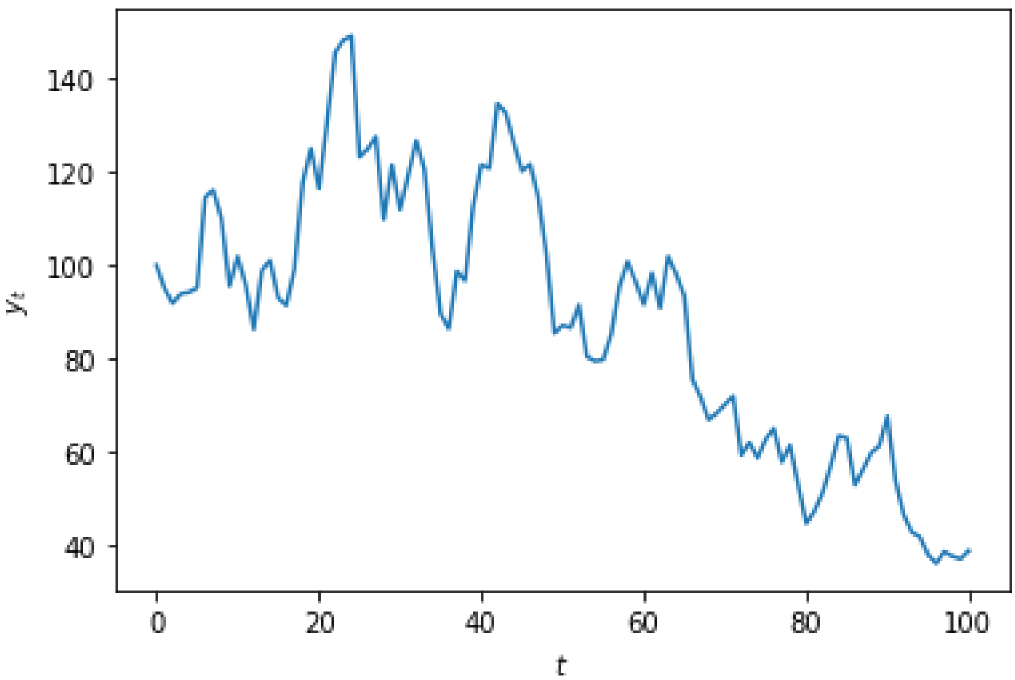


6.16 A discrete multiplicative random walk with log returns

- Let r_t denote a random i.i.d. variable distributed $r_t \sim N(0, \sigma^2)$
- Then $y_t = y_0 \times \exp(\sum_{t=1}^{t_{max}} r_t)$

6.16.1 Plotting a single realization

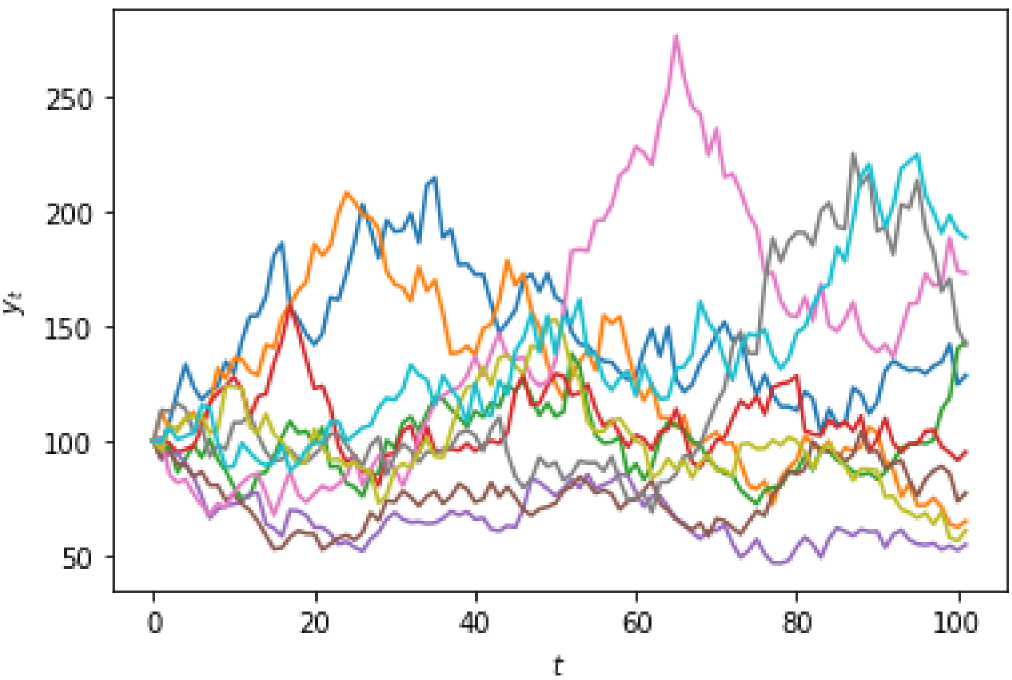
```
from numpy import log, exp, cumsum

t_max = 100
volatility = 1e-2
initial_value = 100.
r = normal(size=t_max) * np.sqrt(volatility)
y = initial_value * exp(cumsum(r))
plt.xlabel('$t$')
plt.ylabel('$y_t$')
ax = plt.plot(np.concatenate(([initial_value], y)))
```



6.17 Multiple realisations of a multiplicative random-walk

- Let's generate $n = 10$ realisations of this process:

```
def random_walk(initial_value = 100, n = 10,
                t_max = 100, volatility = 0.005):
    r = normal(size=(t_max+1, n)) * np.sqrt(volatility)
    return np.concatenate((np.matrix([initial_value] * n),
                           initial_value * exp(np.cumsum(r, axis=0))))

plt.xlabel('$t$')
plt.ylabel('$y_t$')
ax = plt.plot(random_walk(n=10))
```



6.18 Geometric Brownian Motion

- For a continuous-time process, we use Geometric Brownian Motion (GBM):

$$S_{t_k} = S_0 \prod_{i=1}^{k} Y_i \tag{6.1}$$

$$\begin{aligned} Y_i &= \exp(\sigma\sqrt{\Delta_t} z_i + \mu\Delta_t) && (6.2) \\ &= \exp(\sigma\sqrt{\Delta_t} z_i + (\bar{r} - \frac{\sigma^2}{2})\Delta_t) && (6.3) \\ & && (6.4) \end{aligned}$$

- As a cumulative sum:

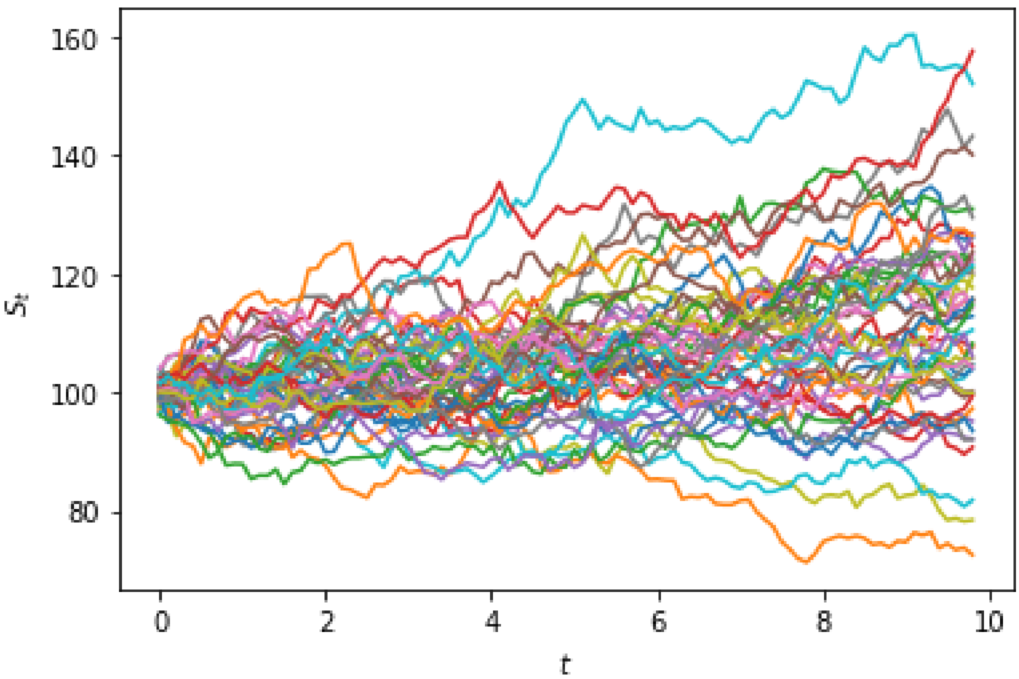
$$S_{t_k} = S_0 \times \exp(\sum_{i=1}^{k} \sigma\sqrt{\Delta_t} z_i + (\bar{r} - \frac{\sigma^2}{2})\Delta_t) \tag{6.5}$$

6.18.1 GBM with multiple paths in Python

```
def gbm(sigma, r, k, t_max, S0, I=1):
    z = np.random.normal(size=(k-1, I))
    dt = t_max/k
    y = sigma * np.sqrt(dt)*z + (r - sigma**2 / 2.) * dt
    return S0 * np.exp(np.cumsum(y, axis=0))
```

6.18.1.1 Plotting multiple realizations of GBM

```python
sigma = 0.05; r = 0.01; k = 100; t_max = 10.; S0 = 100.
T = np.arange(0, t_max - t_max/k, t_max/k)
ax = plt.plot(T, gbm(sigma, r, k, t_max, S0, I=50))
plt.xlabel('$t$'); plt.ylabel('$S_t$'); plt.show()
```



7 Monte-Carlo simulation for option pricing

7.1 Options

- An option gives the right to buy (call) or sell (put) an underlying stock at a prespecified price, called the *strike price*, at a specified date, or period.
  - A European option specifies a single date.
  - An American option specifies a period.
- Options can also specify an index, in which case they are settled in cash.
- People selling options are called option writers.
- People buying options are the option holders.

7.2 Payoff to an option holder

- The payoff to an option holder depends on the index price S_t when the option is exercised.
- For a European option with strike price K, and maturity date T:
  - If the index is below the strike, the option is worthless.
  - Otherwise the holder receives the difference between the index price and the strike price: $S_T - K$.
- Therefore the payoff to the option is:

$$\max(S_T - K, 0). \tag{7.1}$$

7.3 Outcomes

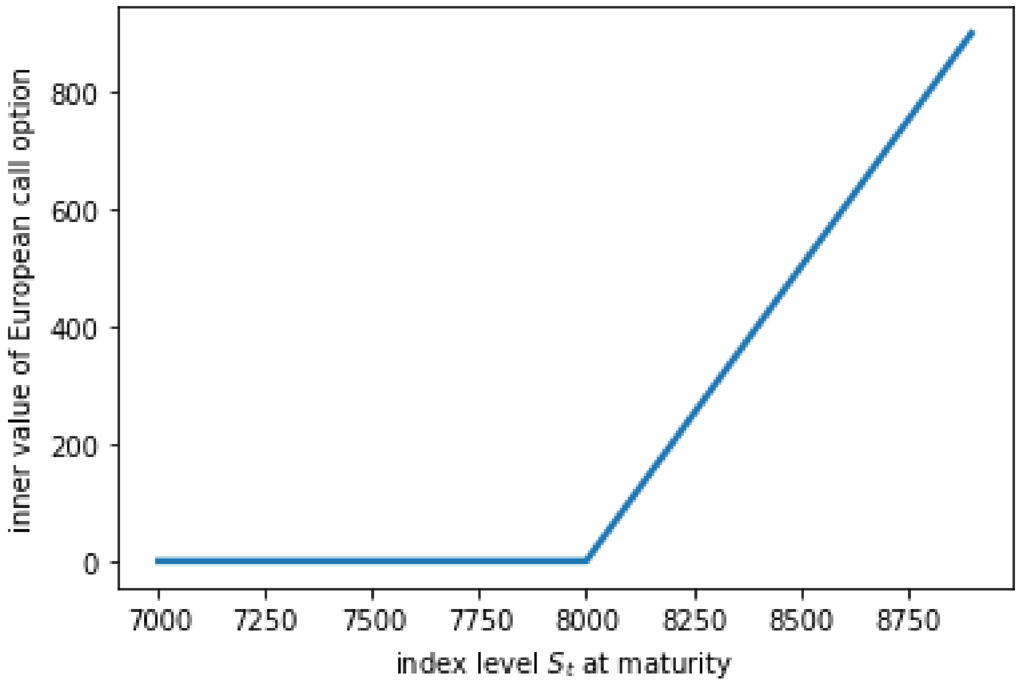
- In-the-money (ITM):
  - a call (put) is in-the-money if $ S > K (S < K) $.
- At-the-money (ATM):
  - an option is at-the-money if $S \approx K$.
- Out-of-the-money (OTM):
  - a call (put) is out-of-the-money if $ S < K (S > K) $.

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

K = 8000 # Strike price
S = np.arange(7000, 9000, 100)  # index level values
h = np.maximum(S - K, 0)  # inner values of call option
```

7.3.1 Plotting the payoff function

```python
plt.plot(S, h, lw=2.5)  # plot inner values at maturity
plt.xlabel('index level $S_t$ at maturity'); plt.ylabel('inner
    ↪ value of European call option')
```

```
Text(0, 0.5, 'inner value of European call option')
```



7.4 Parameters which affect the inner-value

- Initial price level of the index S_0.
- Volatility of the index σ.
- The return(s) of the index.
- Time-to-maturity T.

7.5 Risk-neutral assumptions

When all of the following assumptions hold:

- no arbitrage,
- complete markets (no transaction costs and perfect information),
- law of one price (assets with identical risk and return have a unique price)

7.6 Risk-neutral parameters

we can price options without having to take account of investors' risk-preferences using only the following parameters:

- Initial price level of the index S_0.
- Volatility of the index σ.
- Time-to-maturity T.
- The risk-free rate r.

7.7 Monte-Carlo option valuation

- For a European option, the inner-value is not *path-dependent*.
- Under risk-neutral pricing, we use Geometric Brownian Motion (GBM) with:

$$S_T = S_0 \times \exp((r - \frac{\sigma^2}{2})T + \sigma\sqrt{T}z) \tag{7.2}$$

- where r is the risk-free rate.

7.8 Non-path dependent algorithm to estimate the inner-value:

1. Draw I random numbers $z_1, z_2, \ldots, z_I$ from the standard normal distribution.
2. For $i \in \{1, 2, \ldots, I\}$:

   i). Calculate index level at maturity by simulating geometric Brownian motion using the above equation.

   ii). Compute the inner-value of the option $h_T = \max(S_T - K, 0)$.

   iii). Discount back to the present at the risk-free rate r, giving the present value:

$$C_i = e^{-rT} h_T. \tag{7.3}$$

3. Output the final estimate by computing the Monte-Carlo estimator $\bar{\mathbf{C}} = \frac{\sum_{i=1}^{I} C_i}{I}$

7.9 Monte-Carlo valuation of European call option in Python

In the following we use the following parameterization of the model: initial index price $S_0 = 100$, strike price $K = 105$, time-to-maturity $T = 1$, risk-free rate $r = 0.02$, index volatility $\sigma = 0.02$, number of independent realizations $I = 10^5$.

```python
from numpy import sqrt, exp, cumsum, sum, maximum, mean
from numpy.random import standard_normal

# Parameters
S0 = 100.; K = 105.; T = 1.0
r = 0.02; sigma = 0.1; I = 100000

# Simulate I outcomes
S = S0 * exp((r - 0.5 * sigma ** 2) * T + sigma * sqrt(T) *
                          standard_normal(I))

# Calculate the Monte Carlo estimator
```

```
C0 = exp(-r * T) * mean(maximum(S - K, 0))
print("Estimated present value is %f" % C0)
```

```
Estimated present value is 2.754300
```

7.10 Asian (average-value) option

- The payoff of an asian option is determined by the *average* of the price of the underlying over a pre-defined period of time.
- Payoff for a fixed-strike Asian call option:

$$C_T = \max(A(0,T) - K, 0) \tag{7.4}$$

where:

$$A(0,T) = \frac{1}{T}\int_0^T S_t dt \tag{7.5}$$

If we let $t_i = i \times \frac{T}{n}$ $i \in 0, 1, 2, \ldots, n$:

$$A(0,T) \approx \frac{1}{n}\sum_{i=0}^{n-1} S_{t_i} \tag{7.6}$$

- The payoff is *path dependent*, and therefore we need to simulate intermediate values of S_t.

7.11 Path-dependent Monte-Carlo option pricing

- To simulate GBM at M evenly spaced time intervals t_i with $\Delta_T = T/M$:

$$S_{t_k} = S_0 \times \exp(\sum_{i=1}^{k} \sigma\sqrt{\Delta_t} z_i + (r - \frac{\sigma^2}{2})\Delta_t) \tag{7.7}$$

7.12 Algorithm for path-dependent option pricing

1. Draw $I \times M$ random numbers from the standard normal distribution.
2. For $i \in \{1, 2, \ldots, I\}$:

   i). Calculate index level at times $t_i \in \{\Delta_T, 2\Delta_T, \ldots, T\}$ by simulating geometric Brownian motion with drift $\mu = r$ and volatility σ using the equation for S_{t_k}.

   ii). Estimate the inner-value of the option $\hat{h}_T = \frac{1}{T}\sum_{i=1}^{M} S_{t_i}$.

   iii). Discount back to the present at the risk-free rate r, giving the present value:

$$C_i = e^{-rT}\hat{h}_T. \tag{7.8}$$

3. Output the final estimate by computing the Monte-Carlo estimator $\bar{\mathbf{C}} = \frac{\sum_{i=1}^{I} C_i}{I}$

7.12.1 Monte-Carlo valuation of Asian fixed-strike call option in Python

In the following we use the following parameterization of the model: initial index price $S_0 = 100$, time-to-maturity $T = 1$, number of time-steps $M = 200$, risk-free rate $r = 0.02$, index volatility $\sigma = 0.1$, number of independent realizations $I = 10^5$.

```python
from numpy import sqrt, exp, cumsum, sum, maximum, mean
from numpy.random import standard_normal

# Parameters
S0 = 100.; T = 1.0; r = 0.02; sigma = 0.1
M = 200; dt = T / M; I = 100000

def inner_value(S):
    """ Inner value for a fixed-strike Asian call option """
    return mean(S, axis=0)

# Simulate I paths with M time steps
S = S0 * exp(cumsum((r - 0.5 * sigma ** 2) * dt + sigma * sqrt(dt) *
                            standard_normal((M + 1, I)), axis=0))

# Calculate the Monte Carlo estimator
C0 = exp(-r * T) * mean(inner_value(S))
print("Estimated present value is %f" % C0)
```

```
Estimated present value is 99.052181
```

8 Estimating Value-At-Risk (VaR) in Python

8.1 Value-at-Risk (VaR)

- Value at risk (VaR) is a methodology for computing a risk measurement on a portfolio of investments.
- It is defined over:
  - a duration of time, e.g. one day.
  - a confidence level (or equivalent percentage) α.
- The VaR_α over duration T is the maximum possible loss during T, excluding outcomes whose probability is less than α, according to our model.

8.2 Value-at-Risk (VaR)

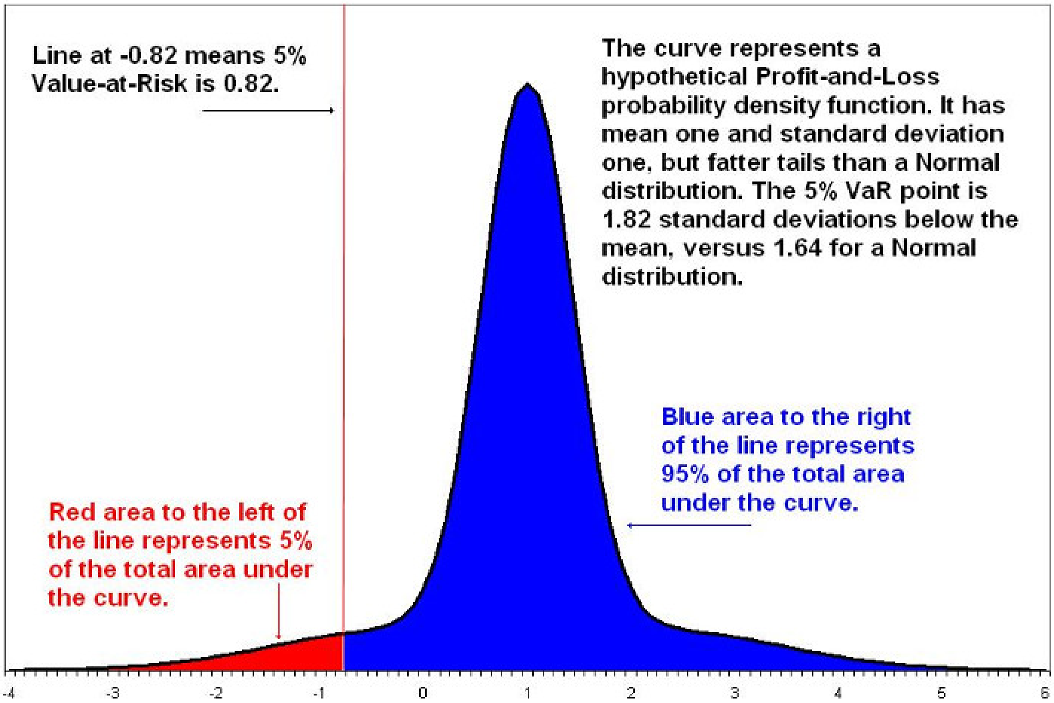


Figure 8.1: VaR_diagram

8.3 Mathematical definition

- Value at Risk with confidence α can be defined

$$VaR_\alpha(X) = \min\{x \in \mathbb{R} : 1 - F_X(-x) \geq \alpha\} \tag{8.1}$$

where X is a random variable representing the value of the portfolio, with cumulative distribution function F_X.

- The $VaR_\alpha(X)$ is simply the negative of the α-quantile.
- We typically assume mark-to-market accounting, and so the value of the portfolio is determined from fair market *prices*.

8.4 Quantiles, Quartiles and Percentiles

0 quartile = 0 quantile = 0 percentile

1 quartile = 0.25 quantile = 25 percentile

2 quartile = .5 quantile = 50 percentile (median)

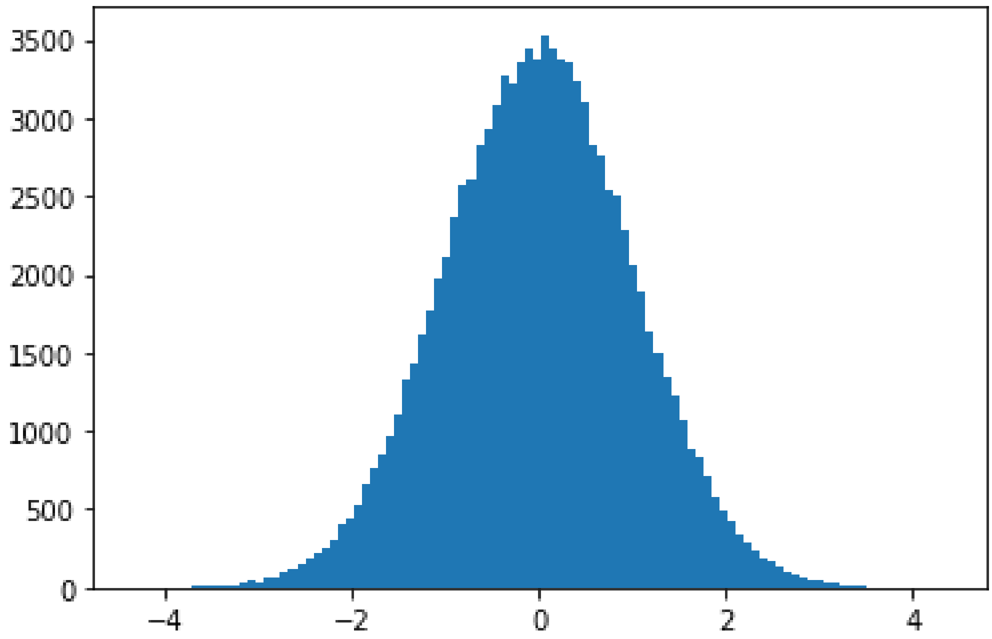
3 quartile = .75 quantile = 75 percentile

4 quartile = 1 quantile = 100 percentile

8.5 Computing quantiles in Python

- First we will generate some random data.

```
import numpy as np

# Draw values sampled iid from standard normal distribution
data = np.random.normal(size=100000)
import matplotlib.pyplot as plt
ax = plt.hist(data, bins=100)
plt.show()
```



8.6 Computing quantiles in Python

```
# Compute the 5th-percentile
np.percentile(data, q=5)
```

```
-
1.6548195402636892
```

8.7 Computing several percentiles

```
for p in range(1, 6):
    print("The %d-percentile is %f" % (p, np.percentile(data, q=p)))
```

```
The 1-percentile is -2.350552
The 2-percentile is -2.072378
The 3-percentile is -1.891520
The 4-percentile is -1.763699
```

```
The 5-percentile is -1.654820
```

8.8 Estimating VaR

- The VaR depends on the distribution of a random variable, e.g. the price of an index, over a specified period of time.
- How can we estimate the quantiles of this distribution?

8.9 Estimating VaR

Common methods:

- Variance/Covariance method.
- Historical simulation- bootstrap from historical data.
- Monte-Carlo simulation.

8.10 Historical simulation

To calculate $VaR_\alpha(X)$ with sampling interval Δ_t over $T = n \times \Delta_t$ using a total of N bootstrap samples:

1. Assuming that the returns are stationary over the entire period, obtain a large sample of historical prices for the components of the portfolio or index.
2. Convert the prices into returns with frequency $1/\Delta_t$.
3. For every $i \in \{1, \ldots, N\}$:
   - Randomly choose n returns $r_1, r_2, \ldots, r_n$ with replacement.
   - Compute $P_i|(r_1, r_2, \ldots, r_n)$ - the profit and loss of the investment given these returns.
4. Compute $Q(\alpha)$ from the sample $\mathbf{P}$, where Q is the quantile function.

8.11 Random choices in Python

- We can use the function `choice()` from the `numpy` module to choose randomly from a set of values.
- To choose a single random value:

```python
import numpy as np
data = np.random.randint(1, 6+1, size=20)
data
```

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

```python
np.random.choice(data, replace=True)
```

```
4
```

```python
np.random.choice(data, replace=True)
```

```
1
```

8.12 Generating a sequence of choices

- We can think of this as a simple bootstrap model of a dice:

```
np.random.choice(data, size=5, replace=True)
```

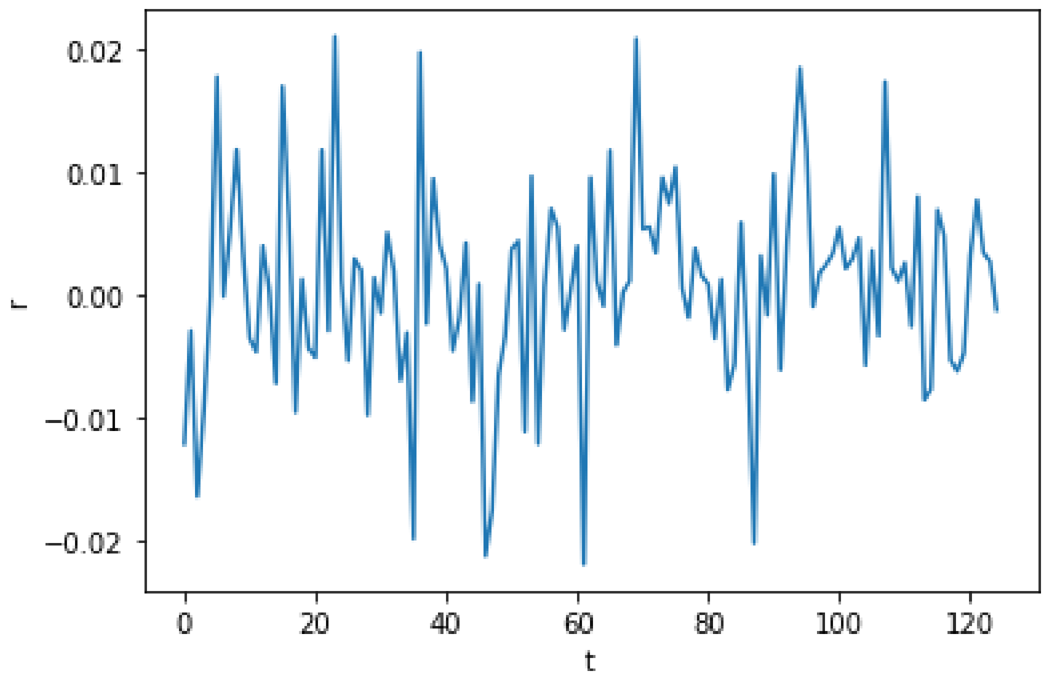
```
array([1, 4, 3, 5, 1])
```

8.13 Bootstrapping from empirical data

- Typically we will collect real-world (empirical) data from a random process whose true distribution is unknown.
- In Finance, we can bootstrap from historical returns.

8.14 Obtaining returns for the Nikkei 225 index

```
import pandas as pd
n225 = pd.read_csv('data/^N225.csv')
n225.set_index('Date', inplace=True)
returns = np.diff(np.log(n225["Adj Close"]))
plt.plot(returns)
plt.xlabel('t')
plt.ylabel('r')
plt.show()
```



8.15 Simulating returns

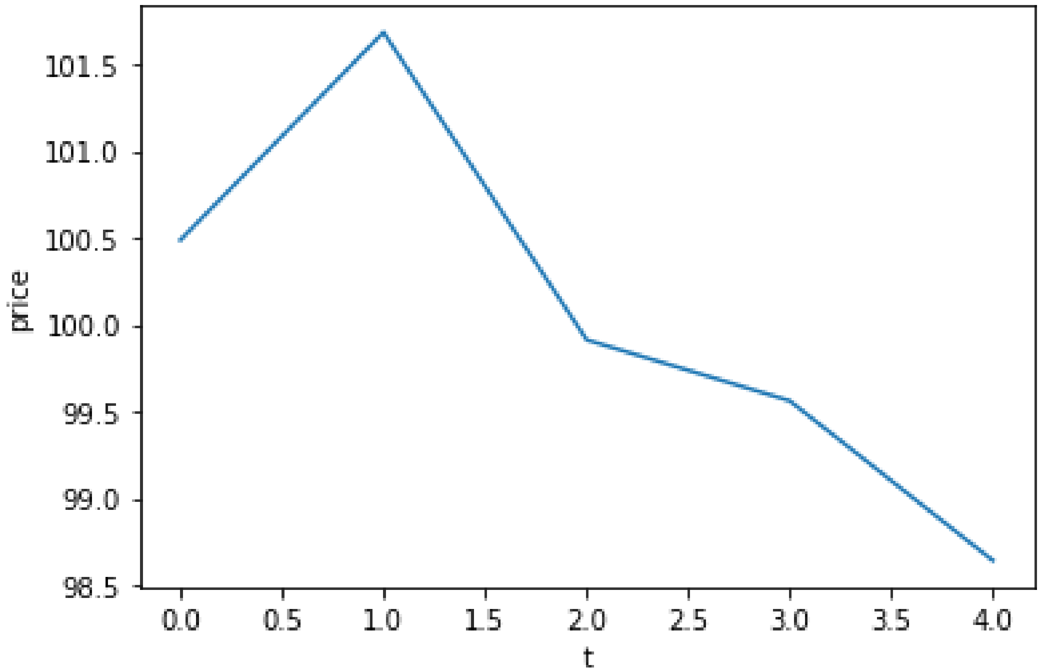
- We will now simulate the returns over the next five days:

```
num_days = 5
simulated_returns = \
    np.random.choice(returns, size=num_days, replace=True)
simulated_returns
```

```
array([ 0.00485614,  0.01182755, -0.01755096, -0.0035224 , -
0.00928068])
```

8.16 Simulating prices

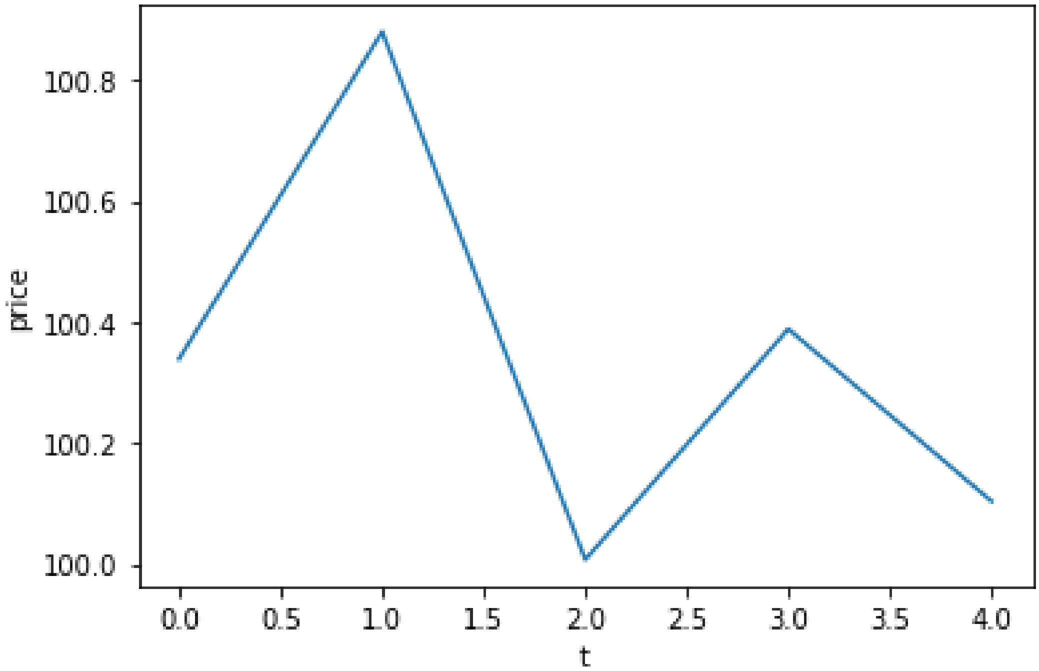
```
initial_price = 100.
prices = initial_price * np.exp(np.cumsum(simulated_returns))
plt.plot(prices)
plt.xlabel('t')
plt.ylabel('price')
plt.show()
```



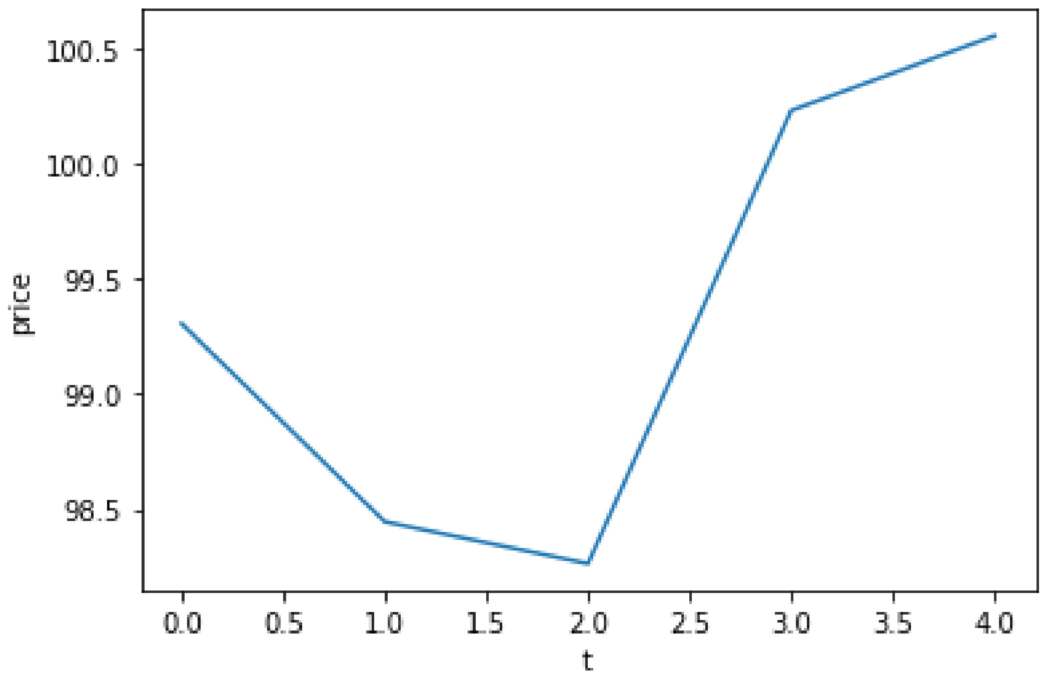
- If we perform this simulation again, will we obtain the same result?

```
num_days = 5
simulated_returns = \
    np.random.choice(returns, size=num_days, replace=False)
prices = initial_price * np.exp(np.cumsum(simulated_returns))
plt.plot(prices)
plt.xlabel('t')
plt.ylabel('price')
```

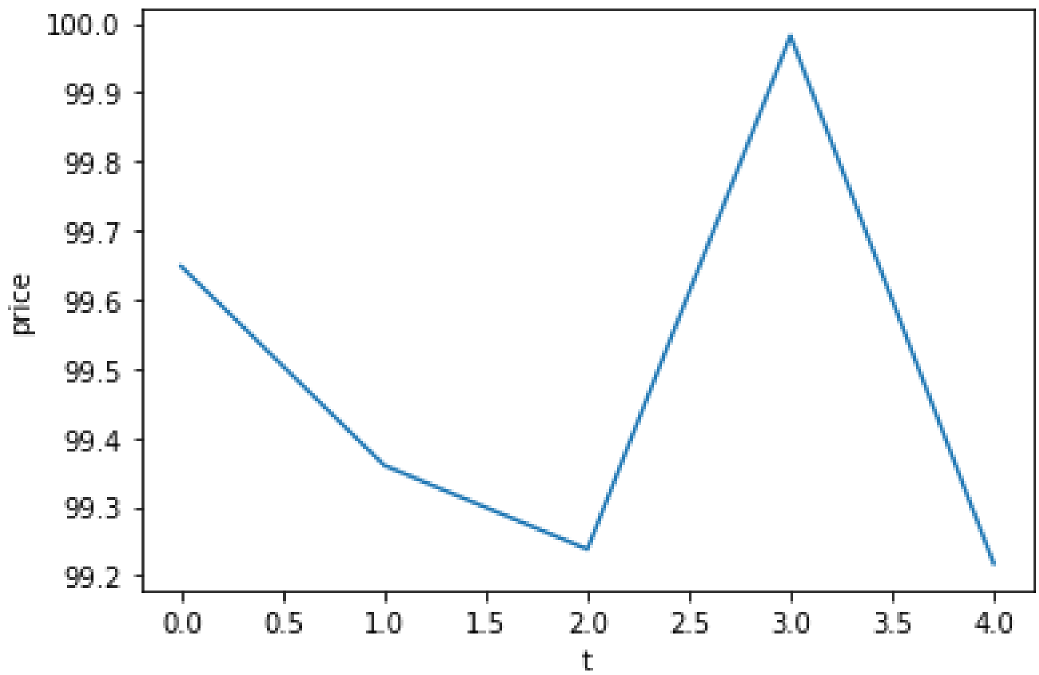
```
plt.show()
```




```
num_days = 5
simulated_returns = \
    np.random.choice(returns, size=num_days, replace=False)
prices = initial_price * np.exp(np.cumsum(simulated_returns))
plt.plot(prices)
plt.xlabel('t')
plt.ylabel('price')
plt.show()
```

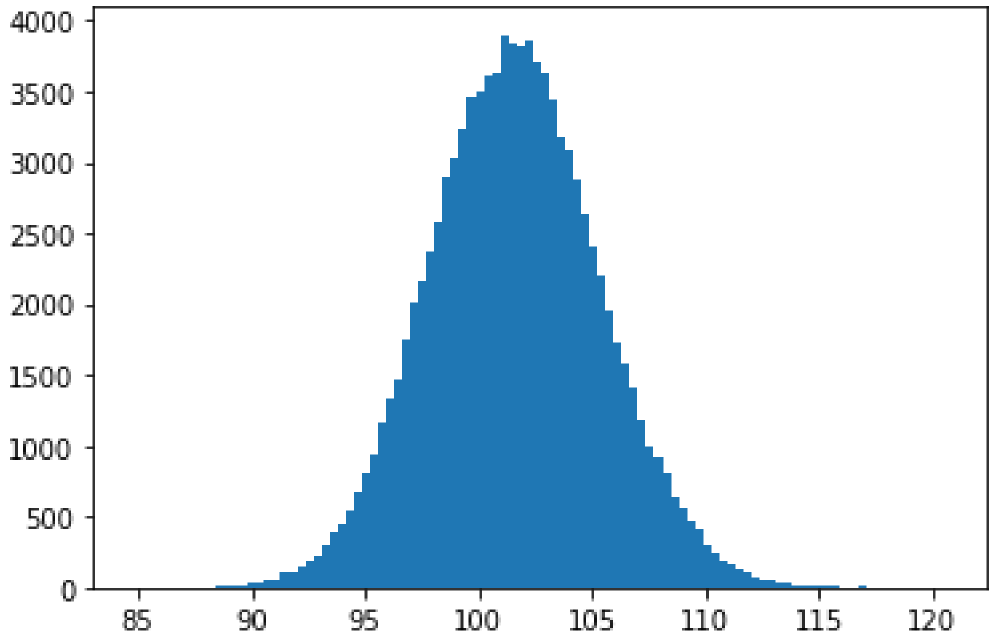



```
num_days = 5
simulated_returns = \
    np.random.choice(returns, size=num_days, replace=False)
prices = initial_price * np.exp(np.cumsum(simulated_returns))
plt.plot(prices)
plt.xlabel('t')
_ = plt.ylabel('price')
```



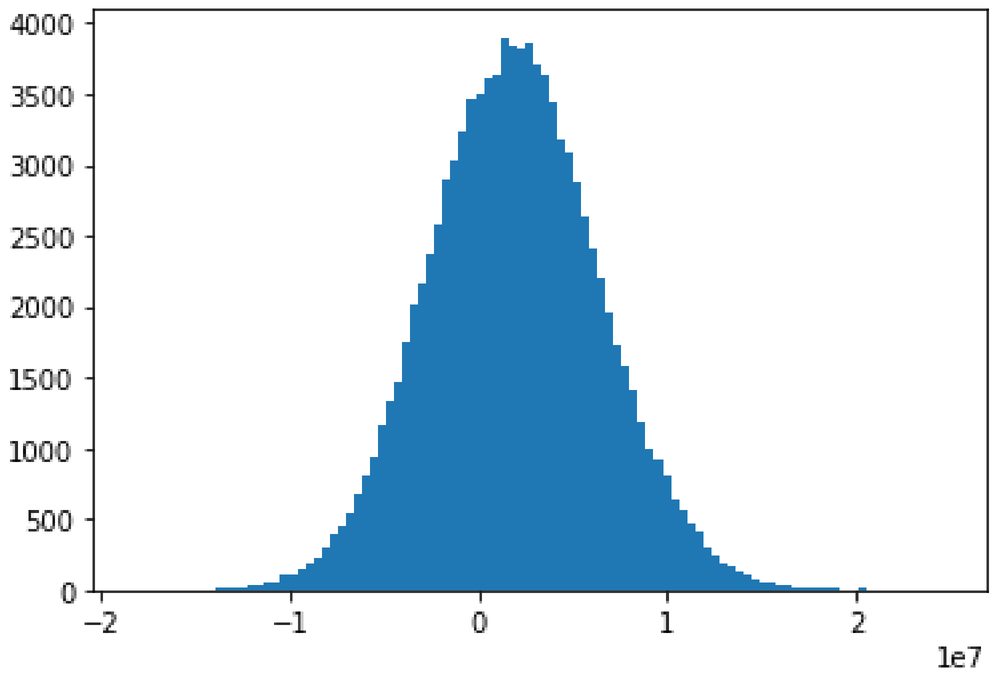
8.17 The distribution of the final price

```
def final_price():
    num_days = 20
    simulated_returns = \
        np.random.choice(returns, size=num_days, replace=True)
    prices = initial_price * np.exp(np.cumsum(simulated_returns))
    return prices[-1]

num_samples = 100000
prices = [final_price() for i in range(num_samples)]
plt.hist(prices, bins=100)
plt.show()
```



8.18 The distribution of the profit and loss

```
def profit_and_loss(final_price):
    return 1200000. * (final_price - initial_price)

p_and_l = np.vectorize(profit_and_loss)(prices)
plt.hist(p_and_l, bins=100)
plt.show()
```



8.19 The quantiles of the profit and loss

```
for p in range(1, 6):
    print("The %.2f-quantile is %.4f" % (p/100., np.percentile(
    ↪ p_and_l, q=p)))
```

```
The 0.01-quantile is -8359991.6042
The 0.02-quantile is -7165047.1254
The 0.03-quantile is -6399125.5624
The 0.04-quantile is -5837986.8979
The 0.05-quantile is -5348176.8969
```

What is the 5% Value-At-Risk?

```
var = -1 * np.percentile(p_and_l, q=5)
print("5%%-VaR is %.4f" % var)
```

```
5%-VaR is 5348176.8969
```